ROYAL INSTITUTE
OF TECHNOLOGY

A System, Tools and Algorithms for Adaptive HTTP-live Streaming on Peer-to-peer Overlays

ROBERTO ROVERSO

Doctoral Thesis in
Information and Communication Technology
KTH - Royal Institute of Technology
Stockholm, Sweden, December 2013

TRITA-ICT/ECS AVH 13:18
ISSN 1653-6363
ISRN KTH/ICT/ECS/AVH-13/18-SE
ISBN 978-91-7501-915-4

KTH School of Information and
Communication Technology
SE-100 44 Stockholm
SWEDEN

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av Doktorandexamen i Informations- och Kommunikationsteknik Torsdagen den 12 Dec 2013 klockan 13.00 i Sal D, ICT School, Kungl Tekniska högskolan, Forum 105, 164 40 Kista, Stockholm.

Tryck: Universitetsservice US AB

Abstract

In recent years, adaptive HTTP streaming protocols have become the de facto standard in the industry for the distribution of live and video-on-demand content over the Internet. In this thesis, we solve the problem of distributing adaptive HTTP live video streams to a large number of viewers using peer-to-peer (P2P) overlays. We do so by assuming that our solution must deliver a level of quality of user experience which is the same as a CDN while trying to minimize the load on the content provider's infrastructure. Besides that, in the design of our solution, we take into consideration the realities of the HTTP streaming protocols, such as the pull-based approach and adaptive bitrate switching.

The result of this work is a system which we call SmoothCache that provides CDN-quality adaptive HTTP live streaming utilizing P2P algorithms. Our experiments on a real network of thousands of consumer machines show that, besides meeting the the CDN-quality constraints, SmoothCache is able to consistently deliver up to 96% savings towards the source of the stream in a single bitrate scenario and 94% in a multi-bitrate scenario. In addition, we have conducted a number of pilot deployments in the setting of large enterprises with the same system, albeit tailored to private networks. Results with thousands of real viewers show that our platform provides an average offloading of bottlenecks in the private network of 91.5%.

These achievements were made possible by advancements in multiple research areas that are also presented in this thesis. Each one of the contributions is novel with respect to the state of the art and can be applied outside of the context of our application. However, in our system they serve the purposes described below.

We built a component-based event-driven framework to facilitate the development of our live streaming application. The framework allows for running the same code both in simulation and in real deployment. In order to obtain scalability of simulations and accuracy, we designed a novel flow-based bandwidth emulation model.

In order to deploy our application on real networks, we have developed a network library which has the novel feature of providing on-the-fly prioritization of transfers. The library is layered over the UDP protocol and supports NAT Traversal techniques. As part of this thesis, we have also improved on the state of the art of NAT Traversal techniques resulting in higher probability of direct connectivity between peers on the Internet.

Because of the presence of NATs on the Internet, discovery of new peers and collection of statistics on the overlay through peer sampling is problematic. Therefore, we created a peer sampling service which is NAT-aware and provides one order of magnitude fresher samples than existing peer sampling protocols.

Finally, we designed SmoothCache as a peer-assisted live streaming system based on a distributed caching abstraction. In SmoothCache, peers retrieve video fragments from the P2P overlay as quickly as possible or fall back to the source of the stream to keep the timeliness of the delivery. In order to produce savings, the caching system strives to fill up the local cache of the peers ahead of playback by prefetching content. Fragments are efficiently distributed by a self-organizing overlay network that takes into account many factors such as upload bandwidth capacity, connectivity constraints, performance history and the currently being watched bitrate.

Acknowledgements

I am extremely grateful to Sameh El-Ansary for the way he supported me during my PhD studies. This thesis would not have been possible without his patient supervision and constant encouragement. His clear way of thinking, methodical approach to problems and enthusiasm are of great inspiration to me. He also taught me how to do research properly given extremely complex issues and, most importantly, how to make findings clear for other people to understand.

I would like to acknowledge my supervisor Seif Haridi for giving me the opportunity to work under his supervision. His vast knowledge of the field and experience was of much help to me. In particular, I take this opportunity to thank him for his guidance on the complex task of combining the work as a student and as an employee of Peerialism AB.

I am grateful to Peerialism AB for funding my studies and to all the company's very talented members for their help: Riccardo Reale, Mikael Högqvist, Johan Ljungberg, Magnus Hedbeck, Alexandros Gkogkas, Andreas Dahlström, Nils Franzen, Amgad Naiem, Mohammed El-Beltagy, Magnus Hylander and Giovanni Simoni. Each one of them has contributed to this work in his own way and has made my experience at the company very fruitful and enjoyable.

I would also like to acknowledge my colleagues at the Swedish Institute of Computer Science and at the Royal Institute of Technology-KTH: Cosmin Arad, Tallat Mahmood Shaffaat, Joel Höglund, Vladimir Vlassov, Ahmad Al-Shishtawy, Niklas Ekström, Fatemeh Rahimian, Amir Payberah and Sverker Janson. In particular, I would like to express my gratitude to Jim Dowling for providing me with invaluable feedback on the topics of my thesis and passionate support of my work.

A special thanks goes to all the people that have encouraged me in the last years and have made my life exciting and cherishable: Christian Calgaro, Stefano Bonetti, Tiziano Piccardo, Matteo Lualdi, Jonathan Daphy, Andrea Ferrario, Sofia Kleban, to name a few and, in particular, Maren Reinecke for her love and continuous moral support.

Finally, I would like to dedicate this work to my parents, Sergio and Odetta, and close relatives, Paolo and Ornella, who have, at all times, motivated and helped me in the journey of my PhD.

To my family

This work was conducted at and funded by

PEERIALISM

CONTENTS

LIST OF FIGURES

Introduction

Mesmerizer: An Effective Tool for a Complete Peer-to-peer Software Development Life-cycle

Accurate and Efficient Simulation of Bandwidth Dynamics for Peer-To-Peer Overlay Networks

NATCracker: NAT Combinations Matter

DTL: Dynamic Transport Library for Peer-To-Peer Applications

Through the Wormhole: Low Cost, Fresh Peer Sampling for the Internet

SmoothCache: HTTP-Live Streaming Goes Peer-To-Peer

SmoothCache 2.0: CDN-quality adaptive HTTP live streaming on P2P overlays

SmoothCache Enterprise: Adaptive HTTP Live Streaming in Large Private Networks

LIST OF TABLES

Mesmerizer: An Effective Tool for a Complete Peer-to-peer Software Development Life-cycle

Accurate and Efficient Simulation of Bandwidth Dynamics for Peer-To-Peer Overlay Networks

NATCracker: NAT Combinations Matter

DTL: Dynamic Transport Library for Peer-To-Peer Applications

Through the Wormhole: Low Cost, Fresh Peer Sampling for the Internet

SmoothCache: HTTP-Live Streaming Goes Peer-To-Peer

Part I

Thesis Overview

CHAPTER 1

INTRODUCTION

Video streaming constitutes 29% of the Internet traffic worldwide and, as of 2013, it is expected to increase four times in volume in the next 3 years [1].

The most common way to distribute video on the Internet is by using a Content Delivery Network (CDN) and adaptive HTTP streaming protocols. A CDN is a collection of servers placed in data centers geographically distributed across the Internet. CDNs not only cache and distribute video but also most of the content on the Internet, such as websites, music and social networks. Adaptive HTTP streaming consists of a set of protocols which all utilize HTTP as a transport protocol [2][3][4][5]. All these protocols are based on a pull model where the player pulls content over HTTP at a rate it deems suitable. This differs substantially from traditional streaming protocols such as the RTSP/RTP that are based on a push model, where the player requests a certain stream and then the server pushes content over UDP to the player controlling the speed of the delivery. On top of that, HTTP streaming protocols have been designed to support an adaptive bitrate mode of operation, which makes the stream available in a number of bitrates. All major actors in the online broadcasting business, such as Microsoft, Adobe and Apple, have developed technologies which embrace HTTP streaming and the concept of adaptive bitrate switching as the main approach for broadcasting. HTTP live has been adopted by content services providers and creators like Netflix, Hulu and the BBC with support across all platforms and OSs, including desktop computers, tablets and smart phones.

Adaptive HTTP content is usually generated by a provider and then pushed across the CDN network to servers at locations that are geographically closer to the consumer of the content. For instance, CDNs install servers inside Internet Service Providers (ISPs) which constitute the lower end of the Internet infrastructure and provide end-users with access to the web.

CDNs serve three main purposes. First, they improve quality of user experience (QoE) by letting users access content quicker. They do so by decreasing the physical distance, i.e. number of hops and latency, the content has to travel before reaching the user. That translates in the case of adaptive HTTP streaming to shorter startup time, lower delay from the live point and lower probability of playback interruptions. Besides that, CDNs can improve throughput towards the user and therefore enable the playback of the video at higher quality or bitrate.

Second, the presence of a CDN reduces the amount of traffic on the content provider's infrastructure, which results in significant cost savings for the video provider on hardware and bandwidth capacity, which are notoriously expensive.

Third, CDNs can improve locality of content and therefore offload bottlenecks and reduce peering costs. By placing CDN nodes inside ISPs, for example, it possible to reduce the amount of incoming traffic from other ISPs as content will reach the CDN node first and then be distributed to users in the ISP rather than all users having to retrieve content from neighboring or distant ISPs.

The major drawback of the CDN approach is that building an infrastructure of multiple nodes geographically distributed across the globe is extremely expensive and cumbersome to administer and maintain. This is particularly challenging in the case of live video streaming, where, for popular events, the number of users accessing that same infrastructure can be extremely large.

An alternative to CDNs is to use peer-to-peer (P2P) as a mean to exploit resources that are readily available in the network to distribute live streams to a large number of users. P2P solutions create an overlay network between the users' machines and on top of the physical layer of the Internet. In a peer-to-peer system, every peer executes the same small set of instructions and together all nodes give raise to an emergent behavior with specific properties. The main argument in favor of P2P live streaming is that of scalability, the more users that are running the P2P application the larger the number of viewers the service is able to sustain. In the P2P model, peers which are part of the system retrieve the video content and then forward it to other peers. Since each node contributes its own resources to the system, namely upload bandwidth capacity and processing power, the capacity of the system usually grows when the number of users increases.

1.1 Motivation

Peer-to-Peer overlays have the potential of providing streaming at large scale and with a fraction of the cost of CDNs by using resources that are already available at the edge of the network, i.e., the user machines, rather than deploying new CDN nodes across the Internet. However, the adoption of peer-to-peer technologies by the industry has been very limited. We argue that this is due to two fundamental reasons.

First, classical peer-to-peer live streaming approaches focus on solving the problem of distributing video to a large number of users with the assumption that the source of the stream has fixed and limited capacity, e.g. a user machine, trying to provide best effort quality of user experience (QoE) to users. This is in contradiction with industrial expectations. Commercial content providers are not willing to compromise on QoE, instead they are willing to sustain the cost of increased capacity at the source of the stream or higher CDN usage to keep the desired level of QoE. As a consequence, we argue that the problem that a peer-to-peer live streaming platform should solve is that of providing *first* a well defined level of quality of user experience, such as delay from the live point, throughput and continuity of playback, and *then* try to minimize the load on the provider's infrastructure.

Second, current state-of-the-art P2P live streaming (PLS) literature is based on the assumption that video streams are distributed utilizing the RTSP/RTP protocol, whereas the industry has shifted almost entirely to the pull-based dictated by HTTP streaming protocols. On top of that, adaptive streaming has been introduced as an integral part of the HTTP streaming protocols as means to cope with heterogeneous bandwidth availability and multiple devices with different video rendering capabilities. This shift from the push-based RTSP/RTP protocol to the pull-based HTTP-live protocols and the introduction in mainstream protocols of adaptive bitrate have rendered many of the classical assumptions made in the PLS literature obsolete. For the sake of peer-to-peer technology adoption, it is therefore necessary to develop solutions which support adaptive HTTP streaming.

A plethora of solutions has been produced for the problem of scalability of live streaming using peer-to-peer in the last years. However, very few of the proposed ideas have been implemented in a real environment, even fewer have known adoption by users. Given the commercial context in which this thesis was conducted, our goal is to design an approach which is feasible and then produce a deployed system which can be used by real viewers. As a consequence of that, we concentrate on testing and evaluation of our ideas in realistic test-beds and live broadcast events, rather than in simulation or using mathematical models.

1.2 Contribution

In this thesis, we solve the problem of providing live video streaming to large number of viewers using peer-to-peer overlays. We do so by assuming that our solution must deliver a level of quality of user experience which is the same of a CDN while trying to minimize the load on the content provider's infrastructure, namely the source of the stream. In addition, our design takes into consideration the realities of the HTTP streaming protocols, such as the pull-based approach and adaptive bitrate switching.

The result of this work is a system which we call SmoothCache and which provides CDN-quality adaptive HTTP live streaming utilizing peer-to-peer algorithms. SmoothCache is a peer-assisted solution, it strives to retrieve most of the content from the overlay but it resorts to existing CDN infrastructure to compensate for deficiencies of the peer-to-peer delivery. We quantify the system's quality of user experience and compare it to the one of a CDN-only delivery service using the following metrics: i) cumulative playback delay, the time it for the video to start after the user requests it plus the cumulative time the playback was interrupted because of lack of data, ii) delivered bitrate, how many peers could watch a certain bitrate, iii) delay from the playing point, the period of time between when the content is made available to peers and when it is consumed by the player.

Considering the aforementioned metrics, in this thesis we show that our system provides performance on the Internet that is equivalent to that of a CDN-only service. To be able to make that claim, we conduct a thorough evaluation of the platform on a subset of around 22.000 from the around 400.000 installations of our system worldwide. This subset is made by customers who agreed to let us conduct experiments on their machines. Our experiments highlight that the system, besides meeting the the CDN-quality constraints, is able to consistently deliver up to 96% savings towards the source of the stream in a single bitrate scenario and 94% in a scenario where the stream is made available at different bitrates. In addition to controlled tests on the Internet, we have conducted a number of pilot deployments in the setting of large enterprise networks with the same system, albeit tailored to enterprise networks rather than the Internet. An enterprise network is a large private network which interconnects multiple geographically distributed branches of a company. Links between branches and towards the rest of the Internet typically constitute bottlenecks during the broadcast of live events. Results with our SmoothCache system collected during pilot events with thousands of real

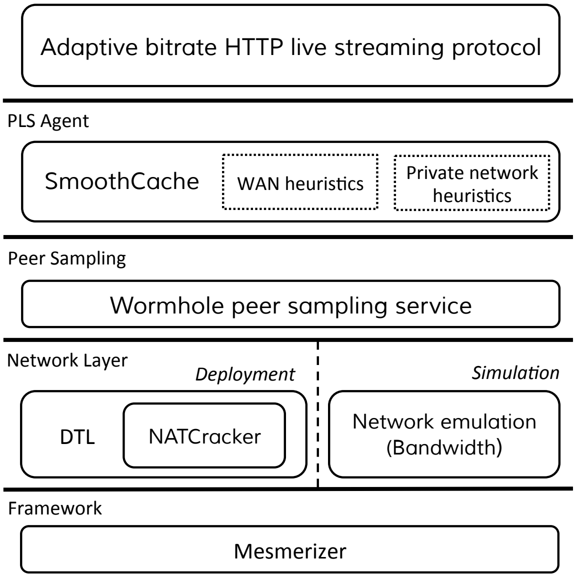

Adaptive bitrate HTTP live streaming protocol
PLS Agent
SmoothCache
WAN heuristics
Private network heuristics
Peer Sampling
Wormhole peer sampling service
Network Layer
Deployment
Simulation
DTL
NATCracker
Network emulation (Bandwidth)
Framework
Mesmerizer

Figure 1.1: Layered view of the SmootchCache system

viewers show that the platform provides more than 90% of savings towards the source of the stream and delivers the expected quality of user experience.

These achievements are the result of advancements made in multiple research areas and these advancements are also presented in this thesis. Each one of those contributions is novel with respect to the state of the art and can be applied outside of the context of our application. However, in our system they are organized as shown in Figure 1.1 and serve the purposes described below.

First, we present a component-based event-driven framework, called *Mesmerizer*, to facilitate the development of our live streaming application. The framework enables the execution of the same code both in simulation and in a real deployment. In simulation, we execute experiments on an emulated network where we model multiple characteristics of physical networks, such as the presence of Network Address Translators, network delay patterns and *bandwidth allocation dynamics*. In order to improve scalability and accuracy of simulations, we designed a novel flow-based bandwidth emulation model based on a variation of the min-max fairness algorithm.

In order to deploy our application on real networks, we developed a novel network library called *DTL* which provides reliability and on-the-fly prioritization of transfers. The library is based on the UDP protocol and supports NAT Traversal techniques. NAT traversal facilitates connectivity between hosts by working around NAT limitations and therefore letting peers establish direct connectivity, that is without relaying to a third party. As part of this thesis, we developed a scheme (*NATCracker*) which improves on the state of the art of NAT Traversal techniques resulting in higher probability of direct connectivity between peers on the Internet.

Because of the presence of NATs in the network, discovery of new peers and collection of statistics on the overlay through peer sampling is problematic. Therefore we created a peer sampling service, the *Wormhole peer sampling service*, which is NAT-resilient and provides with the same overhead as similar protocols based on gossip one order of magnitude fresher samples.

Information provided by the peer sampling service is then used to build our delivery overlay. In order to provide the same QoE as CDNs, while achieving savings towards the source of the stream, we implemented a system called *SmoothCache* which provides a distributed caching abstraction on top of the peer-to-peer overlay. The distributed cache makes sure that fragments are always delivered on-time to the player by either retrieving the data from the overlay if the data is present, or, if the content can't be retrieved quickly enough from the P2P network, SmoothCache retrieves it directly from the source of the stream to maintain the expected QoE. That said, in order to provide savings, we make sure to fill up the local cache of the peers ahead of playback. We do this by implementing prefetching heuristics and by constructing a self-organizing overlay network that takes into account many factors such as upload bandwidth capacity, connectivity constraints, performance history, prefetching point and the currently watched bitrate, all of which work together to maximize flow of fragments in the network.

Here we summarize each contribution of this thesis separately and state its novelty aspects:

- **Development and emulation framework**. We define a set of the best practices in Peer-To-Peer(P2P) application development and combine them in a middleware platform called Mesmerizer. That is a component-based event-driven framework for P2P application development, which can be used to execute multiple instances of the same application in a strictly controlled manner over an emulated network layer for simulation/testing, or a single application in a concurrent environment for deployment purpose. We highlight modeling aspects that are of critical importance for designing and testing P2P applications, such as the emulation of Network Address Translation behavior and bandwidth dynamics. We present this work in Chapter 3.

- **Simulation of bandwidth dynamics**. When evaluating Peer-to-Peer live streaming systems by means of simulation, it is of vital importance to correctly model how the underlying network manages the bandwidth capacity of peers when exchanging large amount of data. For this reason, we propose a scalable and accurate flow-level network simulation model based on an evolution of the classical progressive filling algorithm which employs the max-min fairness algorithm. Our experiments show that, in terms of scalability, our bandwidth allocation algorithm outperforms existing models when simulating large-scale structured overlay networks. Whereas, in terms of accuracy, we show that allocation dynamics of the proposed solution follow those of packet-level simulators. We describe our model in chapter 4.

- **Transport**. We developed a reliable transport library for peer-to-peer applications which provides variable and on-the-fly transfer prioritization. For portability, the library is developed in Java on top of UDP. It implements intra- and inter-protocol prioritization by combining two state of the art congestion control algorithms: LEDBAT and MulTCP. The library supports a range of priority levels, from less-than-best-effort priority up to high. The prioritization level can be configured at runtime by the over-lying application using a single input parameter. Support for on-the-fly priority variation enables applications to tune transfers in order to achieve the desired level of QoS. Transitions between different prioritization levels happen without disruptions in the flow of data and without the need for connection re-establishments, which usually involve time consuming NAT Traversal and peering authentication procedures. We describe the details of our network library in Chapter 6.

- **NAT Traversal**. We improve on the state of the art on Network Address Translation (NAT) traversal by providing a deeper dissection of NAT behaviors resulting in 27 different NAT types. Given the more elaborate set of behaviors, it is incorrect to reason about traversing a single NAT, instead combinations must be considered. Therefore, we provide a comprehensive study which states, for every possible combination, whether direct connectivity with no relay is feasible. Our novel NAT Traversal framework is presented in Chapter 5.

- **Peer Sampling**. We present a novel peer sampling service called wormhole-based peer sampling service (WPSS) which executes short random walks over a stable topology. WPSS improves on the state of the art by decreasing the number of connections established per time unit by one order of magnitude while providing the same level of freshness for samples. We achieve this without sacrificing the desirable properties of a PSS for the Internet, such as robustness to churn and NAT-friendliness. Our work on peer sampling can be found in Chapter 7.

- **Live Streaming**. We designed and developed a distributed cache for adaptive HTTP live streaming content based on peer-to-peer (P2P) overlays. The contribution of this work is twofold. From a systems perspective, it supports live streaming protocols based on HTTP as a transport and the concept of adaptive bitrate switching. From an algorithmic perspective, the system describes a novel set of overlay construction and prefetching techniques that realize: i) substantial savings in terms of the bandwidth load on the source of the stream, and ii) CDN-quality user experience. We describe the first iteration of our live streaming platform, called SmoothCache 1.0, in Chapter 8, where we show that savings towards the source of the stream can be obtained, in the context of adaptive HTTP live streaming, using peer-to-peer overlays. The next iteration of our system, which we call SmoothCache 2.0, is described in Chapter 9. There we present the novel set of heuristics which allows to achieve CDN-quality adaptive HTTP live streaming and a thorough evaluation on thousands of consumer machines. Chapter 10 presents the application of our system to the scenario of large private networks. In that chapter, we outline the changes we implemented in SmoothCache to adapt our algorithms to the private network setting and show initial results with live broadcasts with thousands of real viewers.

1.3 Publications

We here provide a list of publications which have been published as part of this thesis:

- Roberto Roverso, Sameh El-Ansary, Alexandros Gkogkas, and Seif Haridi. *Mesmerizer: a effective tool for a complete peer-to-peer software development life-cycle.* In Proceedings of the 4th International ACM/ICST Conference on Simulation Tools and Techniques [6]. **SIMUTools 2011**, Brussels, Belgium, 2011

- Alexandros Gkogkas, Roberto Roverso, and Seif Haridi. *Accurate and efficient simulation of bandwidth dynamics for peer-to-peer overlay networks.* In Proceedings of the 5th International ICST Conference on Performance Evaluation Methodologies and Tools [7]. **VALUETOOLS 2011**, Brussels, Belgium, 2011

- Roberto Roverso, Sameh El-Ansary, and Seif Haridi. *NATCracker: NAT Combinations Matter.* In Proceedings of 18th International Conference on Computer Communications and Networks 2009 [8]. **ICCCN 2009**, San Francisco, CA, USA, 2009

- Riccardo Reale, Roberto Roverso, Sameh El-Ansary, and Seif Haridi. *DTL: dynamic transport library for peer-to-peer applications.* In Proceedings of the 12th International Conference on Distributed Computing and Networking 2012 [9]. **ICDCN 2012**, Hong Kong, China, 2012

- Roberto Roverso, Jim Dowling, and Mark Jelasity. *Through the wormhole: Low cost, fresh peer sampling for the internet.* In Peer-to-Peer Computing (P2P), 2013 IEEE 13th International Conference on [10]. **P2P 2013**, Trento, Italy, 2013

- Roberto Roverso, Sameh El-Ansary, and Seif Haridi. *Smoothcache: HTTP-live streaming goes peer-to-peer.* In IFIP International Conferences on Networking 2012, volume 7290 of Lecture Notes in Computer Science [11]. **NETWORKING 2013**, Prague, Czech Republic, 2013

- Roberto Roverso, Sameh El-Ansary, and Seif Haridi. *Peer2view: A peer-to-peer HTTP-live streaming platform.* In Peer-to-Peer Computing (P2P), 2012 IEEE 12th International Conference on [12]. **P2P 2012**, Tarragona, Spain, 2012

- Roberto Roverso, Sameh El-Ansary, and Seif Haridi. *SmoothCache 2.0: the HTTP-Live peer-to-peer CDN.* In **Peerialism White Papers**, Stockholm, Sweden 2013

- Roberto Roverso, Sameh El-Ansary, and Mikael Hoeqvist. *On HTTP live streaming in large enterprises.* In Proceedings of the ACM conference on applications, technologies, architectures, and protocols for computer communication [13]. **SIGCOMM 2013**, Hong Kong, China, 2013

List of publications of the same author but not related to this work

- Roberto Roverso, Amgad Naiem, Mohammed Reda, Mohammed El-Beltagy, Sameh El-Ansary, Nils Franzen, and Seif Haridi. *On The Feasibility Of Centrally-Coordinated Peer-To-Peer Live Streaming.* In Proceedings of IEEE Consumer Communications and Networking Conference 2011 [14]. **CCNC 2011**, Las Vegas, NV, USA, 2011
- Roberto Roverso, Amgad Naiem, Mohammed El-Beltagy, Sameh El-Ansary, and Seif Haridi. A GPU-enabled solver for time-constrained linear sum assignment problems. In Informatics and Systems, 2010 The 7th International Conference on [15]. **INFOS 2011**, Cairo, Egypt, 2010.
- Roberto Roverso, Mohammed Al-Aggan, Amgad Naiem, Andreas Dahlstrom, Sameh El-Ansary, Mohammed El-Beltagy, and Seif Haridi. *MyP2PWorld: Highly Reproducible Application-Level Emulation of P2P Systems.* In Proceedings of the 2008 Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops [16]. **SASO Workshops 2008**, Venice, Italy, 2008.
- Roberto Roverso, Cosmin Arad, Ali Ghodsi, and Seif Haridi. *DKS: Distributed K-Ary System a Middleware for Building Large Scale Dynamic Distributed Applications*, Book Chapter [17]. In **Making Grids Work 2007**, Springer US, 2007.

CHAPTER 2

BACKGROUND

In this chapter, we present the main concepts that are necessary to understand the research explained in the rest of the thesis.

We start by describing standard practices for the distribution of live content over the Internet to large audiences in Section 2.1. We then define the challenges of using peer-to-peer networks for Internet live streaming in Section 2.1. After that, we provide an extensive description of state-of-the-art techniques to overcome those challenges. The techniques are classified in two main areas: construction/management of overlays, presented in Section 2.2.1, and data dissemination, discussed in Section 2.2.2. In Section 2.2.3, we highlight methods to leverage infrastructure resources to achieve a quality of user experience in peer-to-peer live streaming that compares to that of client-server solutions.

Besides live streaming, in this chapter we also discuss other topics that are relevant to the design and implementation of peer-to-peer content delivery systems. In Section 2.3, we delve into details of peer connectivity in the presence of Network Address Translators (NATs). In Section 2.4, we introduce peer sampling algorithms that are used to provide inpuy for peer-to-peer overlay construction in P2P live streaming and that are also resilient to NATs. In Section 2.5, we describe transport protocols for peer-to-peer content distribution applications which are employed to transfer data between peers once the overlay network is in place. We conclude the background chapter with Section 2.6 that summarizes a set of best practices that we found useful for development and evaluation of peer-to-peer applications.

2.1 Live Streaming

In live streaming, content is created and broadcasted in real-time. The strongest constraint on the delivery of the stream is that the stream must be received within a small *delay* from the time it is generated by the content provider. Furthermore, users in the same geographical region should not experience significant differences in their playback point. These requirements tie directly into the meaning of live broadcasting.

Live streaming is usually implemented using stateful push-based protocols, such as RTP/RTSP[18] and RDT[19], where a player software installed in the viewer's machine establishes and manages media sessions by issuing commands to the streaming server, e.g. *play*, *stop* and *pause*. The data is delivered to the player by having the server push the content fragments at at the rate it deems suitable to the player. At the transport level, data is delivered over UDP, while TCP is used for control messages.

Recently, the industry has introduced a new technology for live streaming, called *Adaptive HTTP streaming*. Adaptive HTTP streaming consists of a set of protocols which all utilize HTTP as transport [2][3][4][5]. HTTP-live streaming protocols are based on a pull-based model, where it is the player which requests content over HTTP at the pace it deems suitable. On top of that, HTTP-live protocols have been designed to support an adaptive bitrate mode of operation, which provides the stream at a number of bitrates. This shift to HTTP has been driven by a number of advantages such as the following: i) Routers and firewalls are more permissive to HTTP traffic compared to the RTSP/RTP ii) HTTP caching for real-time generated media is straight-forward like any traditional web-content iii) the Content Distribution Networks (CDNs) business is much cheaper when dealing with HTTP downloads [4].

In HTTP streaming, the stream is split into a number of small HTTP packets, i.e. fragments, and the streaming server appears to the player as a standard HTTP server. When a player first contacts the streaming server, it is presented with a descriptor file, called *Manifest*, which outlines the characteristics of the stream. It contains the stream's fragments path on the HTTP server and the quality of bitrates available for the stream. After reading the manifest, the player starts to request fragments from the server. The burden of keeping the timeliness of the live stream is totally upon the player, while the server is stateless and merely serves fragments like any other HTTP server after encoding them in the format advertised in the manifest. That means that the player implements a number of heuristics that determine at which pace fragments should be downloaded and at which bitrate [20][21][22]. In general, the choice of bitrate to request is dictated by the available bandwidth. That is, when the user has more available bandwidth, the player requests a higher quality stream and when the host has lower bandwidth available it requests a lower quality stream. Other factors that can influence the choice of bandwidth are, for example, the computational power of the machine the player is running on or how fast the machine's graphic card can render video content. Different HTTP streaming player implementations use different heuristics and behave in different ways. Thus, it is not possible to predict a player's behavior. In fact, it is common to observe changes in behaviors even in different versions of the same player implementation.

2.2 Peer-to-peer Live Streaming

Live streaming on overlays is inherently difficult due to the fact that participating nodes are not reliable and they might become unavailable at any time because of temporary congestion, connectivity issues or simply because of users leaving the system. The process of continuous joining and leaving of nodes is called churn. Much attention in the design of P2P algorithms for live streaming is therefore dedicated to trying to maintain a certain level of performance in the delivery of the stream while coping with churn. Data collected from the deployment of the peer-to-peer live streaming platform PPLive [23] show that a steady amount of churn is expected when broadcasting linear TV channels. On top of that, flashcrowds and massive failure scenarios are also observed, that is the sudden join and the sudden departure of a large number of nodes, respectively.

Another source of complexity in the design of peer-to-peer systems is network-level *congestion*. Congestion stems from the overloading of physical network's resources, caused by excessive user-generated traffic in the network, which causes the formation of bottlenecks along routing paths between peers on the overlay. The effects of network congestion on P2P live streaming are: diminished throughout, longer transmission delay, packet loss and blocking of new connections. Although it is believed that most of the time bottlenecks form at the last mile segment in the Internet infrastructure, i.e. at residential gateways, congestion can be experienced at different levels in the network. One reason for that is that ISPs and ASs dimension their internal network and border links considering the average usage scenario, rather than the peak. That means that congestion happens every time more users than expected access their ISP network. This can happen, for example, during the streaming of live events of high interest.

In classical peer-to-peer live streaming approaches, the delay from the time the content is generated by the provider to the time it is delivered to the player may vary for each peer. This is because the data has to traverse multiple hops before becoming available for playback at a peer. The number of hops depends on how the overlay is constructed and how quickly the data is transferred at each hop. Depending on the capacity at the source of the stream, the delay might be very long, on the order of many seconds or even minutes. In this thesis however, we work under the assumption that there must be a target delay value and the system must strive to keep this value constant and the same for all peers. Given that peer-to-peer delivery is strongly influenced by churn and congestion, the target delay usually cannot be guaranteed with peer-to-peer delivery alone. For that reason, a new breed of hybrid systems which uses infrastructure resources to compensate for the limitations of P2P delivery was recently introduced. In the following sections, we are going to describe general peer-to-peer overlay techniques which apply both to classical P2P live streaming and hybrid infrastructure/P2P systems. We will then highlight the infrastructural characteristics of the latter in Section 2.2.3.

Peer-to-peer live approaches differ mainly on two levels: *overlay construction* and *data dissemination*. Overlay construction is the set of heuristics a live streaming system uses to create the set of connections between peers which then would be used to transport control messages and the stream data. The data is usually logically split into fixed-size chunks which are transferred between peers according to a certain data dis-

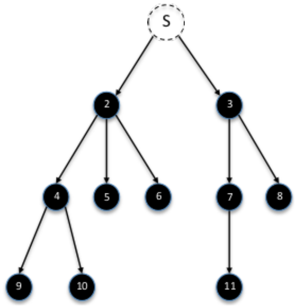


Figure 2.1: Representation of a tree-based overlay structure

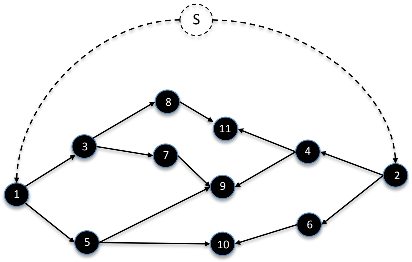


Figure 2.2: Representation of a multi-tree-based overlay structure

semination strategy.

2.2.1 Overlay Construction

There exists three main classes of overlay construction techniques that are commonly used for live streaming: *tree-based, mesh-based* and *hybrid*.

Tree-based

Tree-based overlay construction techniques aim to have peers self-organize in a tree-like structure, like the one presented in Figure 2.1, where the source of the content is positioned at the root of the tree. The content is disseminated from the root of the tree to the other peers by letting peers receive the video from their parents and then forward it to their children. Famous examples of tree-based systems are Overcast [24], Climber [25] and ZigZag [26]. The main advantage of a tree-based overlay is that the distribution delay for each peer is proportional to the number of hops from the source and the delay corresponds to the sum of the delays traversing the tree.

In tree-based systems, a peer has one or more parents which provide it with chunks of data and a set of children which it is forwarding the received chunks to. The number of children which a parent can provide for is proportional to the bandwidth upload capacity of a peer. That is typically a parent has a number of children which equals to its upload capacity divided by the bitrate that peers are watching. It is widely accepted [27][14][28] that the most efficient strategy to place peers in a overlay tree is by sorting them according to the number of children they can afford, that is their upload bandwidth. Peers with highest upload bandwidth capacity are usually placed near the the source of the stream and the others in subsequent rows depending on their upload capacity, in decreasing order. This makes sure that the number of rows in the tree is kept to a minimum and therefore also the amount of time after which the peers receive the content is also minimized.

Tree-based systems may rely on a single tree or on multiple tree structure. In a single tree structure, the leaves of the tree do not contribute to the delivery of the stream, since they have no children. In a multi-tree configuration instead, the content server splits the stream in multiple sub-streams which are then disseminated over separate tree-shaped overlays. As a consequence, a peer might be parent for a sub-stream but only a child for another. Solutions using this approach are, for instance, SplitStream [29] and Orchard [30]. A study on a widely deployed system, i.e. GridMedia [31], has shown that using multi-tree based systems leads to better performance than single-tree approaches and to near-optimal bandwidth utilization. Figure 2.2 shows a multi-tree overlay network structure. In the example, two sub-streams are broadcasted from the source of the stream along two trees, starting at peer 2 and 1.

The main disadvantage of tree-based approaches is that they are not strongly resilient to churn. In fact, when a peer abruptly leaves the overlay, all of its descendants get disconnected also and need to find another ancestor. On top of that, parts of the overlay can become partitioned from the rest of the overlay. Even if solutions have been suggested to alleviate these issues, such as the use of structured networks for peer placement [29], it is widely accepted that tree-based overlays are inferior to mesh-based approaches in presence of churn [32].

The two broad classes of algorithms used to construct tree-based overlays are *centralized* and *decentralized* algorithms.

Centralized Construction. In centralized tree construction approaches, a central coordinator instruct peers about which parent they should receive the stream from. The peers periodically report performance metrics to the coordinator. The central coordinator then keeps an overview of the system which includes the configuration of the tree at a certain point in time. Using this information, the central server makes decisions about where to place new peers in the overlay topology.

The principal drawback of this approach is that the coordinator constitutes a single point of failure in the system. If the coordinator fails, new peers cannot join the overlay. Also, peers that are already in the system cannot replace failed parents. In addition to that, the central coordinator is the main performance bottleneck in centralized systems. It must cope with both significant number of peers joining in a short period of time, i.e. flash crowds, and large amounts of peers leaving the system abruptly, i.e. massive failures. Both phenomena are very common in live streaming [23]. During flash crowds and massive failures periods, the coordinator needs to handle high network traffic coming from peers but also it must re-evaluate the view of the overlay very frequently in order to be able to instruct new and old peers on which parents to use.

Examples of systems using centralized coordination are Peer2View [33] and Coopnet [34]. Central coordination has also been used for classical content distribution in Antfarm [35].

Distributed Construction. A number of distributed algorithms have been designed to construct and maintain a tree-based live streaming overlay. In this approach, peers negotiate by means of gossiping their placement in the tree mainly using their upload bandwidth as a metric. Examples of systems using decentralized tree construction are: SplitStream [29], Orchard [30] and ChunkySpread [36].

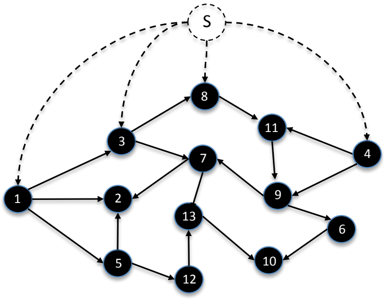


Figure 2.3: Representation of a mesh-based overlay structure

Mesh-based

In mesh-based overlay networks no overlay structure is enforced. Instead, peers create and lose peering relationships dynamically. The overlay is constructed in a way that only few peers are directly connected to the source of the stream while the majority of peers exchanges content through overlay links. An example of a mesh-based overlay is shown in Figure 2.3.

Examples of mesh-based approaches are: SopCast [37], DONet/Coolstreaming [38], Chainsaw [39], BiToS [40] and PULSE [41]. A mesh-based system is usually composed by two main parts: membership and partnership. The *membership* mechanism allows a peer to discover other peers that are watching the same stream and collect information about them, such as type of their connectivity and performance metrics. This is usually achieved by means of a central discovery service to which all peers periodically report, e.g. a tracker. Another way of discovering peers is to use distributed peer sampling. Distributed peer sampling algorithms enable peers to collect information about peers in the overlay without a tracker. The most common way of implementing distributed peer sampling is by means of gossip. In gossip, every peer maintains a *view* of the overlay which is kept updated by continuous exchanges with other peers.

It is quite common in P2P live streaming to use multiple instances of gossip. The first instance of gossip implements unbiased peer sampling to collect uniform samples of the overlay, while other instances use the information provided by the peer sampling to find neighbors that have interesting characteristics, e.g. peers with similar playback point and available upload capacity [28] or that are geographically closer to the requester [42].

Once enough information has been collected by the membership, the *partnership* service establishes temporary peering connections with a subset of the peers a node is aware of in order to transfer the stream of data. Given that peers connect to multiple partners at random, the peering degree and randomness make mesh-based systems extremely robust to both churn and network-related disruptions, such as congestion. Since no fixed stream delivery structure is enforced on the overlay, a peer can quickly switch between different provider peers if a failure occurs or of the the necessary streaming rate cannot be sustained.

A partnership between two peers is commonly established according to the following metrics:

- The *load* on the peer and resource availability at both ends. Possible load metrics include: available upload and download bandwidth capacity and CPU/memory usage.
- *Network Connectivity.* The potential quality of the link between the two peers in terms of delay, network proximity and firewall/NAT configurations, as in [8].
- *Content Availability.* The available data chunks, at both ends, i.e. the parts of the stream which have been already downloaded and are available locally at a peer.

As for drawbacks, control traffic generated by the mesh-based partnership service is usually significant given that peers need to frequently exchange status information for deciding which are the best peers to download chunks from. On top of that, in mesh-based systems, a larger number of connections are maintained than for tree-based systems. A high number of partners gives more flexibility when requesting content so that peers may have more choice to quickly change from one partner to another if complications arise during the delivery.

Another drawback of the mesh-based approach is sub-optimal bandwidth utilization of provider peers since every data chunk is treated as a separate delivery unit, while per-chunk distribution paths and delivery times are not predictable and highly variable.

Hybrid Overlay Construction Approaches

A tree-based approach can be combined with a mesh-based one to obtain better bandwidth utilization. mTreebone[43] elects a subset of nodes in the system as stable and uses them to form a tree structure. The content is broadcasted from the source node along the tree structure. A second mesh overlay is then built comprising both the peers in the tree and the rest of the peers in the system. For content delivery, the peers rely on the few elected stable nodes but default to the auxiliary mesh nodes in case of a stable node failure. The drawback of this approach is that a few stable peers might become congested while the others are not contributing with their upload bandwidth. Thus, this solution clearly ignores the aspect of efficient bandwidth utilization.

CliqueStream [44] presents an alternative approach where clusters of peers are created using both delay and locality metrics. One or more peers are then elected in each cluster to form a tree-like structure to interconnect the clusters. Both in PRIME [45] and NewCoolsteaming [46], peers establish a semi-stable parent-to-child relationship to combine the best of push and pull mesh approach. Typically a child subscribes to a certain stream of chunks from the parent and the parent pushes data to the child as soon as it becomes available. It has been shown that this hybrid approach can achieve near-optimal streaming rates [47] but, in this case as well, no consideration has been given to efficient bandwidth utilization.

2.2.2 Data Dissemination

In classical peer-to-peer live streaming systems, peers typically have full control of the player software. The player is very resilient and can afford temporary starvation, in which case it just stops rendering the video, and loss of consecutive video fragments, then it merely skips the missing fragments. To alleviate the effect of temporary disruptions on data delivery, caused by churn, connectivity issues and congestion, p2p systems introduce a buffer between the overlay and the player. The presence of the buffer increases the probability of downloading video fragments in time before they are due for playback. Since the buffer introduces a delay from the point the data is received by the peer to the time it is provided to the player, it is usually desirable to keep the size of the buffer small.

Once an overlay is in place, peers can start exchanging data to fill the buffer using multiple strategies. The most common are *push*, *pull* and *push-pull*. Push-based dissemination is typical of tree-based systems. Parent peers receive data chunks and forward them to the children as fast as possible. This guarantees high throughput between peers and in general maximizes bandwidth utilization on the overlay. Unfortunately, due to churn, peers may be forced to switch frequently between parents and, therefore, waste upload capacity of parents in the process.

The pull-based technique is instead typical of mesh-based systems. In pull-based systems, partners on the overlay exchange bitmaps which indicate which video chunks they have downloaded up to the moment of the exchange. The pull method is very resilient to churn because content chunks are only requested from partners which have recently advertised said chunks and therefore have a high probability of being alive. On the other hand, there is no guarantee that a peer will receive enough requests for fragments from its partners and thus utilize its full upload capacity.

In pull-based methods, a *scheduler* on each peer is given the task of requesting content chunks as the playback progresses based on availability at its partners. Since each chunk may be requested individually and from different partners, chunks may be delivered over different paths to a peer and arrive out of order. Consequently, the peer re-orders chunks before delivering them to the player. In addition, it may happen that a data chunk can not be downloaded by a node because it is not available at its partners (content bottleneck) or the upload bandwidth of it partners is not insufficient to transfer the chunk in time (bandwidth bottleneck) [48]. For decreasing the probability of failing to download a chunk in time, it is common to employ different retrieval strategies according to the timeline of chunks. GradienTv [28], for instance, normally requests chunks from peers that have a similar upload bandwidth capacity. However, for chunks that are due for playback soon, and have not been dowloaded yet, it issues requests to peers with higher upload capacity. This increases the probability of downloading missing fragments on time. In Bittorrent-based live streaming platforms, such as Pilecast [49], peers use a combination of the *rarest-first* and *greedy* policies [50] to decide which fragments to pull from neighbors. Peers employ the greedy policy to download chunks which are due for playback soon, that is in the next 15 seconds of playback. When all of those fragments have been retrieved, peers switch to the rarest-first policy in order to download future fragments, that is more than 15 seconds ahead of the playback point. Finally, some systems leverage structured overlay networks, such as Distributed Hash Tables, to improve availability of chunks. In HyDeA [51], a custom DHT is used to find and retrieve chunks which, again, were not retrieved quickly enough from the unstructured mesh-based network.

There exists also systems which implement a combination of the push and pull methods to overcome the limitations of both pull, that is content and bandwidth bottlenecks, and push, that is limited resiliency to churn. In push-pull based systems, when a new peer joins the system, it pulls data from nodes that are already in the overlay. After a period of observation and learning about possible candidates, a peer chooses one or multiple parents from its partners which then start to push data to it. If some of the chunks cannot be downloaded in time, a peer resorts again to pull from their partners. Examples of systems implementing push-pull are Gridmedia[52], NewCoolStreaming [46], CliqueStream [44], and mTreebone [43].

2.2.3 Infrastructure-aided P2P Live Streaming

We have, up until now, mentioned systems that rely exclusively on peer-to-peer resources to distribute live content over the Internet. Here instead, we survey a class of hybrid live streaming systems which use both infrastructure services, such as cloud servers or CDNs, and peer-contributed resources to meet a quality of user experience for viewers which compares to that of client-server solutions. In this type of systems, peers leverage the peer-to-peer network to retrieve the majority of content chunks, but they resort to central infrastructure to compensate for deficiencies of the peer-to-peer delivery, such as node-to-node congestion and churn.

We classify infrastructure-aided P2P live streaming systems in two categories: *cloud-assisted* and *peer-assisted*. The former category includes systems which build their own ad-hoc infrastructure to aid the peer-to-peer overlay, while systems in the latter category use existing infrastructure such as Content Delivery Networks for the same purpose. While the approaches may differ, both categories of systems share the same goal of enforcing a specific target quality of user experience for viewers, in terms of delay and delivered bitrate, by using the least amount of infrastructure resources possible.

Cloud-assisted

The main idea behind cloud-assisted approaches is to leverage cloud resources, that is dynamic allocation of helper servers, to build an ad-hoc infrastructure to support the peer-to-peer overlay and achieve the target QoE. In cloud-assisted systems, a coordinator dynamically allocates cloud server instances with a specific amount of resources, that is computation and bandwidth capacity, to serve the peer-to-peer network according to the current state of the overlay. The coordinator continuously receives feedback from all peers and periodically estimates the number of cloud servers needed to enforce the target QoE. This target is typically expressed as a bound on the playback delay from the live point for all peers in the system. The metrics used as input for the coordinator's decision are: the bitrate of the video being watched, the total number of peers, the bandwidth capacity of helper servers and the characteristics of peers, such as upload bandwidth capacity and churn rate. Since cloud servers are allocated and removed dynamically, cloud-assisted solutions can limit the amount of infrastructure resources, and therefore cost, to a minimum. On the other hand, cloud-assisted solutions are cumbersome to deploy in existing commercial live streaming systems that rely on CDN services. They in fact require a complete re-engineering of the server-side of the system to allow feedback between the peer-to-peer overlay and the coordinator. Besides that, a completely new management layer for cloud resources must be developed and deployed for the coordinator to enforce its decisions.

The most popular example of cloud-assisted live streaming system is LiveSky [53]. CALMS [54], Clive [55] and Angelcast [56] are other instances of the same approach. In contrast to standard cloud-assisted solutions, PACDN [57] proposes an approach where the infrastructure is made of pre-existing edge servers rather than dynamically allocated cloud nodes. In PACDN, edge servers form their own centrally-coordinated tree-based overlay network to make sure content is distributed efficiently from the streaming server to all edge nodes. Similarly to cloud-assisted approaches, edge nodes are activated or de-activated dynamically by a coordinator according to the state of the user-contributed peer-to-peer network.

Peer-Assisted

In contrast to cloud-assisted approaches, peer-assisted content delivery has the goal of making use of existing Content Delivery Networks infrastructure to help the peer-to-peer delivery. This, without the need for integration.

From an operational point of view, it means that peers in the system strive to download most of the content from the user-contributed peer-to-peer network but they resort to the CDN in case video chunks cannot be retrieved in time from the overlay. The underlying assumption in this case is that the CDN has enough capacity to complement the peer-to-peer delivery at all times.

CDNs are inherently elastic, they implement mechanisms to automatically scale up or down resources according to the experienced load. This is motivated by the fact that it is very common for a CDN to experience dramatically varying demand for video content on the Internet depending on factors such as increased/decreased popularity and time of the day. Peer-assisted hybrid approaches rely on CDN elasticity to make sure that there is always enough bandwidth at the server side for peers to use in order to compensate for the deficiencies of the peer-to-peer overlay. The clear advantage in this case is that peer-assisted systems do not need coordination with the central infrastructure but rather extend the capacity of CDNs without the need of integration against the server side of the system. This makes deployment much easier than cloud-assisted approaches.

Because of its inherent benefits, the peer-assisted approach has recently gained momentum in the video distribution industry with companies developing their own peer-assisted content delivery software, such as Akamai's NetSession [58], or integrating peer-to-peer in their existing distribution frameworks, as in the case of the Adobe's Flash framework [59]. Both NetSession and the Flash framework also support adaptive bitrate HTTP streaming but they are mostly used for Video-on-Demand (VoD) [60][58] rather than live streaming. Note that, in industrial context, the concept of peer-assisted streaming has a slightly different meaning from the one presented in academic literature. In industrial peer-assisted systems, the CDN can afford the load of serving all users in the system by scaling up and down resources as needed. The peer-to-peer delivery is then leveraged as a method to drive down the amount of bandwidth consumed by VoD viewers but it is never expected to carry most of the delivery, as it does in all academic work.

In academia, the use of peer-assisted delivery has also been limited to video-on-demand streaming. From a theoretical point of view, efforts have been directed at understanding in general, and without considering the details of streaming protocols, what are the best prefetch policies for peers to download content ahead of the playback deadline such that the load on the source of the stream is minimized [61][62]. Recent work, by Maygh [63], instead proposed a practical solution to enable peer-assisted distribution of adaptive bitrate HTTP streaming of VoD content using plugin-less browser-based technologies. Authors show that Maygh can save up to 75% of the load on the CDN in the considered scenarios.

We have mentioned so far approaches that resort to peer-assisted delivery for video on demand streaming, and, in most cases, also provide support for adaptive HTTP streaming. In this thesis instead, we apply the peer-assisted approach to live streaming. The result of our efforts is the SmoothCache system, which is the first peer-assisted solution for the distribution of adaptive bitrate HTTP live streaming over the Internet [11][64] and in large private networks [13].

The two different iterations of the SmoothCache system design are presented in Chapter 8 and Chapter 9. While we describe the application of the same system to the setting of large private networks in Chapter 10.

2.3 Peer-to-peer Connectivity

In the Internet, it is common to encounter a large amount of peers behind Network Address Translators (NATs), typically around to $70-80\%$ [8][66]. The presence of NATs hinders the functioning of peer-to-peer systems because they prevent direct communication between peers which are behind NATs. Conversely, if a peer can't connect to other peers because its NAT prevents it doing so, it can't contribute with its resources to the system and might also be excluded from the service. Consequently, working around NAT connectivity issues is of paramount importance for any peer-to-peer system deployed on the Internet. This is particularly true for content delivery systems, such as P2P live streaming platforms, which rely on large availability of user-contributed bandwidth to provide a good quality of service.

A NAT device allows many hosts to share the same public IP address. Behind the NAT, peers are assigned private addresses. Communication between private and public hosts is possible only if the host behind NAT was the initiator of the communication. NAT routing behavior can be summarized as follows: when a private node tries to contact a host outside the private network, an entry is created in the NAT device's registry, called the NAT translation table, with its destination host's IP and port. The outgoing packets are then translated to appear from the IP address of the NAT device. That is the IP headers of the outgoing packets are re-written to override the private node's IP and port with the public ones of the NAT device. When a response packet is generated by the node outside the private network, the NAT device inspects the source IP and port of the packet and tries to match them using the NAT translation table. If a private host is found which communicated with that source IP and port, the packet is forwarded to the private node, otherwise the packet is dropped without any notification to the originator.

In order to overcome NAT limitations, a number of NAT traversal techniques have been developed in the last decade. These enable connectivity between pairs of peers when one or both of them are behind a NAT. NAT traversal can be carried out both using the TCP and UDP protocol, however it is significantly more effective on the latter [67].

The most widely used NAT traversal protocol for UDP is the STUN protocol [68]. The STUN protocol details a technique known as *simple hole punching* which lets private hosts create NAT entries in their NAT device by sending packets towards a central STUN server which is always available. The STUN server receives the packets and communicates which public IP and port the packets came from to the originator. This information then can be used by other hosts outside the private network to contact the private peer using the previously established NAT entry.

NAT boxes implement a number of behaviors which vary from extremely easy to traverse to extremely difficult [69][70][71]. It is typical for the simple hole punching technique to succeed only on around 40% of the cases [8]. When hole punching fails, most peer-to-peer systems resort to *relaying* though a third-party, which is usually a dedicated server. This approach has been standardized in the TURN RFC [72]. STUN and TURN protocols have also been integrated in a single standard, the ICE RFC [73]. ICE is the de-facto standard in the industry and many commercial peer-to-peer protocols rely on it. Examples of such protocols are WebRTC [74] and the SIP Voice over IP protocol [75]. That said, relaying of packets, while effective for connectivity between peers behind problematic NATs, is extremely expensive since it consumes large amount of bandwidth at the relay side to receive and forward data. For peer-to-peer systems where the content transferred is big, such as live streaming systems, it is not advisable to use relaying.

Besides simple hole punching, a number of other NAT traversal techniques have been developed by the research community [76][71]. These techniques try to exploit detailed knowledge of the NAT device's behavior to establish connections in presence of NAT by using a third party facilitator to help coordinating the establishment process.

Besides NAT Traversal, the other approach for peers behind NAT to become reachable from external hosts is to use application layer protocols such as UPnP [77] and NAT-PMP [78]. These two protocols allow applications to configure static NAT translation entries on NAT devices, such that hosts on the outside of the private network can use the reserved the static entry's IP and port to initiate a communication towards the host that has configured the entry. Unfortunately these protocols are not enabled by default in home routers for security reasons and they are not supported by enterprise NATs.

Research on NAT traversal has not known significant advancements in the last years even though NAT constraints are one of the most relevant issues in peer-to-peer systems, for the simple fact that good connectivity is a precondition for any distributed algorithm to function correctly.

In this thesis, we present in Chapter 5 an extensive classification of NAT behaviors and an elaborate method to explicitly determine which NAT traversal technique should be used for each combination of NAT behaviors. We show that our approach results in a connection establishment success rate of 85% without using relaying or application level protocols such as UPnP and NAT-PMP.

2.4 Peer Sampling

As mentioned earlier, when describing mesh-based peer-to-peer live streaming systems (Section 2.2.1), a peer sampling service (PSS) is a vital component of peer-to-peer live streaming system because it provides the input to overlay construction algorithms.

A peer sampling service (PSS) provides nodes in an overlay with a uniform random sample of live nodes from all nodes in the system, where the size of the sample set or *view* is typically much smaller than the system size. Peer sampling services are widely used in peer-to-peer systems to discover new peers and collect global statistics about the overlay. Also, a PSS can be used as a building block for other services, such as dissemination [85], aggregation [86] as well as overlay construction and maintenance [87].

The most common approach to peer sampling is Gossip. In gossip, peers keep a fixed-size view of the overlay, then periodically select a node from that view and exchange a number of items from the view with it. In [88], a classification of Gossip protocols was presented that categorizes algorithms by each of the following three points: *node selection*, *view propagation* and *view selection*. Node selection defines how a peer chooses which nodes to exchange its view with. Examples of strategies include: random (*rand* strategy) and based on the age of the entry associated with each node in the view. The most common approach is to use the oldest entry (*tail* strategy). View propagation determines if a node only pushes samples to the other node (*push* strategy) or it also receives a number of samples from the destination peer in response (*push-pull* strategy). Finally, view selection defines how a node updates its view after receiving samples from the another node. It can either replace a number of nodes in its view with the ones received in the exchange uniformly at random (*blind* strategy) or replace the oldest in its view (*healer*). In case of push-pull view propagation, another possible strategy is replacing the nodes sent to the other peer with the ones received in the response (*swapper*).

Peer sampling services are typically evaluated by estimating the *global randomness* of samples, that is the amount of correlation between samples returned by the service at different nodes. The common practice for doing that it is to use graph theoretical methods on the overlay graph defined by the set of nodes in the system and their views.

In the overlay topology graph, each peer is a node and there exists an outgoing edge from a node *a* to a node *b* if peer *a* contains peer *b* in its view of the overlay. The goal of this analysis is to estimate how similar the overlay topology graph is to a random graph, that, by assuming that entries in each peer view are a random sample of the set of all nodes in the system.

Two graph-theoretic metrics are used to estimate the global randomness of the overlay graph: the *in-degree* and the *clustering coefficient*. The In-degree of a node is the number of incoming edges that a node has in the overlay graph and it is a measure of how a node is represented in the graph. Intuitively, nodes should not be highly over- or under-represented in the overlay graph to avoid creating hotspots or bottlenecks in the system. On top of that, the in-degree value distribution of all peers should follow a binomial distribution, as it does in random graphs. The clustering coefficient of a node is instead the number of edges between each of the peer's neighbors over the total amount of possible edges between them. Low clustering coefficient is desirable in order to decrease the probability of partitioning of the overlay, that is a cluster becoming isolated from the rest of the overlay.

One common measure of performance of PSSs is freshness of samples. In general, the fresher the samples in a peer's view are, the more probability there is that the nodes in the view are still alive and the information collected is accurate. Age is one freshness metric which, given the samples in a peer's view, estimates the average value of how long ago those samples were generated by the originating nodes. Another metric for freshness in gossip, is the hop count. That is the number of hops the sample in a peer's view has traversed before reaching the peer.

Peer sampling is typically used in peer-to-peer live streaming to provide the membership service in mesh-based systems [46][11][11][28]. The samples produced by the peer sampling service constitute the input for the partnership service which manages overlay construction and maintenance. Consequently, the more reliable and up-to-date the input to the partnership service is, the better the decision about which partners to select for downloading the stream will be and consequently the better the system will perform.

2.4.1 NAT-resilient PSSs

Classical gossip algorithms rely on the assumption that nodes can establish direct communication with any peer in the network with the same probability and this is a necessary pre-condition for ensuring randomness of samples. On the Internet however, this assumption is not true since the majority of peers are behind NATs. Due to the limitations of existing algorithms for NAT traversal, nodes have a different probability of connecting to each others depending on the type of NAT they are behind. Kermarec et al. [89] show that standard gossip-based peer sampling becomes significantly biased in this setting, producing a significant over-representation of nodes that are not behind NAT. Additional results show that the network can become partitioned when the number of private peers exceeds a certain threshold.

In recent years, the community has produced a new class of gossip protocols which take into account the presence of NATs. The proposed approaches have in common the use of open Internet nodes, i.e. not behind NAT, as facilitators for gossip exchanges. Early examples of NAT-aware gossip protocols include [90], Nylon [91] and Gozar [92]. In [90], the authors enabled gossiping with a private node by relaying gossip messages via an open Internet node in the system that had already successfully communicated with that private node. Nylon instead routes packets to a private node using routing tables maintained at all nodes in the system. In contrast, Gozar routes packets to a private node using an existing public node in the system without the need of routing tables as the address of a private node includes the addresses of the public nodes that can act as a relay to it. A more recent NAT-aware peer sampling service is Croupier [93]. Croupier removed the need for relaying gossip messages to private nodes. In Croupier, gossip requests are only sent to public nodes which act as croupiers, shuffling node descriptors on behalf of both public and private nodes.

NAT-aware gossip-based PSSs are based on two assumptions: i) connection establishment from a private to a public peer comes at negligible cost, and ii) the connection setup time is short and predictable. However, these assumptions do not hold for many classes of P2P systems. In particular, in commercial P2P systems such as Spotify [66], P2P-Skype [80], and Google's WebRTC [74] establishing a connection is a relatively complex and costly procedure. This is primarily because security is a concern. All new connections require peers to authenticate the other party with a trusted source, typically a secure server, and to setup an encrypted channel. Another reason is that establishing a new connection may involve coordination by a helper service, for instance, to work around connectivity limitations that are not captured by NAT detection algorithms or that are caused by faulty network configurations.

In our peer-to-peer live streaming system SmoothCache, which we believe is representative of many commercial P2P systems, all these factors combined produce connection setup times which can range from few tenths of a second up to a few seconds, depending on network latencies, congestion, and the complexity of the connection establishment procedure. In addition to these factors, public nodes are vulnerable to denial-of-service attacks, as there exists an upper bound on the rate of new connections that peers are able to establish in a certain period of time.

In this thesis, in Chapter 7, we explore a solution which alleviates the need for frequent connection establishment and still provides the same performance of gossip-based approaches in terms of freshness of samples. The resulting peer sampling approach, which we call the Wormhole Peer Sampling Service (WPSS), is currently used in our SmootchCache system as a membership service.

2.5 Transport

Historically, peer-to-peer applications have utilized the UDP protocol as the transport protocol of choice [79][80][74]. The main reason is that NAT traversal is easier in UDP than in TCP, as described in Section 2.3. In addition to that, UDP is more flexible than TCP because it enables applications to implement their own congestion control system.

For a voice and video conferencing application like Skype this is of paramount importance for instance, as video conferencing and phone calls need a steady low-jitter flow of packets which is really hard to obtain in TCP. On top of that, due to its real-time nature, Skype traffic must benefit from higher priority with respect to any other application's transfer. Thus, Skype developed an application-level proprietary congestion control algorithm [81] known to be very aggressive towards other applications' traffic.

On the other end of the spectrum, a content distribution application like Bittorrent for example, started initially by using TCP but then switched to LEDBAT, in order to be polite as much as possible towards other applications' traffic while saturating the spare link capacity. Politeness was critical to eliminate the reputation of Bittorrent as a protocol which totally hogs the bandwidth and makes all other applications starve. Peer-to-peer live streaming applications are another example of peer-to-peer applications which also implement their own congestion control over UDP. Designs and implementations vary for each platform and there is currently no standard approach to the problem of congestion control management in PLS.

As part of the work presented in this thesis, we designed, developed and evaluated a multi-purpose transport library, called DTL [9], which can be used for any kind of peer-to-peer applications, be that a file-sharing or a live streaming applications. Given the different requirements of P2P applications, we implement a flexible congestion control mechanism which allows for runtime prioritization of transfers. Priority varies from lower-than-best-effort to up to four times the priority of standard TCP. To achieve this goal, the library seamlessly combines two different congestion control mechanisms: LEDBAT and MulTCP.

LEDBAT is widely accepted as an effective solution to provide a less-than-best-effort data transfer service. Initially implemented in the μTorrent BitTorrent client and now separately under discussion as an IETF draft [82], LEDBAT is a delay-based congestion control mechanism which aims at saturating the bottleneck link while throttling back transfers in the presence of flows created by other applications, such as games or VoIP applications. The yielding procedure is engineered to avoid disruptions to other traffic in the network, and it is based on the assumption that growing one-way delays are symptoms of network congestion. With respect to classical TCP, this allows for earlier congestion detection.

MulTCP [83] instead provides a mechanism for changing the increment and decrement parameters of TCP's Additive increase/multiplicative decrease (AIMD) [84] algorithm to emulate the behavior of a fixed number N of TCP flows in a single transfer.

In our peer-to-peer live streaming application SmootchCache, we use DTL to prefetch fragments from partners in the overlay as soon as they become available. Prefetching of fragments happens with lower-than-best-effort priority, while we progressively increase the priority of transfers as the fragment nears the playback deadline.

2.6 Development and Evaluation of Peer-To-Peer Systems

Peer-to-Peer (P2P) systems are inherently difficult to design and implement correctly. Even a very few lines of code running simultaneously on a large number of peers result in interactions that are rather challenging to understand and debug. Despite this, while algorithms and research ideas about P2P systems are abundant, software engineering practices of developing P2P systems, especially in an industrial setting, are less known and shared. In fact, to the best of our knowledge, there exists no comprehensive tool for efficiently implementing, evaluating and deploying distributed algorithms. Our experience indicates that such tools should provide the following features:

- **Simulation-based development**. P2P algorithms are in general complex due to the high amount of exchanged messages, asynchrony and the fact that failures are the norm rather than the exception. Simulation is therefore essential when validating P2P protocol interactions. A common practice in distributed systems development is to develop a prototype of the application first, then validate it and finally re-implement the validated algorithms in another environment for deployment. This leads to two main problems: a dual code base which is very cumbersome to maintain and not being able to evaluate deployment code in a reproducible fashion, i.e. executing the same test scenario many times while preserving the exact same sequence of events.

  For solving the first issue, it is common practice [94][6] to resort to a framework which lets developers prototype algorithms as independent components in a simulator and then easily deploy the same components on a real environment by using another runtime. In such framework, the network library for instance could appear to those components as another component with a single interface, however two implementations would be provided, one real, for deployment, and one simulated, for evaluation.

  For the issue of executing deployment code in a reproducible fashion, a common solution [16][95] is to inject a Discrete Event Simulator (DES) underneath the application components. The DES makes possible to serialize all network and internal component interactions in order to obtain the same sequence of events every time the system is executed in simulation.

- **Message-passing concurrency model**. Message-passing concurrency is the most commonly used programming model in the distributed systems community, as opposed to shared-state concurrency. Message-passing not only scales well with multi-core hardware but it makes easier to reason about concurrency. A popular set of semantics for message-passing in distributed systems is the actor model [96], with programming languages like Erlang [97] and Scala [98] providing support out-of-the-box for it. In the actor model, each process or *actor* communicates with other actors using asynchronous messages. This model fits well the semantics of peer-to-peer protocols where a computation is very often triggered at a node by a message coming from the network. The actor model is inherently concurrent as many actors can be executed at the same time and therefore exploit parallel hardware.

Some development frameworks based on the actor model, such as Kompics [95], provide composability of actor systems. That enables different actors systems to be combined into a larger one in order to extend functionality. Once again, this conveniently matches the common practice in development of distributed systems of combining different protocols to build a larger platform.

- **More Realistic Network Model**. When evaluating/debugging peer-to-peer applications in simulation, it is extremely important to model the behavior of the underlying network in the right way since it directly affects the performance of the system.

  Traditionally peer-to-peer simulators emulate only certain aspects of the physical network, such as delay and packet loss [99] [95]. We argue that this is not sufficient when developing complex systems such as content delivery applications. In those applications, connectivity and bandwidth allocation dynamics play a major role in affecting the performance of the system. For this reason, any complete tool for peer-to-peer applications should contain realistic models for those two physical network characteristics.

As part of this thesis work, we present a framework, called Mesmerizer, which encompasses all best practices which we presented above and that we followed for the development of our peer-to-peer based products: SmoothCache, SmoothCache 2.0 and Peer2View [14]. Mesmerizer is a framework that follows the actor programming model, where components (actors) exchange asynchronous events (messages) with other components on the same machine or on a remote machine. Every component runs in its own thread of control, assigned temporally from a thread pool, when the handling an event. On the same machine, events are buffered in a First-In-First-Out (FIFO) queue at the destination component before being handled by that component, consequently decoupling the generation of the event from the handling of events. Events that are sent to remote instances of components are also delivered in order.

Mesmerizer allows peer-to-peer developers to implement their algorithms once and have them running in simulation and on real deployments. In simulation, it provides a Discrete Event Simulation (DES) runtime which delivers reproducibility and accurate modeling of physical network behavior. In a real deployment, it provides a multithreaded runtime and the possibility of using the DTL library, introduced in Section 2.5 and presented in Chapter 6, as transport for the overlying application.

The various iterations of SmoothCache system were developed in the Mesmerizer framework and they all have have been used in a commercial application.

2.6.1 Emulation of Bandwidth Allocation Dynamics

As mentioned in the previous section, bandwidth modeling is an important tool when testing and evaluating P2P applications. This is particularly true for peer-to-peer content delivery platforms such as PLS systems and file-sharing applications. In those, each peer implements intricate multiplexing strategies to speed up transmission of large chunks of data, thus creating complex effects on the underlying physical network which translate into varying transmission delays and packet loss. These aspects can be accurately reproduced in a simulator by a bandwidth dynamics model.

When emulating bandwidth dynamics, there usually exists a trade-off between *scalability* and *accuracy*. That means that accurate network simulators can usually scale up only to a limited number of simulated peers. This limitation makes it unfeasible to capture the behavior and issues of a larger P2P real-word deployment, such as the effect network congestion on segments of the overlay network. In order to achieve scalability, most of the existing P2P simulators abstract away network interactions by modeling only the structural dynamics of the overlay network and thus completely ignoring the impact of the actual network on application performance.

The best level of accuracy can be achieved with packet-level simulators such as NS-2 [100] or P2PSim[101]. Those model the behavior of each single low level, network layer packet. The penalty for detailed modeling comes in scalability: a single transfer between two peers can generate hundreds of low-level protocol packets which need to be modeled individually. In a large and complex network characterized by links of high bandwidth capacity and intricate multi-peer interactions, the sheer number of events needed to simulate transfers requires a prohibitive amount of computational and memory resources. As a result, only small sized networks can be simulated efficiently with packet-level simulators.

The best trade-off between accuracy and scalability is provided by flow-level simulation, which focuses on a transfer as a whole rather than individual packets. A flow abstracts away the small time scale rate variation of a packet sequence with a constant rate allocated at the sender/receiver's bandwidth. The rate remains allocated for an amount of time which corresponds to the duration of the flow, i.e. the simulated packet sequence transmission time. This approach reduces drastically the number of events to be simulated.

However, our experience with flow-based models is that the desired scalability for peer-to-peer simulations, that is in the order of thousands of node instances, cannot be achieved with state of the art approaches [102][103]. Besides that, the accuracy of the proposed solutions in the state of the art has never been asserted.

As part of this thesis work, in Chapter 4, we have developed a model for estimating bandwidth allocation dynamics which is more scalable that the state of the art. On top of that, we have evaluated its accuracy by conducting a detailed study on realistic scenarios against a packet-level simulator (NS-2). There, we show that the bandwidth allocation dynamics in the model follow the ones of the packet-level simulator.

2.7 Bibliography

[1] 2013 Cisco VNI. Cisco visual networking index: Forecast and methodology, 2012 to 2017, 2012. URL `www.cisco.com/en/US/solutions/collateral/ns341/`

ns525/ns537/ns705/ns827/white_paper_c11-481360.pdf.

[2] Adobe http dynamic streaming protocol. http://www.adobe.com/products/hds-dynamic-streaming.html.

[3] Apple Inc. HTTP Live Streaming. http://developer.apple.com/resources/http-streaming/.

[4] Microsoft Inc. Smooth Streaming. http://www.iis.net/download/SmoothStreaming.

[5] Mpeg-dash protocol. http://dashif.org/mpeg-dash/.

[6] Roberto Roverso, Sameh El-Ansary, Alexandros Gkogkas, and Seif Haridi. Mesmerizer: a effective tool for a complete peer-to-peer software development life-cycle. In *Proceedings of the 4th International ACM/ICST Conference on Simulation Tools and Techniques*, SIMUTools '11, pages 506–515, ICST, Brussels, Belgium, Belgium, 2011.

[7] Alexandros Gkogkas, Roberto Roverso, and Seif Haridi. Accurate and efficient simulation of bandwidth dynamics for peer-to-peer overlay networks. In *Proceedings of the 5th International ICST Conference on Performance Evaluation Methodologies and Tools*, VALUETOOLS '11, pages 352–361, ICST, Brussels, Belgium, Belgium, 2011. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[8] Roberto Roverso, Sameh El-Ansary, and Seif Haridi. NATCracker: NAT Combinations Matter. In *Proceedings of 18th International Conference on Computer Communications and Networks 2009*, ICCCN '09, pages 1–7, San Francisco, CA, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-4581-3.

[9] Riccardo Reale, Roberto Roverso, Sameh El-Ansary, and Seif Haridi. Dtl: Dynamic transport library for peer-to-peer applications. In Luciano Bononi, AjoyK. Datta, StÃ©phane Devismes, and Archan Misra, editors, *Distributed Computing and Networking*, volume 7129 of *Lecture Notes in Computer Science*, pages 428–442. Springer Berlin Heidelberg, 2012.

[10] R. Roverso, J. Dowling, and M. Jelasity. Through the wormhole: Low cost, fresh peer sampling for the internet. In *Peer-to-Peer Computing (P2P), 2013 IEEE 13th International Conference on*, 2013.

[11] Roberto Roverso, Sameh El-Ansary, and Seif Haridi. Smoothcache: Http-live streaming goes peer-to-peer. In Robert Bestak, Lukas Kencl, LiErran Li, Joerg Widmer, and Hao Yin, editors, *NETWORKING 2012*, volume 7290 of *Lecture Notes in Computer Science*, pages 29–43. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-30053-0.

[12] R. Roverso, S. El-Ansary, and S. Haridi. Peer2view: A peer-to-peer http-live streaming platform. In *Peer-to-Peer Computing (P2P), 2012 IEEE 12th International Conference on*, pages 65–66, 2012.

[13] Roberto Roverso, Sameh El-Ansary, and Mikael Hoeqvist. On http live streaming in large enterprises. In *Proceedings of the ACM SIGCOMM 2013 conference on Applications, technologies, architectures, and protocols for computer communication*, SIGCOMM '13, New York, NY, USA, 2013. ACM.

[14] Roberto Roverso, Amgad Naiem, Mohammed Reda, Mohammed El-Beltagy, Sameh El-Ansary, Nils Franzen, and Seif Haridi. On The Feasibility Of Centrally-Coordinated Peer-To-Peer Live Streaming. In *Proceedings of IEEE Consumer Communications and Networking Conference 2011*, Las Vegas, NV, USA, January 2011.

[15] Roberto Roverso, Amgad Naiem, Mohammed El-Beltagy, Sameh El-Ansary, and Seif Haridi. A GPU-enabled solver for time-constrained linear sum assignment problems. In *Informatics and Systems (INFOS), 2010 The 7th International Conference on*, pages 1–6, Cairo, Egypt, 2010. IEEE Computer Society. ISBN 978-1-4244-5828-8.

[16] Roberto Roverso, Mohammed Al-Aggan, Amgad Naiem, Andreas Dahlstrom, Sameh El-Ansary, Mohammed El-Beltagy, and Seif Haridi. MyP2PWorld: Highly Reproducible Application-Level Emulation of P2P Systems. In *Proceedings of the 2008 Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops*, pages 272–277, Venice, Italy, 2008. IEEE Computer Society. ISBN 978-0-7695-3553-1.

[17] Roberto Roverso, Cosmin Arad, Ali Ghodsi, and Seif Haridi. DKS: Distributed K-Ary System a Middleware for Building Large Scale Dynamic Distributed Applications, Book Chapter. In *Making Grids Work*, pages 323–335. Springer US, 2007. ISBN 978-0-387-78447-2.

[18] H. Schulzrinne, A. Rao, and R. Lanphier. RTSP: Real-Time Streaming Protocol. RFC 2326 (Proposed Standard), 1998. URL `http://www.ietf.org/rfc/rfc2326.txt`.

[19] Real Data Transport. https://helixcommunity.org/viewcvs/server/protocol/transport/rdt/.

[20] Saamer Akhshabi, Ali C. Begen, and Constantine Dovrolis. An experimental evaluation of rate-adaptation algorithms in adaptive streaming over HTTP. In *Proceedings of the second annual ACM conference on Multimedia systems*, MMSys, 2011.

[21] Saamer Akhshabi, Sethumadhavan Narayanaswamy, Ali C. Begen, and Constantine Dovrolis. An experimental evaluation of rate-adaptive video players over {HTTP}. *Signal Processing: Image Communication*, 27(4):271 – 287, 2012. ISSN

0923-5965. URL http://www.sciencedirect.com/science/article/pii/S0923596511001159.

[22] Saamer Akhshabi, Lakshmi Anantakrishnan, Ali C. Begen, and Constantine Dovrolis. What happens when http adaptive streaming players compete for bandwidth? In *Proceedings of the 22nd international workshop on Network and Operating System Support for Digital Audio and Video*, NOSSDAV '12, pages 9–14, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1430-5. URL http://doi.acm.org/10.1145/2229087.2229092.

[23] Xiaojun Hei, Chao Liang, Jian Liang, Yong Liu, and Keith W. Ross. A measurement study of a large-scale P2P IPTV system. *Multimedia, IEEE Transactions on*, 2007.

[24] John Jannotti, David K. Gifford, Kirk L. Johnson, M. Frans Kaashoek, and James W. O'Toole, Jr. Overcast: Reliable Multicasting with an Overlay Network. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, pages 14–14, Berkeley, CA, USA, 2000. USENIX Association. URL http://portal.acm.org/citation.cfm?id=1251229.1251243.

[25] Kunwoo Park, Sangheon Pack, and Taekyoung Kwon. Climber: an incentive-based resilient peer-to-peer system for live streaming services. In *Proceedings of the 7th international conference on Peer-to-peer systems*, IPTPS'08, pages 10–10, Berkeley, CA, USA, 2008. USENIX Association. URL http://portal.acm.org/citation.cfm?id=1855641.1855651.

[26] D. Tran, K. Hua, and S. Sheu. ZIGZAG: An Efficient Peer-to-Peer Scheme for Media Streaming. In *Proc. of IEEE INFOCOM*, 2003.

[27] Marc Schiely, Lars Renfer, and Pascal Felber. Self-organization in cooperative content distribution networks. In *Network Computing and Applications, Fourth IEEE International Symposium on*, pages 109–118. IEEE, 2005.

[28] Amir H. Payberah, Jim Dowling, Fatemeh Rahimian, and Seif Haridi. gradienTv: Market-based P2P Live Media Streaming on the Gradient Overlay. In *Lecture Notes in Computer Science (DAIS 2010)*, pages 212–225. Springer Berlin / Heidelberg, Jan 2010. ISBN 978-3-642-13644-3.

[29] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. Splitstream: high-bandwidth multicast in cooperative environments. In *SIGOPS Oper. Syst. Rev.*, volume 37, pages 298–313, New York, NY, USA, October 2003. ACM. URL http://doi.acm.org/10.1145/1165389.945474.

[30] J.J.D. Mol, D.H.J. Epema, and H.J. Sips. The Orchard Algorithm: P2P Multicasting without Free-Riding. *Peer-to-Peer Computing, IEEE International Conference on*, 0:275–282, 2006.

[31] Meng Zhang, Yun Tang, Li Zhao, Jian-Guang Luo, and Shi-Qiang Yang. Gridmedia: A Multi-Sender Based Peer-to-Peer Multicast System for Video Streaming. In *Multimedia and Expo, 2005. ICME 2005. IEEE International Conference on*, pages 614–617, 2005. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1521498`.

[32] N. Magharei, R. Rejaie, and Yang Guo. Mesh or multiple-tree: A comparative study of live p2p streaming approaches. In *INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE*, pages 1424–1432, 2007.

[33] R. Roverso, A. Naiem, M. Reda, M. El-Beltagy, S. El-Ansary, N. Franzen, and S. Haridi. On the feasibility of centrally-coordinated peer-to-peer live streaming. In *Consumer Communications and Networking Conference (CCNC), 2011 IEEE*, pages 1061–1065, 2011.

[34] Venkata N. Padmanabhan and Kunwadee Sripanidkulchai. The case for cooperative networking. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 178–190, London, UK, 2002. Springer-Verlag. ISBN 3-540-44179-4. URL `http://portal.acm.org/citation.cfm?id=646334.758993`.

[35] Ryan S. Peterson and Emin Gün Sirer. Antfarm: efficient content distribution with managed swarms. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 107–122, Berkeley, CA, USA, 2009. USENIX Association. URL `http://portal.acm.org/citation.cfm?id=1558977.1558985`.

[36] Vidhyashankar Venkataraman, Kaouru Yoshida, and Paul Francis. Chunkyspread: Heterogeneous Unstructured Tree-Based Peer-to-Peer Multicast. *Proceedings of the Proceedings of the 2006 IEEE International Conference on Network Protocols*, pages 2–11, 2006. URL `http://portal.acm.org/citation.cfm?id=1317535.1318351`.

[37] SopCast. http://sopcast.com.

[38] Xinyan Zhang, Jiangchuan Liu, Bo Li, and Y. S. P. Yum. CoolStreaming/DONet: a data-driven overlay network for peer-to-peer live media streaming. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, volume 3, pages 2102–2111 vol. 3, March 2005. URL `http://dx.doi.org/10.1109/INFCOM.2005.1498486`.

[39] Vinay Pai, Kapil Kumar, Karthik Tamilmani, Vinay Sambamurthy, and Alexander Mohr. Chainsaw: Eliminating trees from overlay multicast. In *Peer-to-Peer Systems IV*, volume Volume 3640/2005, pages 127–140. Springer Berlin / Heidelberg, 2005. URL `http://dx.doi.org/10.1007/11558989_12`.

[40] A. Vlavianos, M. Iliofotou, and M. Faloutsos. Bitos: Enhancing bittorrent for supporting streaming applications. In *9th IEEE Global Internet Symposium 2006*, April 2006.

[41] Fabio Pianese, Diego Perino, Joaquin Keller, and Ernst W. Biersack. Pulse: An adaptive, incentive-based, unstructured p2p live streaming system. *IEEE Transactions on Multimedia*, 9(8):1645–1660, December 2007. ISSN 1520-9210. URL `http://dx.doi.org/10.1109/TMM.2007.907466`.

[42] David Kempe, Jon Kleinberg, and Alan Demers. Spatial gossip and resource location protocols. In *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, STOC '01, pages 163–172, New York, NY, USA, 2001. ACM. ISBN 1-58113-349-9. URL `http://doi.acm.org/10.1145/380752.380796`.

[43] Feng Wang, Yongqiang Xiong, and Jiangchuan Liu. mTreebone: A Hybrid Tree/Mesh Overlay for Application-Layer Live Video Multicast. In *Proceedings of the 27th International Conference on Distributed Computing Systems*, ICDCS '07, pages 49–, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2837-3. URL `http://dx.doi.org/10.1109/ICDCS.2007.122`.

[44] Shah Asaduzzaman, Ying Qiao, and Gregor von Bochmann. CliqueStream: an efficient and fault-resilient live streaming network on a clustered peer-to-peer overlay. *CoRR*, abs/0903.4365, 2009.

[45] N. Magharei and R. Rejaie. PRIME: Peer-to-Peer Receiver-drIven MEsh-Based Streaming. In *INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE*, pages 1415–1423, May 2007. URL `http://dx.doi.org/10.1109/INFCOM.2007.167`.

[46] B. Li, Y. Qu, Y. Keung, S. Xie, C. Lin, J. Liu, and X. Zhang. Inside the New Coolstreaming: Principles, Measurements and Performance Implications. In *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*, 2008.

[47] Fabio Picconi and Laurent Massoulié. Is there a future for mesh-based live video streaming? In *Proceedings of the 2008 Eighth International Conference on Peer-to-Peer Computing*, pages 289–298, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3318-6. URL `http://portal.acm.org/citation.cfm?id=1443220.1443468`.

[48] D. Ciullo, M.A. Garcia, A. Horvath, E. Leonardi, M. Mellia, D. Rossi, M. Telek, and P. Veglia. Network awareness of p2p live streaming applications: A measurement study. *Multimedia, IEEE Transactions on*, 12(1):54–63, 2010. ISSN 1520-9210.

[49] Aukrit Chadagorn, Ibrahim Khalil, Conor Cameron, and Zahir Tari. Pilecast: Multiple bit rate live video streaming over bittorrent. *Journal of Network and Computer Applications*, 2013. ISSN 1084-8045.

[50] Yipeng Zhou, Dah Ming Chiu, and John CS Lui. A simple model for analyzing p2p streaming protocols. In *Network Protocols, 2007. ICNP 2007. IEEE International Conference on*, pages 226–235. IEEE, 2007.

[51] Vincenzo Ciancaglini. From key-based to content-based routing: system interconnection and video streaming applications, July 2013.

[52] Meng Zhang, Jian-Guang Luo, Li Zhao, and Shi-Qiang Yang. A peer-to-peer network for live media streaming using a push-pull approach. In *Proceedings of the 13th annual ACM international conference on Multimedia*, pages 287–290. ACM, 2005.

[53] Hao Yin, Xuening Liu, Tongyu Zhan, Vyas Sekar, Feng Qiu, Chuang Lin, Hui Zhang, and Bo Li. Livesky: Enhancing cdn with p2p. *ACM Trans. Multimedia Comput. Commun. Appl.*, 6:16:1–16:19, August 2010. ISSN 1551-6857. URL `http://doi.acm.org/10.1145/1823746.1823750`.

[54] Feng Wang, Jiangchuan Liu, and Minghua Chen. Calms: Cloud-assisted live media streaming for globalized demands with time/region diversities. In *INFOCOM, 2012 Proceedings IEEE*, pages 199–207, 2012.

[55] Amir H Payberah, Hanna Kavalionak, Vimalkumar Kumaresan, Alberto Montresor, and Seif Haridi. Clive: Cloud-assisted p2p live streaming. In *Peer-to-Peer Computing (P2P), 2012 IEEE 12th International Conference on*, pages 79–90. IEEE, 2012.

[56] Raymond Sweha, Vatche Ishakian, and Azer Bestavros. Angelcast: cloud-based peer-assisted live streaming using optimized multi-tree construction. In *Proceedings of the 3rd Multimedia Systems Conference*, pages 191–202. ACM, 2012.

[57] Xuening Liu, Hao Yin, Chuang Lin, Yu Liu, Zhijia Chen, and Xin Xiao. Performance analysis and industrial practice of peer-assisted content distribution network for large-scale live video streaming. In *Advanced Information Networking and Applications, 2008. AINA 2008. 22nd International Conference on*, pages 568–574, 2008.

[58] Mingchen Zhao, Paarijaat Aditya, Ang Chen, Yin Lin, Andreas Haeberlen, Peter Druschel, Bruce Maggs, Bill Wishon, and Miroslav Ponec. Peer-assisted content distribution in akamai netsession. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 31–42. ACM, 2013.

[59] Adobe. Rtmfp for developing real-time collaboration applications, 2013. URL `http://labs.adobe.com/technologies/cirrus/`.

[60] Adobe Labs. Voddler, 2013. URL `https://www.adobe.com/content/dam/Adobe/en/casestudies/air/voddler/pdfs/voddler-casestudy.pdf`.

[61] Cheng Huang, Jin Li, and Keith W Ross. Peer-assisted vod: Making internet video distribution cheap. In *IPTPS*, 2007.

[62] Cheng Huang, Jin Li, and Keith W. Ross. Can internet video-on-demand be profitable? *SIGCOMM Comput. Commun. Rev.*, 37(4):133–144, August 2007. ISSN 0146-4833. URL http://doi.acm.org/10.1145/1282427.1282396.

[63] Liang Zhang, Fangfei Zhou, Alan Mislove, and Ravi Sundaram. Maygh: Building a cdn from client web browsers. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 281–294. ACM, 2013.

[64] Roberto Roverso, Riccardo Reale, Sameh El-Ansary, and Seif Haridi. Smoothcache 2.0: the http-live peer-to-peer cdn. In *Report*, volume 7290 of *Peerialism White Papers*, 2013.

[65] Dario Rossi, Claudio Testa, Silvio Valenti, and Luca Muscariello. LEDBAT: The New BitTorrent Congestion Control Protocol. In *Computer Communications and Networks (ICCCN), 2010 Proceedings of 19th International Conference on*, pages 1–6, August 2010. URL http://dx.doi.org/10.1109/ICCCN.2010.5560080.

[66] Gunnar Kreitz and Fredrik Niemela. Spotify – large scale, low latency, P2P Music-on-Demand streaming. In *Tenth IEEE International Conference on Peer-to-Peer Computing (P2P'10)*. IEEE, August 2010. ISBN 978-1-4244-7140-9.

[67] Saikat Guha and Paul Francis. Characterization and measurement of tcp traversal through nats and firewalls. In *IMC '05: Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement*, pages 18–18, Berkeley, CA, USA, 2005. USENIX Association.

[68] J. Rosenberg, J. Weinberger, C. Huitema, and R. Mahy. STUN - Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs). RFC 3489 (Proposed Standard), March 2003. URL http://www.ietf.org/rfc/rfc3489.txt. Obsoleted by RFC 5389.

[69] P. Srisuresh, B. Ford, and D. Kegel. State of Peer-to-Peer (P2P) Communication across Network Address Translators (NATs). RFC 5128 (Informational), March 2008. URL http://www.ietf.org/rfc/rfc5128.txt.

[70] C.Jennings. Nat classification test results. Internet Draft, July 2007. URL http://merlot.tools.ietf.org/search/draft-jennings-behave-test-results-04.

[71] Y. Takeda. Symmetric nat traversal using stun. Internet draft, June 2003. URL http://tools.ietf.org/html/draft-takeda-symmetric-nat-traversal-00.

[72] C. Huitema J. Rosenberg, R. Mahy. Traversal Using Relays around NAT (TURN): Relay extensions to session traversal utilities for NAT (STUN). Internet draft, November 2008. URL http://tools.ietf.org/html/draft-ietf-behave-turn-14.

[73] J. Rosenberg. Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols. Internet draft, October 2007. URL `http://tools.ietf.org/html/draft-ietf-mmusic-ice-19`.

[74] Alan B. Johnston and Daniel C. Burnett. *WebRTC: APIs and RTCWEB Protocols of the HTML5 Real-Time Web*. Digital Codex LLC, USA, 2012. ISBN 0985978805, 9780985978808.

[75] C. Boulton, J. Rosenberg, G. Camarillo, and F. Audet. Nat traversal practices for client-server sip. RFC 6314 (Proposed Standard), 2011. URL `http://tools.ietf.org/html/rfc6314`.

[76] Bryan Ford, Pyda Srisuresh, and Dan Kegel. Peer-to-peer communication across network address translators. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 13–13, Berkeley, CA, USA, 2005. USENIX Association.

[77] M. Boucadair. Universal plug and play (upnp) internet gateway device - port control protocol interworking function (igd-pcp iwf). RFC 6970, 2013. URL `https://tools.ietf.org/html/rfc6970`.

[78] M. Krochmal S. Cheshire. Nat port mapping protocol (nat-pmp). RFC 6886, April 2013. URL `https://tools.ietf.org/html/rfc6886`.

[79] Dario Rossi, Claudio Testa, and Silvio Valenti. Yes, we ledbat: Playing with the new bittorrent congestion control algorithm. In *Passive and Active Measurement*, pages 31–40. Springer, 2010.

[80] Skype Inc. Skype. http://www.skype.com/.

[81] Luca Cicco, Saverio Mascolo, and Vittorio Palmisano. An experimental investigation of the congestion control used by skype voip. In *Proceedings of the 5th international conference on Wired/Wireless Internet Communications*, WWIC '07, pages 153–164, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-72694-4.

[82] Ledbat ietf draft. `http://tools.ietf.org/html/draft-ietf-ledbat-congestion-03`, July 2010.

[83] Jon Crowcroft and Philippe Oechslin. Differentiated end-to-end internet services using a weighted proportional fair sharing tcp. *CoRR*, cs.NI/9808004, 1998.

[84] M. Allman, V. Paxson, and E. Blanton. Tcp congestion control, 9 2009. URL `http://www.ietf.org/rfc/rfc5681.txt`.

[85] P Th Eugster, Rachid Guerraoui, Sidath B Handurukande, Petr Kouznetsov, and A-M Kermarrec. Lightweight probabilistic broadcast. *ACM Transactions on Computer Systems (TOCS)*, 21(4):341–374, 2003.

[86] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Transactions on Computer Systems (TOCS)*, 23 (3):219–252, 2005.

[87] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. T-man: Gossip-based fast overlay topology construction. *Computer networks*, 53(13):2321–2339, 2009.

[88] Márk Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. Gossip-based peer sampling. *ACM Trans. Comput. Syst.*, 25 (3):8, 2007. ISSN 0734-2071.

[89] Anne-Marie Kermarrec, Alessio Pace, Vivien Quema, and Valerio Schiavoni. NAT-resilient gossip peer sampling. In *Proc. 29th IEEE Intl. Conf. on Distributed Computing Systems*, pages 360–367. IEEE Comp. Soc., 2009.

[90] J. Leitão, R. van Renesse, and L. Rodrigues. Balancing gossip exchanges in networks with firewalls. In Michael J. Freedman and Arvind Krishnamurthy, editors, *Proceedings of the 9th International Workshop on Peer-to-Peer Systems (IPTPS '10)*, page 7. USENIX, 2010.

[91] Anne-Marie Kermarrec, Alessio Pace, Vivien Quema, and Valerio Schiavoni. Nat-resilient gossip peer sampling. In *ICDCS 2009*, pages 360–367, Washington, DC, USA, 2009. IEEE Computer Society.

[92] Amir H. Payberah, Jim Dowling, and Seif Haridi. Gozar: Nat-friendly peer sampling with one-hop distributed nat traversal. In Pascal Felber and Romain Rouvoy, editors, *DAIS*, Lecture Notes in Computer Science, 2011.

[93] Jim Dowling and Amir H. Payberah. Shuffling with a croupier: Nat-aware peer-sampling. *ICDCS*, 0:102–111, 2012. ISSN 1063-6927.

[94] Despotovic Galuba, Aberer and Kellerer. Protopeer: A p2p toolkit bridging the gap between simulation and live deployement. *2nd International ICST Conference on Simulation Tools and Techniques*, May 2009.

[95] C. Arad, J. Dowling, and S. Haridi. Building and evaluating p2p systems using the kompics component framework. In *Peer-to-Peer Computing, 2009. P2P '09. IEEE Ninth International Conference on*, pages 93 –94, 2009.

[96] Gul Abdulnabi Agha. Actors: a model of concurrent computation in distributed systems. MIT Press, 1985.

[97] Joe Armstrong. Erlang - a survey of the language and its industrial applications. In *In Proceedings of the symposium on industrial applications of Prolog (INAP96). 16 18*, 1996.

[98] Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, 410(2-3):202–220, 2009. ISSN 0304-3975.

[99] Alberto Montresor Mark Jelasity and Ozalp Babaoglu. A modular paradigm for building self-organizing peer-to-peer applications. In *Engineering Self-Organising Systems*, pages 265–282, 2003.

[100] The ns-2 network simulator. http://www.isi.edu/nsnam/ns/, October 2010.

[101] Thomer M. Gil, Frans Kaashoek, Jinyang Li, Robert Morris, and Jeremy Stribling. p2psim: a simulator for peer-to-peer (p2p) protocols. http://pdos.csail.mit.edu/p2psim/, October 2006.

[102] F. Lo Piccolo, G. Bianchi, and S. Cassella. Efficient simulation of bandwidth allocation dynamics in p2p networks. In *GLOBECOM '06: Proceedings of the 49th Global Telecommunications Conference*, pages 1–6, San Franscisco, California, November 2006.

[103] Anh Tuan Nguyen and F. Eliassen. An efficient solution for max-min fair rate allocation in p2p simulation. In *ICUMT '09: Proceedings of the International Conference on Ultra Modern Telecommunications Workshops*, pages 1 –5, St. Petersburg, Russia, October 2009.

PART II

Framework and Tools

CHAPTER 3

Mesmerizer: An Effective Tool for a Complete Peer-to-peer Software Development Life-cycle

Mesmerizer: An Effective Tool for a Complete Peer-to-Peer Software Development Life-cycle

Roberto Roverso[1,2], Sameh El-Ansary[1], Alexandros Gkogas[1] and Seif Haridi[2]

[1]Peerialism AB
Stockholm, Sweden
[roberto,sameh,alex]@peerialism.com

[2]KTH - Royal Institute of Technology
Stockholm, Sweden
[haridi]@kth.se

Abstract

In this paper we present what are, in our experience, the best practices in Peer-To-Peer(P2P) application development and how we combined them in a middleware platform called Mesmerizer. We explain how simulation is an integral part of the development process and not just an assessment tool. We then present our component-based event-driven framework for P2P application development, which can be used to execute multiple instances of the same application in a strictly controlled manner over an emulated network layer for simulation/testing, or a single application in a concurrent environment for deployment purpose. We highlight modeling aspects that are of critical importance for designing and testing P2P applications, e.g. the emulation of Network Address Translation and bandwidth dynamics. We show how our simulator scales when emulating low-level bandwidth characteristics of thousands of concurrent peers while preserving a good degree of accuracy compared to a packet-level simulator.

3.1 Introduction

Peer-to-Peer (P2P) systems have passed through a number of evolution eras. Starting from being an exotic practice in hacker communities to a rigorously researched and decently funded academic field. Nowadays, products based on P2P technologies such as Bittorrent and Skype are mainstream brands in internet technologies. Despite of that, while algorithms and research ideas about P2P systems are abundant, software engineering practices of developing P2P systems, especially in an industrial setting are less shared. Compared with the process of developing web-based applications, the amount of best practices that has been iterated, re-factored and publicly shared within the communities is huge. Examples include model-driven frameworks such as Ruby on Rails [1] or Django [2] or communication patters like AJAX [3] and COMET [4] and their variants. We argue that while the art of developing P2P applications in terms of shear algorithmic complexity is far beyond web-applications, there are very few best practices shared on how to develop P2P systems.

The point of this paper is to the share best practices that worked for Peerialism. We do that by articulating three main areas where we think we have gained maturity. The first is simulation tools. Namely, how they are an integral part of the software development and not just an assessment tool. We highlight how a clear concurrency model can significantly simplify development and then which modeling aspects are critical for a successful P2P system design and implementation process.

Simulation-Based Development cycle. P2P algorithms are in general complex due to the high amount of exchanged messages, asynchrony and the fact that failures are the norm rather than the exception. Consequently, simulation is not a luxury but a necessity when validating P2P protocol interactions. Even a very few lines of code running simultaneously on one thousand peers result in interactions that are rather challenging to debug. We started with the common practice of authoring the algorithms on our own discrete-event simulator and, when the algorithms were mature enough, we transitioned to real implementation. However, maintenance of a dual code base and irreproducibility of bugs were main concerns that led us to attempt injecting a discrete event simulator underneath our production code. The results of this effort, where mainly a simulated network, simulated time and simulated threading were provided, were published in the MyP2PWorld system [5]. The main advantage of the approach was that developers wrote exactly in the same style they were familiar with. This approach made it possible to debug complex interactions of hundreds of peers on a single development machine and also share reproducible scenarios with the development team. In a sense, the simulated mode served as an extremely comprehensive integration testing tool. That is an achievement which is hard to obtain in uncontrolled, real-world deployments.

Over time, we found that we are using component-based frameworks like Guice [6] extensively to organize and decouple various parts of our code. We realized quickly that the task of switching between real and simulated runs could be achieved in a more elegant fashion using component frameworks where for example the network is a component with one interface and two implementations, one real and one simulated. We noticed that others in the field have later independently reached the same conclusion and we see systems like Kompics [7] and Protopeer [8] which adapted the same practice. We expect more wide dissemination of systems like these in the future.

Message-passing Concurrency Model. In general, the classical way of dealing with concurrency is to use threads and locks, that is shared-state concurrency. For I/O intensive applications, the network programming community has been advocating the message-passing concurrency model. That is more or less an established consensus. In a programming language like Java, one could observe the transition of the standard library to provide more off-the-shelf code for asynchronous I/O based on Java NIO [9]. With that model, any network related activity is done in an message-passing fashion, while the rest of the concurrent modules in the application are written using shared-state concurrency, using threads and locks. Our first application was developed in such a way; the co-existence shared-state concurrency and message-passing concurrency models rendered the applications much harder to design and maintain. The practice that we finally settled on was to unify the concurrency model by having a pure message-passing system that follows the actor model semantics [10]. We have also seen that frameworks like Actors for Scala [11] have a similar concept. It is worth stressing that, when injecting a DES underneath real application code, the message-passing model is much more suited.

More Realistic Network Model. Other than being injected underneath the real application code, our discrete-event simulation layer is in principle very similar to every other peer-to-peer simulator out there. That said, we have two unique features. The first is the ability to simulate the behavior of NAT boxes and the second is an efficient and accurate bandwidth allocation model. For the former, we have taken all the real-life complexities we found in actual deployments and implemented that logic in a NAT box emulator integrated in our framework. Up to our knowledge, this is the first emulator of its kind. For the latter, we have surveyed how others crafted bandwidth allocation in their simulators. Packet-level simulators like NS-2 [12] are the winners in terms of accuracy but fall short on efficiency for the desired scale of thousands of peers. A more efficient solution is to use flow-based simulation where we have found that the max-min fairness approach [13] is the most-widely used model. Nevertheless, implementations of max-min fairness vary a lot, not only in accuracy but in efficiency as well. We have implemented our own max-min based bandwidth allocation model where we substantially improved on efficiency without sacrificing too much accuracy. The ability to provide a more realistic network model is important. In the absence of NAT box emulation, P2P algorithm designers would have a very naive picture of phenomena like long connection setup times, the impossibility of connectivity between different combinations of NAT boxes, and hence peers behind them. Similarly, sloppy models for bandwidth allocation result in overly optimistic estimation of data transfer delays, especially in P2P networks where a lot of transfers are taking place simultaneously.

Our P2P Development Framework. We have combined all of our best-practices in a P2P Java middleware called Mesmerizer. At the moment, all Peerialism applications are written on top of that platform. The rest of this paper is dedicated to explaining how our best practices are realized in the Mesmerizer framework. We start by describing the Mesmerizer programming model in Section 3.2 as well as its internals in Section 3.3. Then, we present the execution modes available to users in Section 3.4 and give more details about the network layer, both real and simulated, in Section 3.5. Finally, we present our conclusions and future work in Section 3.6.

3.2 The Mesmerizer framework

Applications developed in Mesmerizer consist of a set of *components*. Every component has an *interface* and one or more corresponding *implementation(s)*. An interface is bound to a component implementation with an explicit *binding*. A component instance belongs to a certain *group*. Groups contain one or multiple component instances, have explicit identifiers and define the scope of communication between components. A component may communicate with other components in a message-passing fashion using *events*. When an event is triggered by a component, Mesmerizer broadcasts it to the the component group it belongs. Events may be triggered for immediate execution, e.g. messaging, or future execution, e.g. timeouts. Inter-group communication is allowed by using explicit addressing of groups. Static tunnels between groups can be defined to automatically forward one or many types of events from one group to the other in order to avoid the need of explicit addressing. Every handler processes a single type of event. Handlers belonging to a certain component may be executed *sequentially*, i.e. a component instance processes a single handler at a time. This allows for simple concurrency semantics and isolation of components' state. However, event handlers can be defined as *concurrency-safe*. In that case, no synchronization control is applied, i.e. many "safe" handlers may be executed, while only a single "unsafe" handler is running at the same time. Explicit *filters* may be defined as a protection in order to avoid execution of handlers based on the runtime characteristics of events.

3.3 Implementation

Mesmerizer is implemented in Java as an extension of Google Guice. Guice is a lightweight injection framework. It was principally developed to alleviate the use of the explicit factory pattern in Java. It also provides a tight degree of control on how, which and when implementations of a certain interface should be instantiated. In Guice, the instantiation mechanism can be configured using explicit static *bindings*, i.e. interface to implementation mappings, and *scopes*. By default, the library makes use of a completely stateless scope, instances of a specific interface implementation are allocated every time the application requests a new instance. The library provides two stateful scopes: singleton, which causes Guice to return always the same instance of a certain interface, and a request scope, used mainly for Servlet applications, which allocates instances on a request-by-request basis. In general, Guice is principally used for unit-testing. It is common practice to swap dependencies of a Guice component with mock implementations in order to test its functionalities independently.

Mesmerizer uses Guice bindings to map component interfaces to corresponding implementations. Component interfaces are simple Java interfaces which define a number of handlers. Handlers are objects which inherit from a specific abstract Handler class and are statically typed by the event class which they take as an argument. Figure 3.1 illustrates an example of component interface and two corresponding component implementations. In this case, a binding has been created between $TimeInterface$

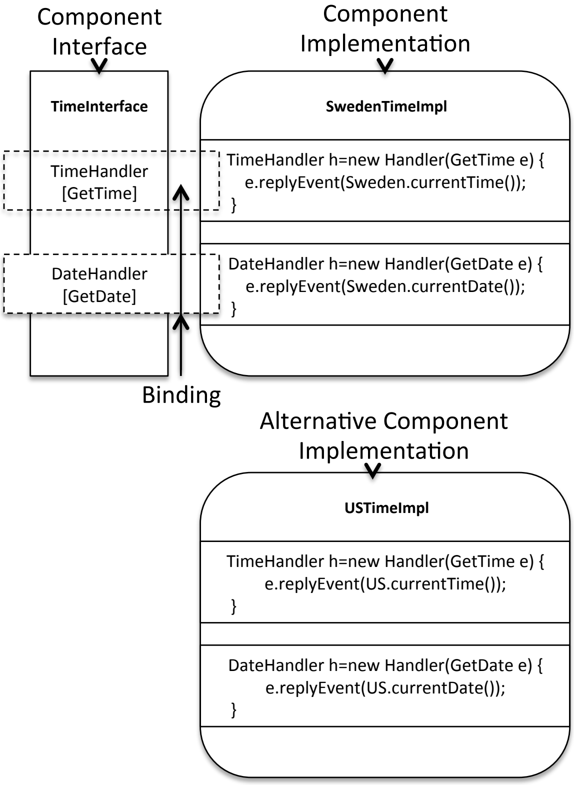

Component Interface
Component Implementation
TimeInterface
SwedenTimeImpl
TimeHandler [GetTime]
TimeHandler h=new Handler(GetTime e) {
e.replyEvent(Sweden.currentTime());
}
DateHandler [GetDate]
DateHandler h=new Handler(GetDate e) {
e.replyEvent(Sweden.currentDate());
}
Binding
Alternative Component Implementation
USTimeImpl
TimeHandler h=new Handler(GetTime e) {
e.replyEvent(US.currentTime());
}
DateHandler h=new Handler(GetDate e) {
e.replyEvent(US.currentDate());
}

Figure 3.1: Example Component Interface and implementations

and *SwedenTimeImpl*.

Component instances live in the context of groups. Upon the creation of a group, a set of the aforementioned Guice bindings must be provided together with the corresponding allocation policy. The latter defines if a component instance should be considered as a singleton or not in the context of the group or of the application. Component instantiation calls are made on group instances. Internally, Mesmerizer groups make use of Guice's scope mechanism to create and keep track of instances in their context. As a consequence of this design, two different groups may be able to bind the same interface to different implementations. When a component implementation gets instantiated, Mesmerizer uses Guice to parse both the component interface and the bound implementation. After the parsing, the framework registers each handler as a possible destination for the event it is typed with. This information is then used upon event triggering to retrieve the handlers to execute. As a design choice, we let many different components implement a handler for the same event type, whereas we allow only a single handler for a specific event type on the same component.

In Figure 3.2 we show the structure of Mesmerizer. The most important part of the

```
......
// Binding
bind(TimeInterface).to(SwedenTimeImpl).as(Scopes.GroupSingleton)
......

// Interface
trait TimeInterface extends ComponentInt {
 @Handler def timeHandler():Handler[GetTime]
 @Handler def dateHandler():Handler[GetDate]
}

// Implementation
class SwedenTimeImpl extends TimeInterface {
 def timeHandler():Handler[GetTime] = {
  return timeHandler;
 }
 def dateHandler():Handler[GetDate] = {
  return dateHandler;
 }
 val timeHandler = new Handler[GetTime]() {
   def handle(e: GetTime):Unit = {
   e.reply(new RespTime(currentTime(SWEDEN)))
 }}

 val dateHandler = new Handler[GetDate]() {
  def handle(e: GetDate):Unit =  {
     e.reply(new RespDate(currentDate(SWEDEN)))
 }}
}
```

Listing 3.1: Example Component Interface and corresponding Component Implementations in the Scala language

platform is the *Core*, which contains the implementation of the model's semantics and wraps around the Guice library. It provides mainly the group and component instantiation mechanisms, in addition to the registration and resolution mechanisms used for event routing. The actual handling of events, that is the scheduling and the execution, is carried out by the *Scheduler*, while the *Timer* is entitled with the task of handling timeouts and recurring operations. A glue *Interface and Management layer* is added to handle setup of one or multiple application instances and provide logging through the *Logger* embedded in the system. As we can see from the figure, one or many component groups can be created by an application, in this case G_1 and G_2 and filled with component instances C_1, C_2 and C_3, which are different instances of the same component implementations residing in the two different groups.

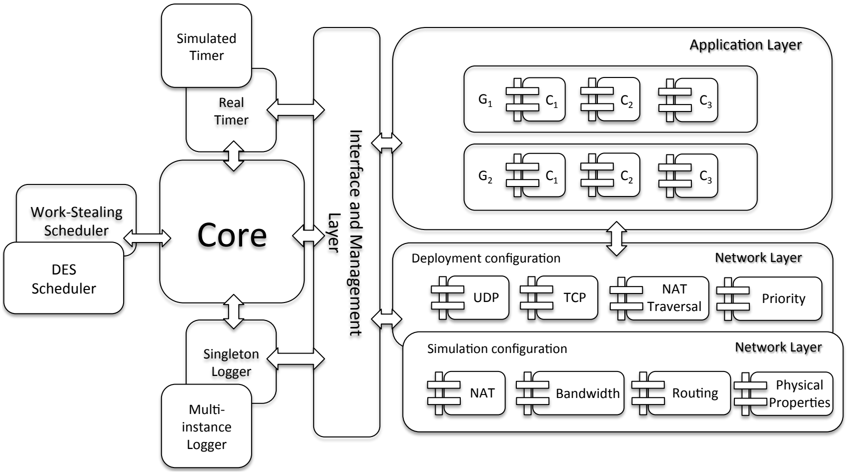


Figure 3.2: The structure of the Mesmerizer Framework

3.3.1 Scala Interface

We implemented an interface layer over Mesmerizer to be able to develop components in Scala [14]. Scala is a programming language designed to express common programming patterns in an easier and faster way. It is an object-oriented language which also supports functional programming. Our experience is that Scala allows for faster prototyping than Java. Algorithms and other complex routines can be written in much shorter time than in Java and expressed in a clearer and more compact way making it easier for the programmer/researcher to implement, understand and improve both its logic and code. An example of the Scala code corresponding to Figure 3.1's depiction of component interface, implementation and binding is shown in Listing 3.1.

We are currently working on a layer to interface the Python language with Mesmerizer by using the Jython library [15]. In principle, any programming language that compiles to Java bytecode can be interfaced with Mesmerizer. The Mesmerizer framework provides two different execution modes: *simulation* and *deployment*. The former allows for a strictly controlled execution of multiple instances of an application. It enables both large-scale reproducible experiments and smaller-scale testing/debugging of applications over an emulated network layer.

Deployment allows for parallel processing of events. In this mode, one or multiple application instances are executed using a concurrent environment while network services are provided by a library which enables TCP and UDP communication between hosts.

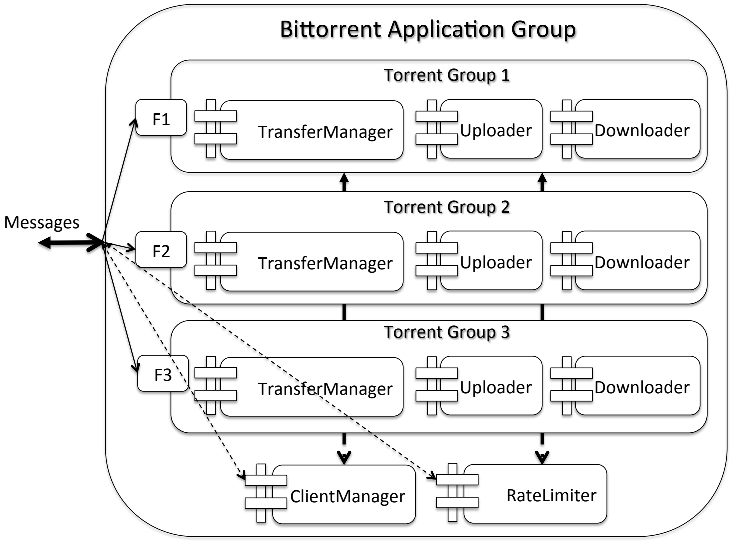


Figure 3.3: The structure of our Bittorrent Application

3.4 Execution modes

The design of Mesmerizer allows an application to be run either in simulation or in emulation mode by simply changing some of the Mesmerizer's system bindings, namely the Scheduler, Timer, Logger and Network layer components as shown in Figure 3.2.

3.4.1 Simulation Mode

In simulation mode, multiple instances of the same application are spawned automatically by Mesmerizer according to a specified configuration provided to the Management Layer. Event execution in this case is controlled by a Scheduler based on a single-threaded Discrete Event Simulator (DES). During execution, triggered events are placed into a FIFO queue and the corresponding handlers executed in a sequential manner. Our previous experience in using a multi-threaded DES based on pessimistic lock-stepping [16], where events of the current time step are executed in a concurrent fashion, has shown that the amount of overhead required for this technique to work is larger than the actual benefits. We found the event pattern to be very sparse for the applications we developed using Mesmerizer: a live streaming platform and a Bittorrent client. We noticed that the synchronization burden required to achieve barrier synchronization between consecutive time intervals is far greater than the speed-up obtained by the actual processing of the few events present in the "concurrency-safe" intervals.

Trying to scale simulation, we mostly concentrated our efforts in improving the performance of what we experienced to be the biggest bottleneck in our P2P simulations:

emulating bandwidth allocation dynamics of the network. We will show in Section 3.5.2 how the Mesmerizer simulation environment performs in terms of scalability when emulating such phenomena and its level of accuracy with respect to other more costly solutions.

Simulation Setup. We provide a set of APIs which enable users to fully configure the execution of multiple application instances and carefully control the behavior of the simulated network layer. Typically in simulation mode, Mesmerizer isolates an application instance in its own component group containing all of its components. Application instances communicate using the same network interface provided in deployment mode. Messages are however routed to an emulated network layer which models a number of characteristics found in real networks.

For large-scale experiments, we implemented a number of churn generators based on probabilistic and fixed time behaviors which can be configured either programmatically or through an XML scenario file. The emulated underlying network can also be configured in the same way. The Mesmerizer simulator allows for the creation of scenarios containing a predefined number of peers interconnected using routers and NAT boxes [17] on simple end-to-end topologies or more complicated consumer LAN infrastructures and/or complex multi-layered corporate networks. The simulation APIs make possible to define bandwidth capacities for each peer/router in the network, dictate the behavior of NAT boxes and configure in detail network characteristics such as delay patterns, packet loss and link failures.

3.4.2 Deployment Mode

Deployment allows for the execution of application instances in a concurrent environment. In this mode, the handlers of the application's components are processed on multiple threads concurrently. From a component's instance point of view, a number of its *safe* handlers can run together at once, however, its *unsafe* handlers will be executed sequentially. On the other hand, many unsafe handlers of different components may be active at the same point in time. In this mode, execution is controlled by a concurrent Scheduler which implements the work-stealing paradigm introduced by Blumofe et al [18] on a number of Java threads, similarly to Kompics [7]. Work-stealing is an efficient scheduling scheme for multi-core machines which is based on an queuing mechanism with low cost of synchronization. We make use of the work-stealing paradigm in the following way: when an event gets scheduled, Mesmerizer finds the component instances which subscribed to that particular event type and hands them to the Scheduler. The Scheduler then checks the availability of a number of free threads corresponding to that of the passed handlers. If enough free threads are found, it schedules the handlers for immediate execution. If no sufficient number of threads is available, the scheduler distributes randomly the remaining handlers to the waiting queues of the currently occupied threads. When a thread completes the execution of a handler, it tries either to take an event from its queue or, if its queue is empty, it steals handlers from another thread's queues. In our experience, this type of scheduling guarantees a good level of fairness and avoids starvation of handlers.

In Figure 3.3 we show the structure of a Bittorrent client that we developed using Mesmerizer, which is currently deployed in our test network. The application is made by two basic components which reside into the main *Bittorrent Application* component group: the *Client Manager* and the *Rate Limiter*. The former is entitled with the task of setting up and remove Bittorrent transfers, while the latter controls load balancing and priority levels between transfers. When a new torrent file is provided to the application, the $ClientManager$ creates a component group for the new transfer, for instance $TorrentGroup1$. The group contains all transfer's components, which are instances of the interfaces $Downloader$, $Uploader$ and $TransferManager$. Which implementation should be used for the aforementioned interfaces is defined in a Guice binding when adding the torrent. We designed a number of different implementations of the transfer components which provide various transfer strategies, such as partial in-order or random. This kind of design allow for the use of multiple transfer policies in the same client by simply providing the right binding when adding new transfers. During transfer setup, the $ClientManager$ also proceeds to create automatic tunneling of message events from the network layer to $TorrentGroup1$ using filters based on message characteristics. We use the mechanisms of tunneling and filtering for automatic inter-group routing and multiplexing of incoming messages respectively. Routing of events is also carried out internally to the application between $Uploader/Downloader$ component instances and both the $ClientManager$ and the $RateLimiter$. The latter in particular has the important task of keeping the view of all transfer rates and dynamically adjust, by issuing the correct events, the uploading/downloading rates of the $Uploader$ and $Donwloader$ components, when they excess the priority level or max speed.

3.5 Network Layer

We provide two different sets of components to be used as network layer: one for deployment and another for simulation mode. In deployment mode, components need to transfer messages, i.e. remote events, to other remote hosts using the TCP or UDP protocol. In simulation mode instead, the network layer is entitled with the task of modeling those same transfers between a number of application instances which are running in the same context, i.e. the same instance of the Mesmerizer framework. We detail the composition of the network layer in both modes in the following sections.

3.5.1 Deployment Configuration

Our network layer deployment includes a number of components that offer TCP and reliable UDP communication to the overlying applications through a simple event-based interface. Our TCP and UDP components deliver out-of-the-box support for transparent peer authentication and encrypted data exchange. We have implemented two components to achieve NAT traversal and improve peer-to-peer connectivity; support for explicit NAT traversal protocols such as UPnP and NAT-PMP, and state-of-the-art techniques for UDP hole-punching, based on our previous work [19]. If direct connectivity between two peers cannot be achieved, the network layer automatically relays the communication over a third host. However, we use this feature only for signaling purpose since relaying traffic is a very expensive operation, in particular for data intensive applications such as video streaming or content delivery platforms where the amount of data to be transfered is significant.

On top of NAT traversal and reliable communication, the deployment network layer provides three strategies for traffic prioritization based on different UDP congestion control methods. For low priority traffic, we implemented the LEDBAT delay-based congestion control [20], which yields with respect to other concurrent TCP streams. For what we call fair level of priority, or medium, we adopted a variation of the TCPReno [21] protocol which enables equal sharing of bandwidth among flows generated by our library and other applications using TCP. Finally, when the delivery of data is of critical importance, the library provides high priority through an adaptation of the MulTCP [22] protocol. The level of priority can be dynamically chosen on a flow-to-flow basis at runtime.

3.5.2 Simulation Configuration

In Simulation mode, we make use of a number of components which emulate different aspects of the network. For instance, on the IP layer, we provide routing and NAT emulation through the corresponding components. For modeling lower layer network characteristics, we implemented components that model bandwidth allocation dynamics, non-deterministic delays and packet loss. As mentioned in Section 3.1, some of these emulated characteristics are found in almost all P2P simulators. We detail the NAT and bandwidth emulation components; the first being notable for its novelty and the second for its high level of efficiency/accuracy trade-off.

NAT Emulation

Network Address Translators constitute a barrier for peer-to-peer applications. NAT boxes prevent direct communication between peers behind different NAT boxes [17]. Even though an attempt has been made to standardize Network Address Translation [23], in particular to improve support for Peer-To-Peer applications, not all vendors have complied to the standard. This is either because most of the routers ship with legacy NAT implementations or because manufacturers claim the standard to be too permissive. In particular, in the context of corporate NAT implementations, the translation behavior may vary drastically between different vendors or even different models of the same manufacturer.

In our previous work, we have tried to classify most of the behaviors of current NAT implementations using a real deployment of our test software. The outcome of this effort is a model that encompasses 27 types of NATs as a combination of *three* behavioral policies: filtering, allocation and mapping. NAT traversal is a pairwise connection establishment process: in order to establish a communication channel between machines behind two different NAT boxes, it is necessary to carry out a connection setup process dictated by a NAT traversal strategy, which should vary according to the type of the two considered NAT boxes. Given a set of *seven* known traversal strategies, the NAT problem translates to understanding which strategy should be used in each one of the 378 possible combinations of the 27 NAT types. On top of that, each traversal strategy has its own configuration parameters. These parameters could be as detailed as deciding which source port should be used locally by the two source hosts to initiate the setup process in order to maximize the connection establishment success probability.

At first, we tried to understand the mapping between NAT type combinations and traversal strategies by formal reasoning. However, this task turned out to be too complex due to the size of the possibility space. As a consequence of that, we developed a configurable NAT box implementation which could emulate all the aforementioned NAT types. On top of it, we implemented a small simulator which would perform the process of traversal, according to the *seven* strategies, on all of the emulated 240 NAT type combinations. By using this small framework, we discovered many counter-intuitive aspects of NAT traversal and mistakes in the strategy's decision process which we would not have discovered by simple reasoning. The outcome of our effort of mapping strategies to combinations is described in [19].

For the purpose of testing the correct and non-trivial implementation of the Traversal strategies in our application, we included NAT emulation in the network layer configuration of Mesmerizer. In other words, we built an emulated network where the connection establishment process by means of NAT traversal is modeled in a very detailed manner. Thus, we were able to test not only the correctness of our implementation of the traversal strategies, but also to measure the impact of the connection establishment delays on the overlying application. Delays are very important factors to be considered when designing audio and video streaming systems, due to the time-constrained nature of the application.

The model which we designed for the emulation of NAT boxes encompasses most of the behaviors which are found in current routers and corporate firewalls. However, after the deployment of our live streaming application on a real network, we noticed a number of exceptional characteristics, e.g. non-deterministic NAT mapping timeouts, inconsistent port allocation behaviors and the presence of multiple filtering policies on the same router. We thus tried to formalize those exceptions and integrate them into our simulator. We further improved our simulator by modeling connection establishment failures based on real-world measurements. We emulate the observed probability of success between NAT types combinations according to what we observed during our real-world tests. Currently, we still experience cases that are not covered by our emulation model and we keep improving our model based on those observations. The source code of our NAT box emulator is publicly available as an open source project [24].

The detailed emulation of NAT boxes has provided us with better insight on how peer-to-peer applications should be designed and implemented in order to avoid connectivity issues.

Bandwidth Modeling

It is common for P2P networks to create complex interactions between thousands of participant peers, where each peer typically has very high inbound and outbound connection degree [25] [26] [27]. Connections are used either for signaling or for content propagation. In the latter case, each peer implements intricate multiplexing strategies to speed up transmission of large chunks of data [28], thus creating complex effects on the underlying physical network which translate into varying transmission delays and packet loss at the receiving side.

Most of the existing P2P simulators abstract away network interactions by modeling only the structural dynamics of the overlay network [29] [30] [31] [32] [33] and thus totally ignoring the impact of the actual network on application performance. On the other end, accurate packet-level simulators like SSFNet [34] and NS-2 [12] can usually scale up only to a limited number of simulated peers. This limitation makes it infeasible to capture the behavior and issues of a larger P2P real-word deployment, such as the effect of network congestion on segments of the overlay network. In order to study the complex interactions between large scale overlays and the physical network, a proper network simulation model is required. The level on which the model abstracts the network transfers directly affects both its scalability and accuracy.

Flow-level network simulation focuses on a transfer as a whole rather than individual packets, introducing a viable trade-off between accuracy and scalability. A flow abstracts away the small time scale rate variation of a packet sequence with a constant rate allocated at the sender/receiver's bandwidth. The rate remains allocated for an amount of time which corresponds to the duration of the flow, i.e. the simulated packet sequence transmission time. This approach reduces drastically the number of events to be simulated. The driving force behind the event creation in flow-level simulation is the interaction between the flows since an upload/download link might have many flows happening at the same time. A new or completed flow might cause a rate change on other flows competing for that same link's capacity. A flow rate change may also propagate further in the simulated network graph. This phenomenon is known as *"the ripple effect"* and has been observed in a number of studies [35] [36]. The impact of the ripple effect on the scalability of the model is directly dependent on the efficiency of the bandwidth allocation algorithm which is used to mimic the bandwidth dynamics.

Bertsekas and Gallager [13] introduce the concept of *max-min fairness* for modeling Additive-Increase Multiplicative-Decrease congestion control protocols like TCP. Max-min fairness tries to maximize the bandwidth allocated to the flows within a minimum share thus guaranteeing that no flow can increase its rate at the cost of a flow with a lower rate. In every network exists a unique max-min fair bandwidth allocation and can be calculated using the progressive filling algorithm [13]. The basic idea behind this algorithm is to start from a situation where all flow rates are zero and then progressively increment each rate equally until reaching the link's capacity, i.e. the sum of all flow rates of a link equals its capacity. In this algorithm, the network, including its internal structure, e.g. routers and backbone links, is modeled as an undirected graph. A recent accuracy study [37] showed that this approach offers a good approximation of the actual network behavior. Nevertheless, having to simulate the flow interactions that take place on the internal network links magnifies the impact of the ripple effect on the algorithm's scalability by making the simulation significantly slower.

In order to gain more scalability, the *GPS* P2P simulator [38] uses a technique called *minimum-share allocation*, defined in [39], which avoids the propagation of rate changes through the network. Instead, only the flow rates of the directly affected nodes are updated, i.e. only the flow rates of the uploading and downloading nodes of the flow triggering the reallocation. Not considering the cross-traffic effects of the flows obviously has a positive impact on the simulation time but also makes the model highly inaccurate. *Narses* [39] uses the same technique as GPS but it further promotes scalability by ignoring the internal network topology and considers only the bandwidth capacity of the access links of the participating peers. The result is what we call an *end-to-end* network overlay where the backbone network is completely abstracted away from the modeling and rate changes happen between pairs of peers. This is a reasonable abstraction if we consider that the bottlenecks on a P2P network usually appear in the "last mile" rather than the internet backbone. In doing so, the number of events simulated is further reduced, however in this case the inaccuracy remains since only the end-to-end effects are taken into account while the cascading effect on other nodes, as modeled by max-min fair allocation, is completely overlooked.

There exists two bandwidth allocation algorithms in the state of the art which apply the progressive filling idea on end-to-end network models, thus keeping the advantages of simulating only access links but still considering the effects and propagation of rate changes throughout the peer interconnections. The first algorithm proposed by F. Lo Piccolo et Al. [40] models the end-to-end network as an undirected graph. In each iteration, the algorithm finds the *bottleneck* nodes in the network, the nodes with the minimum fair bandwidth share available to their flows. Then it proceeds to allocate the calculated minimum fair share to their flows. The algorithm iterates until all nodes are found saturated or a rate is assigned to all their flows. The main disadvantage of this node-based max-min fair bandwidth allocation algorithm lies in the modeling of the network as an undirected graph. In order to simulate a network with separate upload and download capacities, two node instances are required per actual network peer. The memory footprint is therefore larger than the one needed to model a direct network graph.

An alternative edge-based max-min bandwidth allocation algorithm is given by Anh Tuan Nguyen et al. [41]. It is an edge-based algorithm which uses a directed network model, differently from the approaches we introduced until now. In one iteration, the algorithm calculates the minimum fair share of the two ends of every unassigned flow. Then, on the same iteration and based on the previously calculated shares, the algorithm finds the bottleneck nodes, derives the flows' rates and applies them. The algorithm iterates until all flows have a rate assigned. It is important to underline that during the second phase of each iteration, the algorithm might find one or multiple bottleneck nodes, thus assigning rates to the flows of multiple nodes at the same iteration. This edge-based max-min fair bandwidth allocation algorithm addresses the shortcoming of the undirected network modeling, that is the memory footprint. However, the algo-

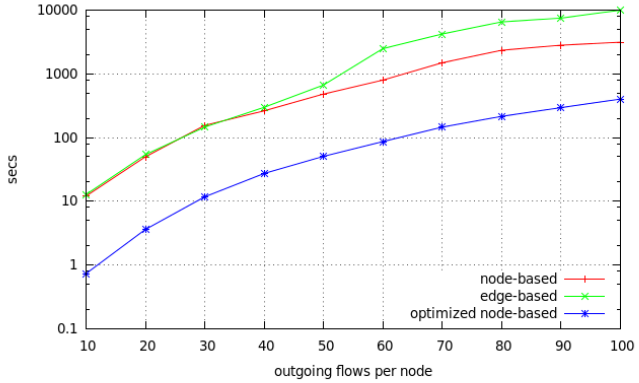


Figure 3.4: Performance comparison for structured network overlays with 1000 nodes and different number of outgoing flows per node.

rithm performance's dependence on the edge-set size constitutes a major drawback. On top of that, a further iteration of the node set is required in order to find the saturated nodes.

It is common in large simulated networks for a new or finished flow to only affect the rates of a subset of the existing network flows, as the propagation of a rate change does not reach all nodes in the network but rather few of them. Based on this observation, F. Lo Piccolo et Al. [40] partially outline an affected subgraph discovery algorithm that can be applied on an undirected network graph.

Using this optimization algorithm before applying an undirected node-based max-min fair bandwidth allocation algorithm leads to a large performance gain. Unfortunately, F. Lo Piccolo et Al. apply this only on an undirected network model. Moreover, the authors provide only a sketch of the affected subgraph idea rather than a state-complete algorithm. In our simulator we leverage the benefits of the affected subgraph optimization on a directed network model. The result is an algorithm whose computational complexity is independent of the edge set size. Our node-based max-min fair bandwidth allocation algorithm iterates until all flows have a rate assigned. Each iteration has two phases. In the first, we find the node(s) which provide the minimum fair share by calculating the fair share of the upload and the download capacity of each node. The lower of these rates is set as the minimum fair share and the corresponding node sides (uploading or downloading) are considered saturated i.e. they constitute the bottleneck of the network in this iteration. In the second phase, we allocate this minimum fair share to the flows of each saturated node, downloading or uploading depending on their saturated side. In order to improve scalability, we adapt the affected subgraph discovery algorithm for use with directed end-to-end network models. Given a flow that triggers a bandwidth reallocation, we initiate two graph traversals that each one has as root one of the flow's end nodes. In each hop of a traversal we find the affected flows of the last reached nodes and continue the traverse to their other ends.

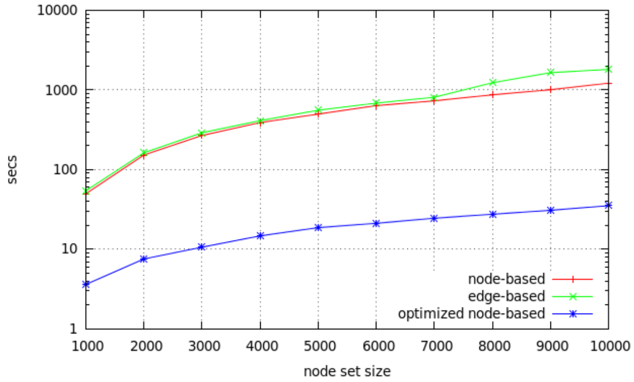


Figure 3.5: Performance comparison for structured network overlays with varying size and 20 outgoing flows per node.

This procedure continues until no newly affected nodes are discovered by any of the two traversals.

Evaluation For our scalability evaluation, we consider structured overlay scenarios with two main parameters: the size of the node set and the number of outgoing flows per node. The nodes enter the system in groups at defined time intervals and the starting times their flows are distributed uniformly in that same time interval. The destination of the flows is chosen following a specific structure, i.e. a DHT-based one. The bandwidth capacities of the nodes are chosen randomly from the set: {100Mbps/100Mbps,24Mbps /10Mbps,10Mbps/10Mbps,4Mbps/2Mbps,2Mbps/500Kbps} with corresponding probabilities of {20%,40%,10%,10%}.
Our experiments show, Figures 3.4-3.5, that our node-based max-min fair allocation algorithm constantly outperforms the edge-based algorithm proposed by Anh Tuan Nguyen et al. for large-scale and structured network overlays. An important conclusion drawn from these results is that the number of connections per node has a bigger impact in the performance of the simulation models rather than the node set size. For example, the simulation of a random overlay of 1000 nodes with 70 outgoing flows requires a similar simulation time to a 10000 nodes random overlay having 20 outgoing flows per node. We also would like to point out that the performance gain when using flow-level simulation instead of packet-level is paramount. In order to simulate a random scenario of 1000 nodes with 10 flows each, a time of three orders of magnitude longer is required. Running the same scenarios using the optimization algorithm significantly reduces the simulation time. In Figure 3.4 we can see that the required time is one order of magnitude lower when simulating network overlays with the same size but different number of outgoing flows per node. The performance improvement is much higher, three orders of magnitude, when we increase the network size and keep the number of flows per node fixed, as shown in Figure 3.5.

c^u/c^d	flows	std. deviation	avg. deviation
20/10	10	3.8±0.4%	3±0.4%
	20	3.9±0.2%	3.1±0.1%
	30	4.1±0.3%	3.3±0.2%
	40	3.4±0.2%	2.8±0.2%
	50	3±0.1%	2.5±0.2%
20/10,10/10	10	6.4±0.4%	5±0.4%
	20	6±0.4%	4.9±0.3%
	30	4.8±0.4%	3.9±0.3%
	40	3.4±0.9%	3.2±0.3%
	50	3.5±0.2%	2.8±0.2%

Table 3.1: Deviation of simulated transfer times.

Finally, in our accuracy study we compare the simulated transfer times of our proposed solution with the ones obtained with NS-2 for the same scenarios. In NS-2, we use a simple star topology, similar to the one used in [37] [42]. Each node has a single access link which we configure with corresponding upload and download capacities. All flows pass from the source access link to the destination access link through a central node with infinite bandwidth capacity. Unfortunately, the size of our experiments is limited by the low scalability of NS-2. We run scenarios of 100 nodes with a number of flows per node that varies between 10 and 50. The size of each flow is 4MB and the bandwidth capacities of a node are either asymmetric, 20Mbps/10Mbps, or mixed, 20Mbps/10Mbps and 10Mbps/10Mbps. The results of our experiments are shown in Table 3.1. We can see that our flow-level max-min fair bandwidth allocation follows the trends of the actual packet-level simulated bandwidth dynamics by a nearly constant factor throughout the experiments. We can see that the presence of the symmetric capacities affects the transfer time deviation negatively. The negative impact is more visible when fewer outgoing flows per node are used. When the links are less congested, the slower convergence of the flow rates of the nodes with smaller symmetric capacities is more apparent.

3.6 Conclusion & Future Work

In this paper we presented what we found to be the three most important practices in P2P software development: a simulation-based development cycle, a clear concurrency model, and a realistic network model when running in simulation mode. We then presented the Mesmerizer framework which we built to encompass all the aforementioned. We detailed its design and implementation and how it can be used both for controlled evaluation/testing/debugging and deployment of production code. Regarding simulation-based development, we stress how NAT box and bandwidth dynamics emulation are of vital importance when testing a large number of a P2P application instances. From the point of view of the scalability, we demonstrate that our simulation framework is able to emulate the bandwidth characteristics of thousands of peers while preserving a good level of accuracy compared to the NS2 packet-level simulator.

Our ongoing work includes the improvement of the NAT and bandwidth emulation model's accuracy and the release of all our utilities as open source software.

3.7 References

[1] Ruby on rails. http://rubyonrails.org/.

[2] django. http://www.djangoproject.com/.

[3] Jesse James Garrett. Ajax: A new approach to web applications. http://adaptivepath.com/ideas/essays/archives/000385.php, February 2005. URL `http://adaptivepath.com/ideas/essays/archives/000385.php`. [Online; Stand 18.03.2008].

[4] Comet. http://cometdaily.com/.

[5] Roberto Roverso, Mohammed Al-Aggan, Amgad Naiem, Andreas Dahlstrom, Sameh El-Ansary, Mohammed El-Beltagy, and Seif Haridi. Myp2pworld: Highly reproducible application-level emulation of p2p systems. In *Decentralized Self Management for Grid, P2P, User Communities workshop, SASO 2008*, 2008.

[6] Google guice. http://code.google.com/p/google-guice.

[7] C. Arad, J. Dowling, and S. Haridi. Building and evaluating p2p systems using the kompics component framework. In *Peer-to-Peer Computing, 2009. P2P '09. IEEE Ninth International Conference on*, pages 93 –94, sept. 2009.

[8] Galuba W. et al. Protopeer: A p2p toolkit bridging the gap between simulation and live deployement. *2nd International ICST Conference on Simulation Tools and Techniques*, May 2009.

[9] Ron Hitchens. *Java Nio*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002. ISBN 0596002882.

[10] Gul Abdulnabi Agha. Actors: a model of concurrent computation in distributed systems. MIT Press, 1985.

[11] Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, 410(2-3):202–220, February 2009. ISSN 0304-3975. URL `http://dx.doi.org/10.1016/j.tcs.2008.09.019`.

[12] The ns-2 network simulator. http://www.isi.edu/nsnam/ns/, October 2010.

[13] Dimitri Bertsekas and Robert Gallager. *Data Networks*. Prentice Hall, second edition, 1992.

[14] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Incorporation, USA, 1st edition, 2008. ISBN 0981531601, 9780981531601.

[15] The jython project. http://www.jython.org/.

[16] Shiding Lin, Aimin Pan, Rui Guo, and Zheng Zhang. Simulating large-scale p2p systems with the wids toolkit. In *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 2005. 13th IEEE International Symposium on*, pages 415 – 424, sept. 2005.

[17] P. Srisuresh, B. Ford, and D. Kegel. State of Peer-to-Peer (P2P) Communication across Network Address Translators (NATs). RFC 5128 (Informational), March 2008. URL `http://www.ietf.org/rfc/rfc5128.txt`.

[18] R.D. Blumofe and C.E. Leiserson. Scheduling multithreaded computations by work stealing. In *Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on*, pages 356 –368, November 1994.

[19] Roberto Roverso, Sameh El-Ansary, and Seif Haridi. Natcracker: Nat combinations matter. In *Proceedings of the 2009 Proceedings of 18th International Conference on Computer Communications and Networks*, ICCCN '09, pages 1–7, Washington, DC, USA, 2009. IEEE Computer Society. ISBN Bad - remove. URL `http://dx.doi.org/10.1109/ICCCN.2009.5235278`.

[20] S Shalunov. Low extra delay background transport (ledbat) [online]. http://tools.ietf.org/html/draft-ietf-ledbat-congestion-02.

[21] J. Mo, R.J. La, V. Anantharam, and J. Walrand. Analysis and comparison of TCP Reno and Vegas. *IEEE INFOCOM '99*, 1999. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=752178`.

[22] J Crowcroft and P Oechslin. Differentiated end-to-end Internet services using a weighted proportional fair sharing TCP. *ACM SIGCOMM Computer*, 1998. URL `http://portal.acm.org/citation.cfm?id=293927.293930`.

[23] F. Audet and C. Jennings. Network Address Translation (NAT) Behavioral Requirements for Unicast UDP. RFC 4787 (Best Current Practice), January 2007. URL `http://www.ietf.org/rfc/rfc4787.txt`.

[24] Roberto Roverso. NATCracker Box Emulator Software. http://code.google.com/p/natcracker/.

[25] Anwar Al Hamra, Arnaud Legout, and Chadi Barakat. Understanding the properties of the bittorrent overlay. Technical report, INRIA, 2007.

[26] Chuan Wu, Baochun Li, and Shuqiao Zhao. Magellan: Charting large-scale peer-to-peer live streaming topologies. In *ICDCS '07: Proceedings of the 27th International Conference on Distributed Computing Systems*, page 62, Washington, DC, USA, 2007. IEEE Computer Society.

[27] Guillaume Urvoy-Keller and Pietro Michiardi. Impact of inner parameters and overlay structure on the performance of bittorrent. In *INFOCOM '06: Proceedings of the 25th Conference on Computer Communications*, 2006.

[28] B. Cohen. Incentives Build Robustness in BitTorrent. In *Econ '04: Proceedings of the Workshop on Economics of Peer-to-Peer Systems*, Berkley, CA, USA, June 2003.

[29] Ingmar Baumgart, Bernhard Heep, and Stephan Krause. Oversim: A flexible overlay network simulation framework. In *GI '07: Proceedings of 10th IEEE Global Internet Symposium*, pages 79–84, Anchorage, AL, USA, May 2007.

[30] Sam Joseph. An extendible open source p2p simulator. *P2P Journal*, 0:1–15, 2003.

[31] Jordi Pujol-Ahullo, Pedro Garcia-Lopez, Marc Sanchez-Artigas, and Marcel Arrufat-Arias. An extensible simulation tool for overlay networks and services. In *SAC '09: Proceedings of the 24th ACM Symposium on Applied Computing*, pages 2072–2076, New York, NY, USA, March 2009. ACM.

[32] Alberto Montresor Mark Jelasity and Ozalp Babaoglu. A modular paradigm for building self-organizing peer-to-peer applications. In *Engineering Self-Organising Systems*, pages 265–282, 2003.

[33] Nyik San Ting and Ralph Deters. 3ls - a peer-to-peer network simulator. In *P2P '03: Proceedings of the 3rd International Conference on Peer-to-Peer Computing*, page 212. IEEE Computer Society, August 2003.

[34] Sunghyun Yoon and Young Boo Kim. A design of network simulation environment using ssfnet. pages 73–78, 2009. URL `http://portal.acm.org/citation.cfm?id=1637862.1638154`.

[35] G. Kesidis, A. Singh, D. Cheung, and W.W. Kwok. Feasibility of fluid event-driven simulation for atm networks. In *GLOBECOM '96: Proceedings of the Global Communications Conference*, volume 3, pages 2013 –2017, November 1996.

[36] Daniel R. Figueiredo, Benyuan Liu, Yang Guo, James F. Kurose, and Donald F. Towsley. On the efficiency of fluid simulation of networks. *Computer Networks*, 50(12):1974–1994, 2006.

[37] Abdulhalim Dandoush and Alain Jean-Marie. Flow-level modeling of parallel download in distributed systems. In *CTRQ '10: Third International Conference on Communication Theory, Reliability, and Quality of Service*, pages 92 –97, june 2010.

[38] W. Yang and N Abu-Ghazaleh. Gps: a general peer-to-peer simulator and its use for modeling bittorrent. In *MASCOTS '05: Proceedings of 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 425–432, Atlanta, Georgia, USA, September 2005.

[39] Thomas J. Giuli and Mary Baker. Narses: A scalable flow-based network simulator. *Computing Research Repository*, cs.PF/0211024, 2002.

[40] F. Lo Piccolo, G. Bianchi, and S. Cassella. Efficient simulation of bandwidth allocation dynamics in p2p networks. In *GLOBECOM '06: Proceedings of the 49th Global Telecommunications Conference*, pages 1–6, San Franscisco, California, November 2006.

[41] Anh Tuan Nguyen and F. Eliassen. An efficient solution for max-min fair rate allocation in p2p simulation. In *ICUMT '09: Proceedings of the International Conference on Ultra Modern Telecommunications Workshops*, pages 1 –5, St. Petersburg, Russia, October 2009.

[42] Tobias Hossfeld, Andreas Binzenhofer, Daniel Schlosser, Kolja Eger, Jens Oberender, Ivan Dedinski, and Gerald Kunzmann. Towards efficient simulation of large scale p2p networks. Technical Report 371, University of Wurzburg, Institute of Computer Science, Am Hubland, 97074 Wurzburg, Germany, October 2005.

CHAPTER 4

ACCURATE AND EFFICIENT SIMULATION OF BANDWIDTH DYNAMICS FOR PEER-TO-PEER OVERLAY NETWORKS

Accurate and Efficient Simulation of Bandwidth Dynamics for Peer-To-Peer Overlay Networks

Alexandros Gkogkas[1,2], Roberto Roverso[1,2] and Seif Haridi[2]

[1]Peerialism AB
Stockholm, Sweden
[roberto,sameh]@peerialism.com

[2]KTH - Royal Institute of Technology
Stockholm, Sweden
[haridi]@kth.se

Abstract

When evaluating Peer-to-Peer content distribution systems by means of simulation, it is of vital importance to correctly mimic the bandwidth dynamics behavior of the underlying network. In this paper, we propose a scalable and accurate flow-level network simulation model based on an evolution of the classical progressive filling algorithm which implements the max-min fairness idea. We build on top of the current state of art by applying an optimization to reduce the cost of each bandwidth allocation/deallocation operation on a node-based directed network model. Unlike other works, our evaluation of the chosen approach focuses both on efficiency and on accuracy. Our experiments show that, in terms of scalability, our bandwidth allocation algorithm outperforms existing directed models when simulating large-scale structured overlay networks. Whereas, in terms of accuracy we show that allocation dynamics of the proposed solution follow those of the NS-2 packet-level simulator by a small and nearly constant offset for the same scenarios. To the best of our knowledge, this is the first time that an accuracy study has been conducted on an improvement of the classical progressive filling algorithm.

4.1 Introduction

It is common for existing P2P networks to create complex interactions between thousands of participant peers. Each peer usually has a very high inbound and outbound connection degree [1] [2] [3]. Connections are used either for signaling or content propagation. In the latter case, each peer implements intricate multiplexing strategies to speed up transmission of large chunks of data [4], thus creating complex effects on the underlying physical network which translate into varying transmission delays and packet loss at the receiving side.

In order to study the complex interactions between overlays and physical networks, a proper performance evaluation technique is required. Currently, no consensus has been reached in the scientific community on a reference P2P testing platform [5]. It is of a common practice to test a P2P application in a simulator before real deployment in order to enable controlled testing and evaluation. However, accurate network simulators can usually scale up only to a limited number of simulated peers. This limitation makes it infeasible to capture the behaviour and issues of a larger P2P real-word deployment, such as the effect network congestion on segments of the overlay network. In order to achieve scalability, most of the existing P2P simulators abstract away network interactions by modeling only the structural dynamics of the overlay network [6] [7] [8] [9] [10] and thus totally ignoring the impact of the actual network on application performance.

Packet-level network simulation allows for in-depth study of the influence of the network characteristics and the lower level protocols on the performance of a P2P system. The trade-off for this detailed analysis comes in scalability: a single transfer can generate hundreds of low-level protocol packets. In a large and complex network characterised by links of high bandwidth capacity and intense in-line traffic, the sheer number of events needed to simulate the transfers require a prohibitive amount of computational and memory resources. As a result, only small sized networks can be simulated efficiently. A number of packet-level network simulation frameworks has been conceived in the last decade [11] [12], being NS-2 [13] the most prominent among them. In turn, a number of P2P network simulators have been developed on top of NS-2, e.g. P2PSim [14] and NDP2PSim [15].

Flow-level simulation focuses instead on a transfer as a whole rather than individual packets, introducing a viable trade-off between accuracy and scalability. A flow abstracts away the small time scale rate variation of a packet sequence with a constant rate allocated at the sender/receiver's bandwidth. The rate remains allocated for an amount of time which corresponds to the duration of the flow, i.e. the simulated packet sequence transmission time. This approach reduces drastically the number of events to be simulated. The driving force behind the event creation in flow-level simulation is the interaction between the flows, since an upload/download link might have many flows happening at the same time. A new or completed flow might cause a rate change on other flows competing for that same link's capacity. For instance, in order for a link to accommodate an extra flow, bandwidth needs to be allocated to it. This might lead to a decrease of the bandwidth rates of other competing flows. Similarly, when a flow is completed, more bandwidth becomes available for the other flows to share. A flow rate change may also propagate further in the simulated network graph. This phenomenon is known as *"the ripple effect"* and has been observed in a number of studies [16] [17] . The impact of the ripple effect on the scalability of the model is directly dependent on the efficiency of the bandwidth allocation algorithm.

In this paper we provide a description of the current state of the art in flow-level network bandwidth simulation in Section 4.2. Later we present a concise description of our contribution in Section 4.3, before delving into the details of the proposed model in Section 4. We then discuss the results of the scalability and accuracy evaluation of our implementation in Section 5 and, finally, in Section 6, we draw our conclusions and discuss future work.

4.2 Related Work

Flow-level network simulators implement algorithms which mimic the bandwidth dynamics happening in the transport protocol layer used of the real network. Bertsekas and Gallager [18] introduce the concept of *max-min fairness* for modeling Additive-Increase Multiplicative-Decrease congestion control protocols like TCP. Max-min fairness tries to maximize the bandwidth allocated to the flows with minimum share thus guaranteeing that no flow can increase its rate at the cost of a flow with a lower rate. In every network exists a unique max-min fair bandwidth allocation and can be calculated using the progressive filling algorithm [18]. The basic idea behind this algorithm is to start from a situation where all flow rates are zero and then progressively increment each rate equally until reaching the link's capacity, i.e. the sum of all flow rates of a link equals its capacity. In this algorithm, the network, including its internal structure, e.g. routers and backbone links, is modeled as an undirected graph. A recent accuracy study [19] showed that this approach offers a good approximation of the actual network behavior. Nevertheless, having to simulate the flow interactions that take place on the internal network links, magnifies the impact of the ripple effect on the algorithm's scalability by making the simulation significantly slower.

In order to gain more scalability, the *GPS* P2P simulator [20] uses a technique called *minimum-share allocation*, defined in [21], which avoids the propagation of rate changes through the network. Instead, only the flow rates of the directly affected nodes are updated, i.e. only the flow rates of the uploading and downloading nodes of the flow triggering the reallocation. Not considering the cross-traffic effects of the flows obviously has a positive impact on the simulation time but makes also the model highly inaccurate.

Narses [21] uses the same technique as GPS but it further promotes scalability by ignoring the internal network topology and considering only the bandwidth capacity of the access links of the participating peers. The result is what we call an *end-to-end* network overlay where the backbone network is completely abstracted away from the modeling and rate changes happen between pairs of peers. This is a reasonable abstraction if we consider that the bottlenecks on a P2P network usually appear in the "last mile" rather than the Internet backbone. In doing so, the number of events simulated is further reduced, however in this case the inaccuracy remains since only the end-to-end effects are taken into account while the cascading effect on other nodes, as modeled by max-min fair allocation, is completely overlooked.

There exists two bandwidth allocation algorithms in the state of the art which apply the progressive filling idea on end-to-end network models, thus keeping the advantages of simulating only access links but still considering the effects and propagation of rate changes throughout the peer interconnections. The first algorithm proposed by F. Lo Piccolo et Al. [22] models the end-to-end network as an undirected graph. In each iteration, the algorithm finds the *bottleneck* nodes in the network, the nodes with the minimum fair bandwidth share available to their flows. Then it proceeds to allocate the calculated minimum fair share to the flows. The algorithm iterates until all nodes are found saturated or a rate is assigned to all their flows.

The main disadvantage of this node-based max-min fair bandwidth allocation algorithm lies in the modeling of the network as an undirected graph. In order to simulate a network with separate upload and download capacities, two node instances are required per actual network peer. The memory footprint is therefore larger than the one needed to model a direct network graph. Another weakness is that the computational complexity of the approach depends directly on the cardinality of the node set.

An alternative edge-based max-min bandwidth allocation algorithm is given by Anh Tuan Nguyen et al. [23]. It is an edge-based algorithm which uses a directed network model, differently from the approaches we introduced till now. In one iteration, the algorithm calculates the minimum fair share of the two ends of every unassigned flow. Then, on the same iteration and based on the previously calculated shares, the algorithm finds the bottleneck nodes, derives the flows' rates and applies them. The algorithm iterates until all flows have a rate assigned. It is important to underline that during the first phase of each iteration, the algorithm might find one or multiple bottleneck nodes, thus assigning rates to the flows of multiple nodes at the same iteration.

This edge-based max-min fair bandwidth allocation algorithm addresses the shortcomings of the undirected network modeling. However, the algorithm performance's dependence on the edge-set size constitutes a major drawback. This because each iteration requires the calculation of the minimum fair share for each unassigned flow. On top of that, a further iteration of the node set is required in order to find the saturated nodes. As we will see, this hidden complexity makes it unsuitable for simulating large networks with high adjacency.

It is common in large simulated networks for a new or finished flow to only affect the rates of a subset of the existing network flows, as the propagation of a rate change does not reach all nodes in the network but rather few of them. Based on this observation, F. Lo Piccolo et Al. [22] partially outline an affected subgraph traversal algorithm that can be applied on an undirected network graph. Starting from the end nodes of the flow triggering the reallocation, the algorithm traverses the segment of the network graph that is affected by the change. In fact, this is an attempt to trace the aforementioned ripple effect. The traverse continues through affected flows/edges until all the affected network subgraph is visited. The nodes reached by the traversal are treated based on the parity of the hop distance (odd or even) from any of the two root nodes, i.e. the initiators of the allocation/deallocation. Depending on the parity, a different subset of their flows is affected. The traverse continues until no new nodes are affected. Since a node can be reached in an odd or even hop distance from a root, a node can be visited maximum twice during a traverse. This results in a linear complexity with respect to the node set cardinality, assuming that a node is not reconsidered when reached again in the same distance from a root.

Using this optimization algorithm before applying an undirected node-based max-min fair bandwidth allocation algorithm leads to a large performance gain. Unfortunately, F. Lo Piccolo et Al. apply this only on an undirected network model. Moreover, the authors provide only a sketch of the affected subgraph idea rather than a state-complete algorithm.

In general, the aforementioned works which propose an improvement to the classical progressive filling focus their evaluation of the algorithms on performance while completely overlooking the accuracy of their approaches.

4.3 Contribution

We propose a scalable and efficient flow-level simulator which leverages the benefits of the affected subgraph optimization on a directed network model. We give a state-complete description of the subgraph optimization introduced by F. Lo Piccolo et Al. and we evaluate its performance gain when used in combination with our max-min fair bandwidth allocation algorithm. The result is an algorithm whose computational complexity is independent of the edge set size. Our experiments show that our solution constantly outperforms the edge-based algorithm proposed by Anh Tuan Nguyen et al. for large-scale and structured network overlays. Finally, we conduct a detailed accuracy study where we compare the simulated transfer times of our proposed solution with the ones obtained with NS-2 for the same realistic scenarios. We show that the bandwidth allocation follows the trends of the actual packet-level simulated bandwidth dynamics.

4.4 Proposed Solution

We model the network as a directed graph $G = (U, F)$, where the vertex set U represents the set of end nodes and the edge set F represents the set of directed flows between pairs of end nodes. We define as c_i^u and c_i^d the uploading and downloading capacities of node i which belongs to the node set U. The rate of a flow f_{ij} is denoted as r_{ij} where $i, j \in U$ and $i \neq j$. The existence of a flow from i to j does not imply the existence of a flow from j to i. As a result, every node $i \in U$ has two separate sets of flows, F_i^u for the outgoing, and F_i^d for the incoming. Finally, $r_{i_u}^*$ and $r_{i_d}^*$ are the fair shares of the upload and download capacity respectively of a node $i \in U$. A fair share is the equal division of a node's upload or donwload capacity to its flows.

4.4.1 Bandwidth allocation

We now present the first part of our solution: our node-based max-min fair bandwidth allocation in Algorithm 1.

The algorithm consists of a number of iterations up to the point where all flows have a rate assigned. Each iteration has two phases. In the first, we find the node(s) which

Algorithm 1 Node-based max-min fair bandwidth allocation algorithm

```
Input: A set of nodes $U$ with their upload and download bandwidth capacities
    $c_i^u, c_i^d, \forall i \in U$, and their corresponding flow sets $F_i^u$, $F_i^d$.
Output: The Max-Min fair rate allocation $\vec{r}$ of the bandwidth capacities to the flows.
begin
 1: $SatUp \leftarrow \emptyset$; $SatDown \leftarrow \emptyset$
 2: while $F \neq \emptyset$ do
 3:     $SatUp \leftarrow \{s_u | r^*_{s_u} = min_{i \in U, c_i^u \neq 0} \frac{c_i^u}{|F_i^u|}\}$
 4:     $SatDown \leftarrow \{s_d | r^*_{s_d} = min_{i \in U, c_i^d \neq 0} \frac{c_i^d}{|F_i^d|}\}$
 5:     $r^* = min(r^*_{s_u}, r^*_{s_d})$
 6:     if $r^*_{s_u} > r^*_{s_d}$ then
 7:         $SatUp \leftarrow \emptyset$
 8:     else if $r^*_{s_u} < r^*_{s_d}$ then
 9:         $SatDown \leftarrow \emptyset$
10:     end if
11:     if $SatUp \neq \emptyset$ then
12:         for all $i \in SatUp$ do
13:             for all $f_{ij} \in F_i^u$ do
14:                 $r_{ij} = r^*$
15:                 $c_j^d = c_j^d - r^*$
16:                 $F_j^d \leftarrow F_j^d - \{f_{ij}\}$
17:             end for
18:             $c_i^u = 0$; $F_i^u \leftarrow \emptyset$
19:         end for
20:     end if
21:     if $SatDown \neq \emptyset$ then
22:         for all $i \in SatDown$ do
23:             for all $f_{ji} \in F_i^d$ do
24:                 $r_{ji} = r^*$
25:                 $c_j^u = c_j^u - r^*$
26:                 $F_j^u \leftarrow F_j^u - \{f_{ji}\}$
27:             end for
28:             $c_i^d = 0$; $F_i^d \leftarrow \emptyset$
29:         end for
30:     end if
31: end while
end
```

provide the minimum fair share r^* (lines 4-10). The process happens as follows: we calculate the fair share of the upload and the download capacity of each node. Then, we find the nodes that provide the minimum upload fair share $r^*_{s_u}$ and the minimum

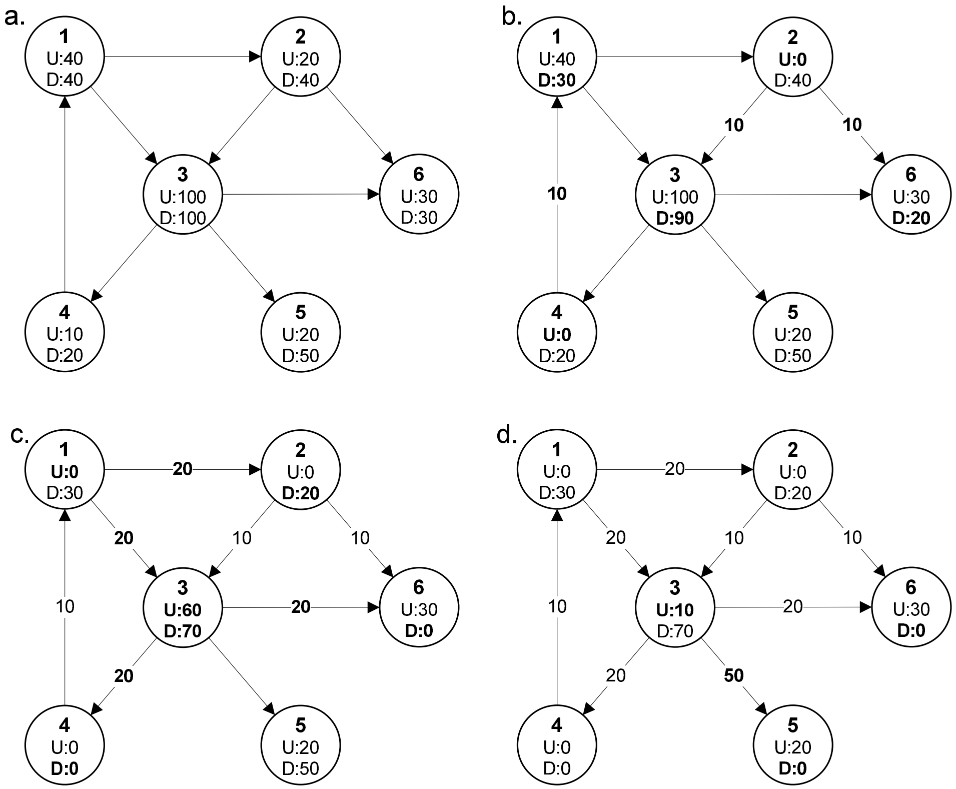


Figure 4.1: An example of the max-min bandwidth allocation algorithm.

download fair share $r^*_{s_d}$ and add them to node set *SatUp, SatDown* respectively (lines 4-5). The smallest rate between $r^*_{s_u}$ and $r^*_{s_d}$ is set as the minimum fair share r^* and the nodes of the corresponding set are considered saturated (line 6-10), i.e. they constitute the bottleneck of the network in this iteration. In the case where $r^*_{s_d} = r^*_{s_d}$, then all nodes in *SatUp* and *SatDown* are considered saturated. In the second phase, we allocate the minimum fair share r^* to the flows of each saturated node, either to their downloading or uploading side (lines 11-24). The available bandwidth on the other end of each flow (c^u_j for a downloading flow or c^d_j for an uploading flow) is reduced and the assigned flows are removed from the flow sets (lines 15-16, 22-23). The saturated nodes have no available capacity left on their affected side (downloading/uploading) and their corresponding flow sets are empty (17, 24).

An example of how our max-min fair allocation algorithm works is shown in Figure 4.1. We consider the network graph shown in Figure 4.1.a. In the first iteration, the minimum fair share in the graph is found to be 10, this is provided by the upload capacity of node 2 and node 4. The state of the graph after the allocation of the minimum fair share to flows $f_{2,3}$, $f_{2,6}$ and $f_{4,1}$ is shown in Figure 4.1.b. In the second iteration, the minimum fair share is 20 and is provided by the upload side of node 1, as well as the downloading side of node 4 and node 6. The minimum fair share is assigned to flows $f_{1,2}$, $f_{1,3}$, $f_{3,4}$ and $f_{3,6}$, which results to the network state shown in Figure 4.1.c. Finally, in the last iteration, the download side of node 5 provides the minimum fair share of 50, which is allocated to its downloading flow $f_{3,5}$. The final rate allocation for this network graph is shown in Figure 4.1.d.

The complexity of the algorithm is $O(N^2)$ since at least one node side is found saturated in each iteration. In order to improve scalability, we adapt the affected subgraph discovery algorithm for use with directed end-to-end network models, i.e. where only access links capacity is modeled.

4.4.2 Affected subgraph algorithm

This optimization is essentially an affected subgraph discovery algorithm. Given a flow that triggers a bandwidth reallocation, the algorithm initiates two graph traversals that each one has as root one of the flow's end nodes. In each hop of a traversal we find the affected flows of the last reached nodes and continue the traverse to their other ends. This procedure continues until no newly affected nodes are discovered by any of the two traversals. We characterize the nodes reached by one of these traversals based on three criteria: (1) *the parity of the distance in hops from the root node*, i.e. odd or even; (2) *the type of the root node from which the traversal originates*, i.e. the uploading or downloading end of the triggering flow; and (3) *the type of the triggering event*, i.e. new or finished flow.

Furthermore, the flows of a node i are distinguished between locally bottlenecked, L_i, and remotely bottlenecked, R_i. A node's flow is referred as *locally bottlenecked* if its own capacity is restricting the bandwidth rate of the flow. In contrast, a node's flow is referred as *remotely bottlenecked* if the node at the other end of the flow is the one restricting its bandwidth rate. For example, the flow $f_{4,1}$ in Figure 4.1.d is locally bottlenecked for node 4 since it is its upload capacity that restricts the flow's rate. In contrast, for node 1 the flow is considered remotely bottlenecked since its rate is restricted by the capacity available to the node on the other end of the flow.

A node can be affected in two different ways. First, it might need to accommodate a new flow or a rate increase of an already existing flow. Second, it might have some of its bandwidth freed due to a finished flow or a flow that has decreased its rate. Since we distinguish between the upload and download side of a node, we have a total of four different ways in which a a node can be affected:

- **upload rate increase**
  A node in this category is asked for a rate increase. The criteria configurations which correspond to this group of nodes are {even, uploading, new} and {odd, downloading, finished}. This means that a node will have all the locally bottlenecked uploading flows affected since they will need to reduce their rates in order to accommodate a new flow or a bandwidth increase. Moreover, it might hap-

Algorithm 2 Directed affected subgraph algorithm

Input: A set of nodes U with their upload and download capacities $c_i^u, c_i^d \forall i \in U$, their flow sets F_i^u, F_i^d and a triggering flow f_{r_1,r_2}.

Output: The subsets AU and AF of U and F respectively, that correspond to the affected network subgraph.

```
begin
$hop \leftarrow 0;\ AU \leftarrow \emptyset;\ AF \leftarrow \emptyset$
if $f_{r_1,r_2} \in F$ then
    $AF \leftarrow AF \cup \{f_{r_1,r_2}\}$
end if
$SAU \leftarrow \{r_1\};\ DAU \leftarrow \{r_2\}$
while $SAU \neq DAU \neq \emptyset$ do
    $AU \leftarrow AU \cup SAU \cup DAU$
    $SAU' \leftarrow \emptyset;\ DAU' \leftarrow \emptyset$
    for all $i \in SAU$ do
        if $(hop \bmod 2 \neq 0$ and $f_{r_1,r_2} \in F)$ or $(step \bmod 2 == 0$ and $f_{r_1,r_2} \notin F)$ then
            $SAU' \leftarrow SAU' \cup DownDecrease(i)$
        else if $(hop \bmod 2 \neq 0$ and $f_{r_1,r_2} \notin F)$ or $(hop \bmod 2 == 0$ and $f_{r_1,r_2} \in F)$ then
            $SAU' \leftarrow SAU' \cup DownIncrease(i)$
        end if
    end for
    for all $i \in DAU$ do
        if $(hop \bmod 2 \neq 0$ and $f_{r_1,r_2} \in F)$ or $(hop \bmod 2 == 0$ and $f_{r_1,r_2} \notin F)$ then
            $DAU' \leftarrow DAU' \cup UpDecrease(i)$
        else if $(hop \bmod 2 \neq 0$ and $f_{r_1,r_2} \notin F)$ or $(hop \bmod 2 == 0$ and $f_{r_1,r_2} \in F)$ then
            $DAU' \leftarrow DAU' \cup UpIncrease(i)$
        end if
    end for
    $SAU \leftarrow SAU';\ DAU \leftarrow DAU'$
    $hop \leftarrow hop + 1$
end while
end
```

Algorithm 3 "UpDecrease(node)" returns the nodes reached by means of locally bottlenecked uploading flows.

```
procedure UpDecrease(A node $i$) ⟨The set of nodes reached by the affected locally bottlenecked uploading flows of $i$⟩
    $AF \leftarrow AF \cup \{f_{ij} | f_{ij} \in F_i^u$ and $f_{ij} \in L_i^u\}$
    return $\{j | f_{ij} \in F_i^u$ and j unaffected by download increase and $f_{ij} \in L_i^u\}$
end procedure
```

Algorithm 4 "UpIncrease(node)" returns the nodes reached by means of locally and remotely bottlenecked uploading flows.

```
procedure UpIncrease(A node $i$) ⟨The set of nodes reached by affected locally and remotely bottlenecked uploading flows of $i$⟩
    $AF \leftarrow AF \cup \{f_{ij} | f_{ij} \in F_i^u$ and $(f_{ij} \in L_i^u$ or $(f_{ij} \in R_i^u$ and (4.1) false for $r_{ij}))\}$
    return $\{j | f_{ij} \in F_i^u$ and j unaffected by download decrease and $(f_{ij} \in L_i^u$ or $(f_{ij} \in R_i^u$ and (4.1) false for $r_{ij}))\}$
end procedure
```

pen that some or all of the remotely bottlenecked outgoing flows might turn into locally bottlenecked. We can find the remotely affected bottlenecked flows of a node x by evaluating the following expression, as proposed by [22]:

$$A : r_i < \frac{c_x^u - \sum_{j=1}^{i-1} r_j}{|F_x^u| - i + 1}, i \in |R_x^u|, \tag{4.1}$$

c_x^u is the upload capacity of x, and r_i, where $i \in R_x^u$, the rate of a remotely bottlenecked uploading flow i of x. We consider the remotely bottlenecked flows ordered by increasing rate size: $r_1 \leq r_2 \leq \cdots \leq r_{|R_x^u|}$. The above condition is evaluated starting from $i = 1$ until it is either true for all $i \in R_x^u$ or becomes false for a certain i. In the first case all remotely bottlenecked flows remain unaffected. In the second case the flows with rates $r_i, r_{i+1}, ..., r_{|R_x^u|}$ might turn into locally bottlenecked by the rate increase and thus should be considered as affected.

- **upload rate decrease**
  The nodes included in this category got affected due to a rate decrease of one of their uploading flows. The criteria configurations for these nodes are {odd, downloading, new} and {even, uploading, finished}. Nodes falling into this category need to allocate a released amount of upload/download bandwidth to their currently active flows. This leads all their locally bottlenecked outgoing flows to be affected by the operation.

- **download rate increase**
  These nodes are affected by a rate increase in the same manner as the ones in the first category but by means of a downloading flow. The criteria configurations corresponding to this node category are {even, downloading, new} and {odd, uploading, finished}.

- **download rate decrease**
  Like the nodes of the second category, these nodes experience a rate decrease but this time caused by a downloading flow. The criteria configurations which represent this category are {odd, uploading, new} and {even, downloading, finished}.

During the same round of affected subgraph discovery, a node side can be reached both by a request to increase and by a request to decrease its flow rates. This makes for a clear conflict. In the original algorithm, the conflict is resolved by ignoring the change request that affects less children nodes from the current node's point of view. The approach clearly introduces inaccuracy since the propagation of a rate change is stopped abruptly. In our solution, we do not discriminate between the way by which a node is affected. However, this might introduce loops along the traversal path, in order to avoid this situation, we stop the traversal if a node gets affected twice in the same way, e.g. if it gets an increase request from its uploading flows twice in the same affected subgraph discovery.

The state-complete algorithm is shown in Algorithm 2 and Procedures $UpIncrease$ and $UpDecrease$. We do not show procedures $DownIncrease$ and $DownDecrease$ since they are the complement of $UpIncrease$ and $UpDecrease$. The algorithm begins by checking the type of the triggering event. In the case that it is a new flow f_{r_1,r_2}, it is added to the affected flow set AF (lines 3-4). A traversal is then initiated from each of the two root nodes, r_1, r_2. Two sets SAU and DAU containing the last reached nodes, for the traversals originating from r_1 and r_2 respectively, are updated at each hop. Initially, both sets contain only the root node (line 5). At each iteration, we check the last reached nodes and classify them based on the proposed categorization (lines 9-18). Depending on the parity of the hop distance, the root node (uploading/downloading) and the type of the triggering event (new or finished flow), we define which will be the nodes affected in the next iteration using Procedures $UpIncrease$, $UpDecrease$, $DownIncrease$ and $DownDecrease$. As a result, two sets of newly affected nodes are produced after the iteration: one set (SAU') for the traversal originating from the uploading side of the triggering flow and one (DAU') for the traversal originating from the downloading side of the triggering flow. These node sets will be provided as input to the next iteration of the algorithm. The traversal continues until no more affected nodes are found (line 6). The complexity of the algorithm is $O(N)$ since each node can be reached at maximum in 4 different ways.

In order to illustrate the affected subgraph discovery algorithm, we extend the annotation used in [22] in order to show both the direction and the bottleneck side of a flow. The source of a flow is depicted with a circle and the destination with an arrow. Their color shows the end of the flow that constitutes the bottleneck on a pairwise fashion. We denote with a white circle or arrow the restrictive end, i.e. the flow is locally bottlenecked. In contrast, we denote as a black circle or arrow the end of a flow that is remotely bottlenecked. If we use this annotation on the previous network graph example in Figure 4.1.a, we obtain the graph shown in Figure 4.2.a.

Lets consider the case of a new flow from node 2 to node 1. The algorithm firsts checks the outgoing flows of the uploading root node 2 and the incoming flows of the downloading root node 1 to find the affected ones, Figure 4.2.b. Since the triggering flow is a new one, all the locally and probably some remotely bottlenecked flows of its end nodes are affected. The uploading root node 2 has two locally bottlenecked uploading flows, $f_{2,3}$ and $f_{2,6}$, that are affected. The downloading root node 1 has only one remotely bottlenecked downloading flow $f_{4,1}$ which is not affected since (4.1) is true for its rate. In the next hop, the download flows of nodes 3 and 6 are checked, Figure 4.2.c. Since they are in an odd hop distance from the uploading side of a new transfer, only

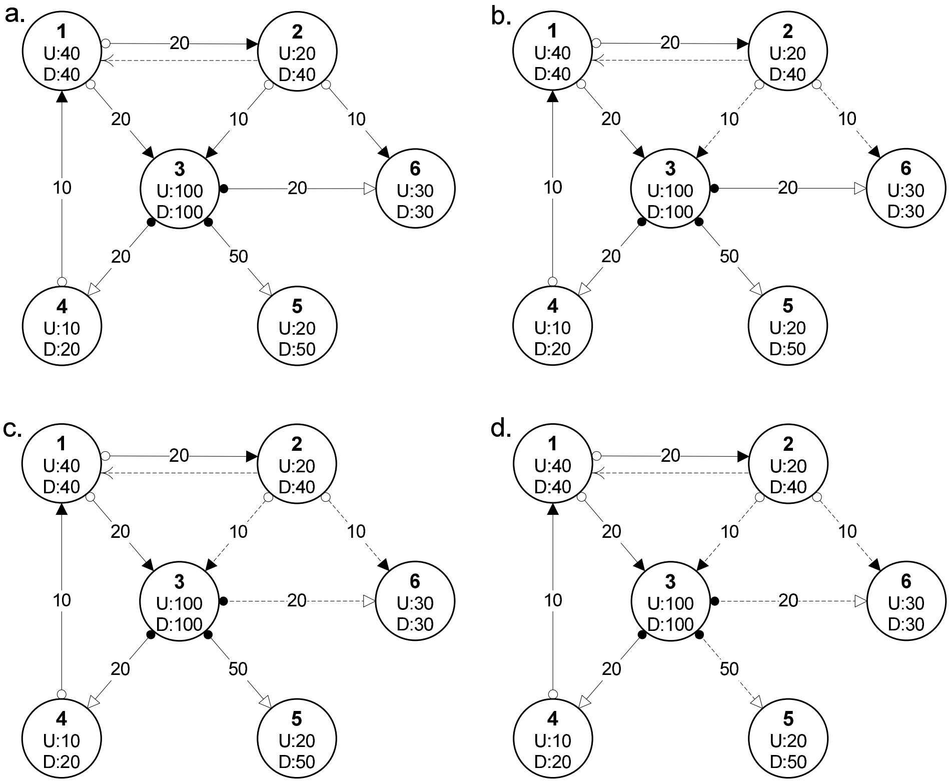


Figure 4.2: An example of the affected subgraph discovery.

their locally bottlenecked downloading flows will be affected by means of a download decrease. Node 3 has no locally bottlenecked downloading flows while node 6 has only one, $f_{3,6}$. The next hop of the traversal reaches node 6 at an even hop distance from the uploading side of new flow, it should have all his locally and maybe some of his remotely bottlenecked flows affected, Figure 4.2.d. Node 6 has only remotely bottlenecked uploading flows and one of them, $f_{3,5}$, has a rate for which (4.1) is evaluated as false. This flow is considered thus as affected. Finally, since node 5 has no unaffected flows, the algorithm terminates.

4.5 Evaluation

4.5.1 Methodology

We focus in the evaluation of our proposed solution on the two desirable characteristics of a P2P network simulator: scalability and accuracy. In our scenarios, we consider two main parameters: the size of the node set and the number of outgoing flows per node.

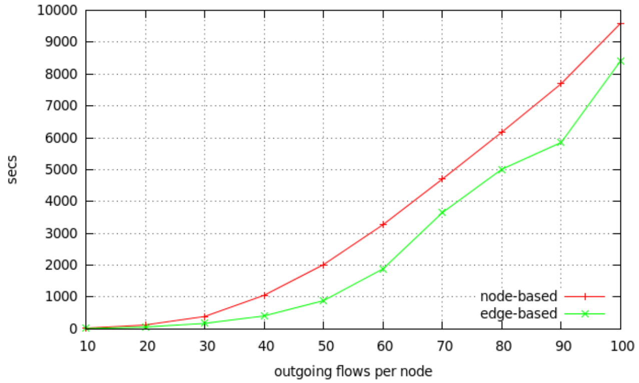


Figure 4.3: Simulation times of a random network overlay. 1000 nodes network size and varying number of outgoing flows per node.

The destination of the flows is chosen either randomly or following a specific structure, i.e. a DHT-based one. The nodes enter the system in groups at defined time intervals. The starting times of a joined node's flows are distributed uniformly in that same time interval. The node's bandwidth capacities are either symmetric, asymmetric or mixed depending on the experiment. Finally, the amount of transferred bytes per flow is also a parameter.

For our accuracy study we implemented the same scenario both on our simulator and on the NS-2 packet-level simulator. In NS-2, we use a simple star topology, similar to the one used in [19] [24]. Each node has a single access link which we configure with corresponding upload and download capacities. All flows pass from the source access link to the destination access link through a central node with infinite bandwidth capacity. We use the TCP Reno implementation with a packet size of 1460 Bytes. The queue mechanism used is Drop Tail and all links have 2ms delay, resulting in a 8ms RTT. Finally, the TCP queue and the maximum window sizes are defined taking into consideration the bandwidth delay product.

We repeat each experiment a number of times with different random seeds and we take the average value of the resulting measurements. In the scalability study, we consider the average simulation time of the runs, while in the accuracy study we analyze the average deviation between the NS-2 data transfers and the corresponding flows' time modeled in our simulator. The machine used for this evaluation has a dual core processor with 2.1GHz per core, 3MB cache and 3GB RAM.

4.5.2 Scalability

In our scalability evaluation, we vary the size of node sets, in the scale of thousands, and the number of outgoing flows per node, in increments of tens. Both random and structured overlays are characterized by a specific degree of inter-flow dependency. We

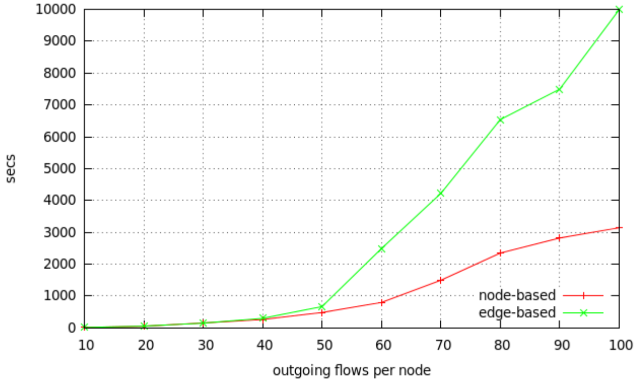


Figure 4.4: Simulation times for a DHT-like structured network overlay. 1000 nodes and varying number of outgoing flows per node.

can define the latter as how many flows on average are affected by a new or finished flow. i.e. the average ripple effect's scope. In the random scenario, it is more likely for the destination of the flow not being saturated, whereas in the structured, we guarantee a minimum level of saturation at each receiving end.

The rest of the scenario parameters are set as follows: the size of each flow is 2MB plus the size of TCP headers. Nodes join in groups of 50 every 20 seconds. The starting times of the flows of a newly joined node are uniformly distributed on the interval period. The bandwidth capacities of a node are chosen randomly from the set: {100Mbps/100Mbps,24Mbps /10Mbps,10Mbps/10Mbps,4Mbps/2Mbps,2Mbps/500Kbps} with corresponding probabilities of {20%,40%,10%,10%}.

We first compare our proposed node-based max-min fair bandwidth allocation algorithm with the edge-based proposed by [23]. The simulation times required by both algorithms for a random scenario of networks with 1000 nodes and varying number outgoing flows per node are shown in Figure 4.3. The edge-based algorithm appears to perform slightly better than our solution. This is expected since the edge-based algorithm finds more saturated nodes per iteration, as mentioned in Section 4.2.

We run the same scenarios but this time selecting the destination nodes of the flows in a more structured manner. The selection is done in such a way that nodes form a circle where each node's outgoing flows are directed to the nearest neighbors in a clockwise fashion. The simulation times required by both algorithms for this scenario are shown in Figure 4.4. It is clear that the edge-based algorithm is significantly slower than our proposed algorithm when it comes to networks with strong inter-flow dependencies. This because the higher computational complexity of the edge-based algorithm emerges in those cases where stronger relations between the flows exist.

We next study the simulation times when dealing with larger size networks. We fix each node's outgoing flows to 20 and vary the node set size between 1000 and 10000 nodes. The results for the random and structured overlay scenarios are shown in Figure

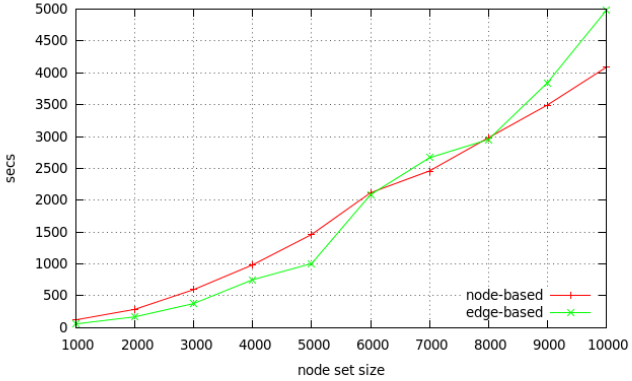


Figure 4.5: Simulation time for varying sizes of a random overlay networks with a fixed 20 outgoing flows per node.

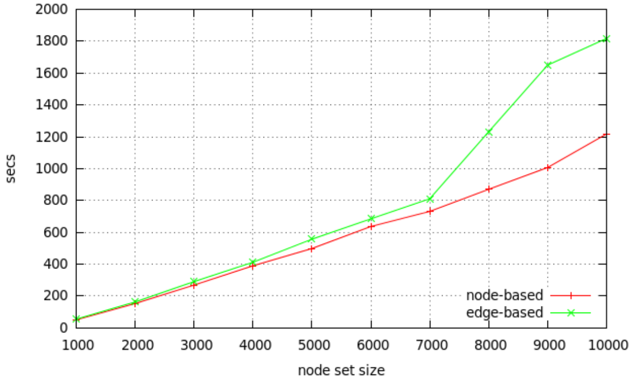


Figure 4.6: Simulation time for different sizes of a structured overlay networks with fixed 20 outgoing flows per node.

4.5 and Figure 4.6 respectively. When random overlays are considered, we observe that, as the node set gets bigger, the edge-based algorithm performance deteriorates and is outperformed by our proposed algorithm. This can be explained by the fact that the edge-based algorithm includes an extra check of the complete node set per each iteration. This may not have a big impact when considering relatively small networks, but it might constitute a performance bottleneck for large scale simulations. The impact of this bottleneck becomes more apparent in the case of structured overlays where less nodes are found saturated at every iteration.

Based on these results, we can state that our proposed max-min fair bandwidth allocation algorithm performs better when simulating large and strongly connected networks with high inter-flow dependency. An other important conclusion drawn from

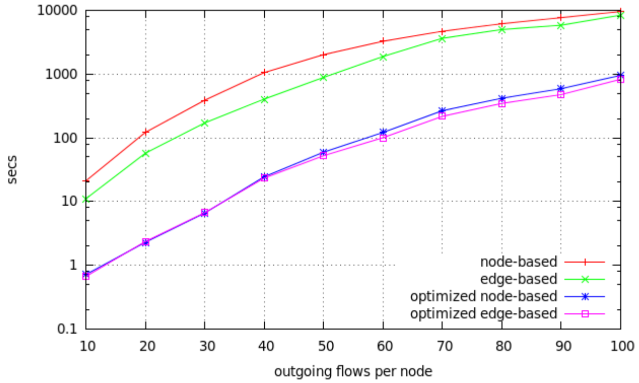


Figure 4.7: Performance improvement when using the affected subgraph algorithm on random network overlays with 1000 nodes and different number of outgoing flows per node.

these results is that the number of connections per node has a bigger impact in the performance of the simulation models rather than the node set size. For example, the simulation of a random overlay of 1000 nodes with 70 outgoing flows requires a similar simulation time to a 10000 nodes random overlay having 20 outgoing flows per node.

We also would like to point out that the performance gain when using flow-level simulation instead of packet-level is paramount. In order to simulate a random scenario of 1000 nodes with 10 flows each, a time of three orders of magnitude longer is required.

We run the same overlay scenarios using the directed affected subgraph algorithm defined in Algorithm 2 before running the max-min fair bandwidth allocation algorithm. The results are shown in Figures 4.7-4.10. It is clear that the optimization reduces significantly the simulation time. In Figures 4.7 and 4.9 we can see that the required time is one order of magnitude lower when simulating network overlays with the same size but different number of outgoing flows per node. The performance improvement is much higher, three orders of magnitude, when we increase the network size and keep the number of flows per node fixed, as shown in Figures 4.8 and 4.10. This can be explained by the fact that the affected subgraph size, on which a max-min fair bandwidth allocation algorithm is applied, mainly depends on the connectivity of the network. It should be also pointed out that the performance is similar for both our and the edge-based max-min bandwidth allocation algorithms when they are used together with the affected subgraph algorithm. This is because the affected subgraphs in these cases are not big enough to point out the performance differences between the two algorithms.

4.5.3 Accuracy

The goal of this study is to find out how accurately the bandwidth capacity is allocated to competing flows with respect to a packet-level simulator, such as NS-2. Unfortunately, the size of our experiments is limited by the low scalability of NS-2. We run a scenario

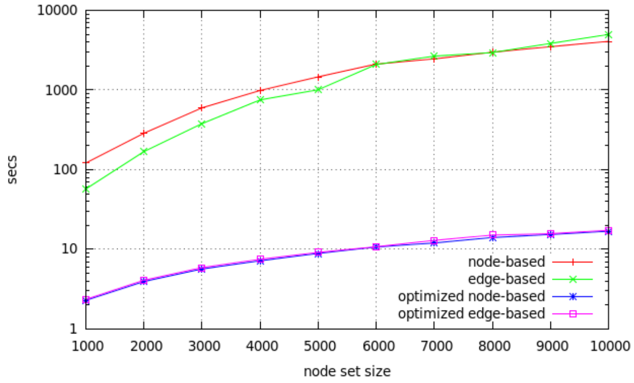


Figure 4.8: Performance improvement when using the affected subgraph algorithm on random network overlays with varying size and 20 outgoing flows per node.

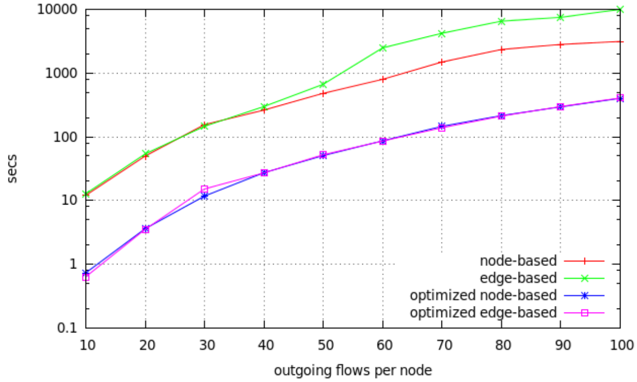


Figure 4.9: Performance improvement when using the affected subgraph algorithm on structured network overlays with 1000 nodes and different number of outgoing flows per node.

with 100 nodes joined at the start of the simulation. The destination of the nodes' outgoing flows is random, the size of each flow is 4MB and their number vary between 10 and 50 per node.

As in [19], we use the the relative error in transfer times as our accuracy metric. Let t_{NS-2} be the time at which a flow terminates in NS-2 and $t_{max-min}$ the time at which our max-min fair flow terminates. The relative error is then given by:

$$RE = \frac{t_{flow} - t_{NS-2}}{t_{NS-2}} \tag{4.2}$$

In our flow simulation, we ignore any segmentation or retransmission of packets. The packet header overhead introduced by the packet level simulation is added to the flow

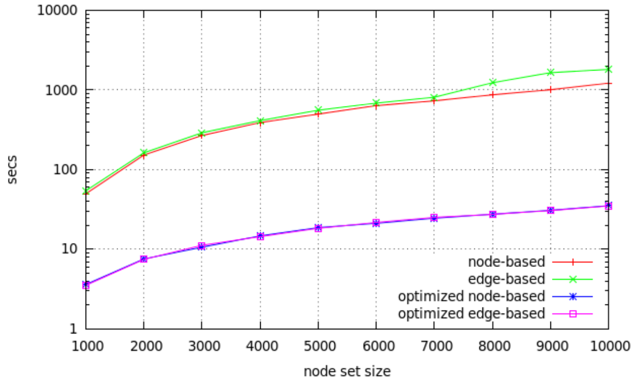


Figure 4.10: Performance improvement when using the affected subgraph algorithm on structured network overlays with varying size and 20 outgoing flows per node.

size.

We fist consider the case where the access link's capacities are symmetric, i.e. same upload and download capacity that we set to 10Mbps. Results for different number of outgoing flows per node are shown in Table 4.1. The standard and average deviation of the relative error of the transfer times are given for each of the scenarios.

We can see that the deviation is smaller when the network is more under stress, i.e. more outgoing flows per node. In these situations, TCP converges quicker causing the transfer times to deviate less. Moreover, since both sides of every flow have the same bandwidth capacity, the share that they provide should be similar. This leads to an oscillation of the window size that translates into an extra transfer time deviation. However, while the absolute of the deviation varies with the number of links, we can still assert that the deviation is nearly constant, e.g. max 0.5%, between different iterations of the same experiment and different flows in the same experiment, for each one of the experiments conducted. In terms of absolute deviation, a flow-level simulator cannot compete with a packet-level because of the different level of abstraction. But we can safely state that, if the deviation is constant, the flow-level simulation follows the behavior of the packet-level simulation by the amount of that constant value, which is the desired effect of the simulator.

In the next experiment, we show the impact of the capacity size as well as the interaction between nodes with different symmetric capacities. For this purpose, we run the same scenario but this time setting half of the nodes in the network to symmetric bandwidth capacities of 20Mbps. The results are shown in Table 4.1. The introduction of nodes with higher capacity speeds up the transfer times, thus providing less time for TCP to converge. This effect leads to more deviation. We observe that the number of the outgoing flows per node does not affect the deviation. This might be because the impact of the higher bandwidth capacities to the time deviation is larger. Again, we can assert that the flow-level simulation follows the behavior of the packet-level one by a

c^u/c^d	flows	std. deviation	avg. deviation
10/10	10	9.6±0.6%	7.9±0.6%
	20	7.3±0.3%	5.9±0.2%
	30	6.1±0.3%	5±0.2%
	40	5.1±0.3%	4±0.3%
	50	4.4±0.2%	3.6±0.2%
10/10, 20/20	10	12.7±1.0%	10.5±0.9%
	20	13.7±0.4%	11.3±0.4%
	30	12.6±0.3%	10.8±0.4%
	40	13.1±0.4%	11.4±0.5%
	50	13.2±0.3%	11.5±0.4%

Table 4.1: Deviation of transfer times.

nearly constant degree.

Next we consider the case of asymmetric bandwidth capacities. We assign to every node 20Mbps of download capacity and 10Mbps of upload capacity. The results for different number of outgoing flows per node are shown in Table 4.2. It appears that the deviation is significantly smaller and not affected by the per node outgoing flows comparing to the previous symmetric scenario. This can be explained by the absence of oscillation and quicker TCP convergence due to the larger download capacity.

Finally, we investigate the deviation when both symmetric and asymmetric node capacities are used. We set half of the nodes with a symmetric 10Mbps capacity and the rest with asymmetric capacity of 20Mbps download and 10Mbps upload. The results are also given in Table 4.2. We can see that the presence of the symmetric capacities affects negatively the transfer time deviation. The negative impact is more visible when fewer outgoing flows per node are used. When the links are less congested, the slower convergence of the flow rates of the nodes with smaller symmetric capacities is more apparent.

In the last two experiments, as in the first two, we measured a deviation from the NS-2 packet-level simulation but our max-min fair simulation followed the trends of the ones provided by NS-2 by an almost constant deviation factor.

4.6 Conclusion & Future Work

In this paper we presented a scalable and efficient flow-level network simulation model based on the max-min fairness idea. We evaluated our solution in terms of scalability by showing that it outperforms the existing state-of-the-art for large-scale and structured network overlays where a directed network is used for the modeling. In terms of accuracy we showed that our approach follows the trends of the NS-2 packet-level simulator network by a nearly constant factor throughout the experiments.

Our ongoing work includes the improvement of the model's accuracy by considering the time period that a flow requires to converge to a new rate, and also the use of time-

c^u/c^d	flows	std. deviation	avg. deviation
20/10	10	3.8±0.4%	3±0.4%
	20	3.9±0.2%	3.1±0.1%
	30	4.1±0.3%	3.3±0.2%
	40	3.4±0.2%	2.8±0.2%
	50	3±0.1%	2.5±0.2%
20/10,10/10	10	6.4±0.4%	5±0.4%
	20	6±0.4%	4.9±0.3%
	30	4.8±0.4%	3.9±0.3%
	40	3.4±0.9%	3.2±0.3%
	50	3.5±0.2%	2.8±0.2%

Table 4.2: Deviation of transfer times in the presence of asymmetric node capacities.

stepping aggregation to achieve higher performance with minimum cost in accuracy.

4.7 References

[1] Anwar Al Hamra, Arnaud Legout, and Chadi Barakat. Understanding the properties of the bittorrent overlay. Technical report, INRIA, 2007.

[2] Chuan Wu, Baochun Li, and Shuqiao Zhao. Magellan: Charting large-scale peer-to-peer live streaming topologies. In *ICDCS '07: Proceedings of the 27th International Conference on Distributed Computing Systems*, page 62, Washington, DC, USA, 2007. IEEE Computer Society.

[3] Guillaume Urvoy-Keller and Pietro Michiardi. Impact of inner parameters and overlay structure on the performance of bittorrent. In *INFOCOM '06: Proceedings of the 25th Conference on Computer Communications*, 2006.

[4] B. Cohen. Incentives Build Robustness in BitTorrent. In *Econ '04: Proceedings of the Workshop on Economics of Peer-to-Peer Systems*, Berkley, CA, USA, June 2003.

[5] S. Naicken, B. Livingston, A. Basu, S. Rodhetbhai, I. Wakeman, and D. Chalmers. The state of peer-to-peer simulators and simulations. *SIGCOMM Computer Communication Review*, 37(2):95–98, 2007.

[6] Ingmar Baumgart, Bernhard Heep, and Stephan Krause. Oversim: A flexible overlay network simulation framework. In *GI '07: Proceedings of 10th IEEE Global Internet Symposium*, pages 79–84, Anchorage, AL, USA, May 2007.

[7] Sam Joseph. An extendible open source p2p simulator. *P2P Journal*, 0:1–15, 2003.

[8] Jordi Pujol-Ahullo, Pedro Garcia-Lopez, Marc Sanchez-Artigas, and Marcel Arrufat-Arias. An extensible simulation tool for overlay networks and services. In *SAC '09:*

Proceedings of the 24th ACM Symposium on Applied Computing, pages 2072–2076, New York, NY, USA, March 2009. ACM.

[9] Alberto Montresor Mark Jelasity and Ozalp Babaoglu. A modular paradigm for building self-organizing peer-to-peer applications. In *Engineering Self-Organising Systems*, pages 265–282, 2003.

[10] Nyik San Ting and Ralph Deters. 3ls - a peer-to-peer network simulator. In *P2P '03: Proceedings of the 3rd International Conference on Peer-to-Peer Computing*, page 212. IEEE Computer Society, August 2003.

[11] David M. Nicol, Jason Liu, Michael Liljenstam, and Guanhua Yan. Simulation of large scale networks i: simulation of large-scale networks using ssf. In *WSC '03: Proceedings of the 35th Conference on Winter simulation*, pages 650–657. Winter Simulation Conference, December 2003.

[12] Andras Varga and Rudolf Hornig. An overview of the omnet++ simulation environment. In *Simutools '08: Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*, pages 1–10, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[13] The network simulator ns-2. http://www.isi.edu/nsnam/ns/, October 2010.

[14] Thomer M. Gil, Frans Kaashoek, Jinyang Li, Robert Morris, and Jeremy Stribling. p2psim: a simulator for peer-to-peer (p2p) protocols. http://pdos.csail.mit.edu/p2psim/, October 2006.

[15] Wu Kun, Dai Han, Yang Zhang, Sanglu Lu, Daoxu Chen, , and Li Xie. Ndp2psim: A ns2-based platform for peer-to-peer network simulations. In *ISPA '05: Proceedings of Parallel and Distributed Processing and Applications, Workshops*, volume 3759, pages 520–529. Springer Berlin / Heidelberg, November 2005.

[16] G. Kesidis, A. Singh, D. Cheung, and W.W. Kwok. Feasibility of fluid event-driven simulation for atm networks. In *GLOBECOM '96: Proceedings of the Global Communications Conference*, volume 3, pages 2013 –2017, November 1996.

[17] Daniel R. Figueiredo, Benyuan Liu, Yang Guo, James F. Kurose, and Donald F. Towsley. On the efficiency of fluid simulation of networks. *Computer Networks*, 50(12):1974–1994, 2006.

[18] Dimitri Bertsekas and Robert Gallager. *Data Networks*. Prentice Hall, second edition, 1992.

[19] Abdulhalim Dandoush and Alain Jean-Marie. Flow-level modeling of parallel download in distributed systems. In *CTRQ '10: Third International Conference on Communication Theory, Reliability, and Quality of Service*, pages 92 –97, 2010.

[20] W. Yang and N Abu-Ghazaleh. Gps: a general peer-to-peer simulator and its use for modeling bittorrent. In *MASCOTS '05: Proceedings of 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 425–432, Atlanta, Georgia, USA, September 2005.

[21] Thomas J. Giuli and Mary Baker. Narses: A scalable flow-based network simulator. *Computing Research Repository*, cs.PF/0211024, 2002.

[22] F. Lo Piccolo, G. Bianchi, and S. Cassella. Efficient simulation of bandwidth allocation dynamics in p2p networks. In *GLOBECOM '06: Proceedings of the 49th Global Telecommunications Conference*, pages 1–6, San Franscisco, California, November 2006.

[23] Anh Tuan Nguyen and F. Eliassen. An efficient solution for max-min fair rate allocation in p2p simulation. In *ICUMT '09: Proceedings of the International Conference on Ultra Modern Telecommunications Workshops*, pages 1 –5, St. Petersburg, Russia, October 2009.

[24] Tobias Hossfeld, Andreas Binzenhofer, Daniel Schlosser, Kolja Eger, Jens Oberender, Ivan Dedinski, and Gerald Kunzmann. Towards efficient simulation of large scale p2p networks. Technical Report 371, University of Wurzburg, Institute of Computer Science, Am Hubland, 97074 Wurzburg, Germany, October 2005.

PART III

Network Layer

CHAPTER 5

NATCracker: NAT Combinations Matter

NATCracker: NAT Combinations Matter

Roberto Roverso[1,2], Sameh El-Ansary[1] and Seif Haridi[2]

[1]Peerialism AB
Stockholm, Sweden
[roberto,sameh]@peerialism.com

[2]KTH - Royal Institute of Technology
Stockholm, Sweden
[haridi]@kth.se

Abstract

In this paper, we report our experience in working with Network Address Translators (NATs). Traditionally, there were only 4 types of NATs. For each type, the (im)possibility of traversal is well-known. Recently, the NAT community has provided a deeper dissection of NAT behaviors resulting into at least 27 types and documented the (im)possibility of traversal for some types. There are, however, two fundamental issues that were not previously tackled by the community. First, given the more elaborate set of behaviors, it is incorrect to reason about traversing a single NAT, instead combinations must be considered and we have not found any study that comprehensively states, for every possible combination, whether direct connectivity with no relay is feasible. Such a statement is the first outcome of the paper. Second, there is a serious need for some kind of formalism to reason about NATs which is a second outcome of this paper. The results were obtained using our own scheme which is an augmentation of currently-known traversal methods. The scheme is validated by reasoning using our formalism, simulation and implementation in a real P2P network.

5.1 Introduction

Dealing with Network Address Translators (NATs) is nowadays an essential need for any P2P application. The techniques used to deal with NAT have been more or less “coined” and there are several widely-used methods[1][2]. Some of them are rather a defacto standard like STUN [3],TURN [4],ICE [5].

In the context of our a P2P live video streaming application PeerTV, we are mainly concerned with media streaming using UDP and therefore the scope of this paper is UDP NAT traversal. Moreover, we are strictly interested in solutions that do not use relay, such as TURN for instance, due to the high bandwidth requirements of video streaming.

We have found lots of of previous work on the subject that aims to answer the following question: *For every t in the set of NAT types $\mathscr{T}$, which s in the set of traversal strategies $\mathscr{S}$ should be used to traverse t? The answer is of the form $f : mathcal T \rightarrow \mathscr{S}$.* i.e. the following is an example with a couple of types f : { Simple Hole Punching, Port-Prediction } → { Full-Cone, Symmetric} [6].

However, the point which we found not gaining enough attention is that the presence of a feasible traversal technique that enables two peers behind NAT to communicate depends on the "combination" of the NAT types and not on the type of each peer separately. Thus, the question should be: *"Given* 2 *peers p_a and p_b with respective NAT types $t(p_a)$ and $t(p_b)$, which traversal strategy s is needed for p_1 and p_2 to talk? The answer is of the form $f : \mathscr{T} \times \mathscr{T} \rightarrow \mathscr{S}$"*, i.e we need to analyze traversable combinations rather than traversable types.

Most works contain a few examples of combinations for explanation purposes [6][7]. However, we have failed to find any comprehensive analysis that states, for every possible combination of NAT types, whether *direct* (i.e. with no relay) connectivity is possible and how. The analysis is more topical given that NAT community is switching from the classical set of NAT types $\mathscr{T}_{classic}$ = { Full-Cone, Restricted-Cone, Port-Restricted, Symmetric} [3] to a more elaborate set that defines a NAT type by a combination of three different policies, namely, port mapping, port allocation and port filtering [8]. With that, a statement like "two peers behind symmetric NAT can not communicate" becomes imprecise, as we will show that in many cases it is possible given the nuances available in the presently wide spectrum of NAT types.

5.2 Related Work

The work in [7] includes a matrix for a number of combinations, however mostly drawn from $\mathscr{T}_{classic}$ rather than the more elaborate classification in [8]. The work in [6] is probably the closest to ours, one can see our work as a superset of the set of combinations mentioned in that work.

5.3 NAT Types as Combinations of Policies

In this section we try to semi-formally summarize the more elaborate classification of NATs known as "BEHAVE-compliant"[8] and craft the notation that we will use in the rest of the paper.

Notation. Let n_a and n_b be NAT gateways. For $i \in \{a, b\}$, Let $P_i = \{p_i, p'_i, p''_i, \ldots\}$ be the set of peers behind n_i. An "endpoint" e is a host-port pair $e = (h, p)$, where $h(e)$ is the host of e and $p(e)$ is its port. Let $V_i = \{v_i, v'_i, v''_i, \ldots\}$ denote the set of all private endpoints of all peers behind n_i and $U_i = \{u_i, u_i', u_i'' \ldots\}$ be the set of public endpoints of n_i. i.e $\forall v \in V_i, h(v) \in P_i$ and $\forall u : U_i, h(u) = n_i$.

When a packet is sent out from a certain private endpoint v_i of a peer p_i behind a gateway n_i, to some public endpoint d, a rule in the NAT table of n_i is created. We define the set of NAT table rules $R_i = \{r_i, r_i', r_i'''\}$ at n_i, the rule records the fact that some public port u_i and some private port v_i are associated, e.g $r_a = (v_a \leftrightarrow u_a)$.

The behavior of a gateway n_i is defined by three policies, namely, port mapping, port filtering and port allocation. We use the notation $f(n_i), m(n_i), a(n_i)$ to denote the respective policies of gateway n_i.

5.3.1 Mapping Policy

The mapping policy is triggered every time a packet is sent from a private endpoint v_i behind the NAT to some external public port d. The role of a mapping policy is deciding whether a new rule will be added or an existing one will be reused. We use the notation:

1. $\overrightarrow{v_i, d} \vDash r_i$ to specify that the sending of a packet from v_i to d resulted in the creation of a new NAT table rule r_i at n_i. That is the binding of a new public port on n_i. However, we say that a rule was created because we care not only about the binding of the port but also the constraints on using this new port.

2. $\overrightarrow{v_i, d} \Rightarrow r_i$ to specify that the sending of the packet reused an already existing rule r_i.

3. $\overrightarrow{v_i, d} \overset{reason}{\not\Rightarrow} r_i$ to specify that the *sending* of the packet did not reuse some r_i in particular because of some *"reason"*.

Irrespective of the mapping policy, whenever a packet is sent from a private port v_i to an arbitrary public destination endpoint d and $\nexists r_i \in R_i$ of the form $r_i = (v_i \leftrightarrow u_i)$, for an arbitrary u_i, the following is true $\overrightarrow{v_i, d} \vDash r_i$. However, if such a mapping exists, the mapping policy would make the reuse decision based on the destination. For all subsequent packets from v_i to d, naturally, $\overrightarrow{v_i, d} \Rightarrow r_i$. However, for any $d' \neq d$, there are 3 different behaviors:

- Endpoint-Independent, $m(n_i) = \text{EI}$:
  $\overrightarrow{v_i, d'} \Rightarrow r_i$, for any d'

- Host-Dependent, $m(n_i) = \text{HD}$:
  $\overrightarrow{v_i, d'} \Rightarrow r_i$, iff $h(d) = h(d')$
  $\overrightarrow{v_i, d'} \vDash r_i'$, iff $h(d) \neq h(d')$, where $r_i' = (v_i \leftrightarrow u_i')$
  and $u_i' \neq u_i$

- Port-Dependent, $m(n_i) = \text{PD}$:
  $\overrightarrow{v_i, d'} \vDash r_i'$

Having introduced the different policies, we decorate the notation of the rule to include the criteria that will be used to decide whether a certain rule will be reused as follows:

$$r_i = \left\{ \begin{array}{lr} \left(v_i \xleftrightarrow[m:v_i \rightarrow *]{} u_i \right) & \text{if } m(n_i) = \text{EI} \\ \left(v_i \xleftrightarrow[m:v_i \rightarrow (h(d),*)]{} u_i \right) & \text{if } m(n_i) = \text{HD} \\ \left(v_i \xleftrightarrow[m:v_i \rightarrow d]{} u_i \right) & \text{if } m(n_i) = \text{PD} \end{array} \right\}$$

Where the syntax $m : x \rightarrow y$ means that the rule will be reused if the source endpoint of the packet is x and the destination is y. The $*$ denotes any endpoint.

Order. We impose the EI < HD < PD according to the increasing level of restrictiveness.

5.3.2 Allocation Policy.

Every time a new r_i is added to R_i, a new public endpoint u_i is bound. This policy allocates $p(u_i)$. That is, the mapping policy decides *when* to bind a new port and the allocation policy decides *which* port should be bound as follows:

1. Port-Preservation, $a(n_i) = \text{PP}$:
   Given $\overrightarrow{v_i, d} \vDash r_i$, where $r_i = (v_i \leftrightarrow u_i)$, it is always the case that: $p(u_i) = p(v_i)$. Naturally, this may cause conflicts if any two p_i and p'_i behind n_i decided to bind private endpoints with a common port.

2. Port Contiguity, $a(n_i) = \text{PC}$:
   Given any two sequentially allocated public endpoints u_i and u'_i it is always the case that: $p(u'_i) = p(u_i) + \Delta$, for some $\Delta = 1, 2, ...$

3. Random, $a(n_i) = \text{RD}$:
   $\forall u_i, p(u_i)$ is allocated at random.

Order. We impose the order PP < PC < RD according to the increasing level of difficulty of handling.

5.3.3 Filtering Policy.

The filtering policy decides whether a packet from the outside world to a public endpoint of a NAT gateway should be forwarded to the corresponding private endpoint. Given an existing rule $r_i = (v_i \leftrightarrow u_i)$ that was created to send a packet from v_i to d, we use the notation:

1. $r_i \Leftarrow \overleftarrow{u_i, s}$ to denote that the *receival* of a packet from the public endpoint s to n_i's public endpoint u_i is permitted by r_i

2. $r_i \overset{reason}{\not\Leftarrow} \overleftarrow{u_i, s}$ to denote that the receival is not permitted because of some "reason".

There are 3 filtering policies with the following conditions for allowing receival:

- Endpoint-Independent, $f(n_i) = \text{EI}$:
  $r_i \Leftarrow \overleftarrow{u_i, s}$, for any s

- Host-Dependent, $f(n_i) = \text{HD}$:
  $r_i \Leftarrow \overleftarrow{u_i, s}$, iff $h(s) = h(d)$

- Port-Dependent, $f(n_i) = \text{PD}$:
  $r_i \Leftarrow \overleftarrow{u_i, s}$, iff $s = d$

We also decorate the rules to include conditions for accepting packets as follows:

$$r_i = \left\{ \begin{array}{ll} \left(v_i \xleftrightarrow{f:u_i \leftarrow *} u_i\right) & \text{if } f(n_i) = \text{EI} \\ \left(v_i \xleftrightarrow{f:u_i \leftarrow (h(d),*)} u_i\right) & \text{if } f(n_i) = \text{HD} \\ \left(v_i \xleftrightarrow{f:u_i \leftarrow d} u_i\right) & \text{if } f(n_i) = \text{PD} \end{array} \right\}$$

Order. We impose the order EI < HD < PD according to the increasing level of restrictiveness.

5.3.4 The Set of NAT Types

Having defined the above policies, the NAT type of a given NAT gateway is simply a matter of listing which behavior is used for each of the policies. We define the set of triplets representing all possible NAT types $\tau = \{(m, a, f) | f, m \in \{\text{EI}, \text{HD}, \text{PD}\}, a \in \{\text{PP}, \text{PC}, \text{RD}\}\}$.

5.4 NAT Type Discovery

Before traversing a NAT gateway, one needs to know its type. STUN [3] is the most-widely used method for accomplishing this and there exists many publicly-available STUN servers that assist in the discovery process. The original STUN algorithm produces a classification withdrawn from the set $\tau_{classic}$. More recently, [6, 8] have re-used the STUN infrastructure to get more detailed information, namely, knowing the filtering and the mapping policies.

Due to space limitations and the fact that our main focus is on traversal strategies, we will not delve into the details of performing the discovery process. However, we just need to clarify that in the spirit of [6, 8], we have expanded the scope of the discovery process to discover information about the allocation policy. With that, our classification is capable of reporting all elements in the set τ.

5.5 NAT Traversal Techniques

We explain our traversal techniques which are an augmented version of the well-known techniques in [1]. A time diagram of the techniques is available in the appendix in Chapter 12.

Basic Assumptions. We assume that there is a Rendez-vous server with public IP referred to by z. The traversal process always starts after: i) two Peers p_a and p_b respectively behind NATs n_a and n_b register themselves at z and have an "out-of-band" communication channel with z, which is in our case a TCP connection initiated by the peer, we refer to all endpoints of z and z itslef by the same symbol; ii) The 2 peers know that they need to communicate and know the other peer's public IP, i.e. the corresponding NAT IP, some peers supply additional information during registration as we will shortly explain in Section 5.7.2; iii) all the policies of p_a, p_b are known to z using a discovery process before any traversal process takes place.

5.6 Simple hole-punching (SHP)

5.6.1 Traversal Process

1. p_a sends from some v_a to z through n_a.

2. n_a creates $r_a = (v_a \leftrightarrow u_a)$ and forwards to z.

3. z receives and consequently knows u_a.

4. z informs p_b about u_a (Out-of-band).

5. p_b sends from some v_b to u_a through n_b.

6. n_b creates $r_b = (v_b \leftrightarrow u_b)$ and forwards to u_a.

7. n_a receives, if the filtering allows, forwards to v_a

8. p_a sends from v_a to u_b through n_a, if the mapping allows, r_a is reused. Otherwise, r'_a will be created and sending will occur from some other public endpoint $u'_a \neq u_a$

9. n_b receives, if the filtering allows, forwards to v_b

5.6.2 SHP Feasibility

Theorem 5.6.1. *Simple hole punching is feasible for establishing direct communication between two peers p_a and p_b respectively behind n_a and n_b if $\exists n_x \in \{n_a, n_b\}$ s.th. $f(n_x) =$* EI*, and either $m(n_x) =$* EI *or $m(n_x) >$* EI *and $f(n_{x' \neq x}) <$* PD.

Proof. We consider the most restrictive case where $f(n_a) = f(n_b) = m(n_a) = m(n_b) = \textsc{pd}$ and $a(n_a) = a(n_b) = \textsc{rd}$ and show the minimum relaxations that we need to do for SHP to work. By looking at the steps in section 5.6, and considering all the very restrictive mapping and filtering on both sides, we can see that after steps 5 and 6, r_a and r_b will be as follows:
$r_a = \left(v_a \xleftrightarrow[m:v_a \rightarrow u_z]{f:u_a \leftarrow u_z} u_a\right), r_b = \left(v_b \xleftrightarrow[m:v_b \rightarrow u_a]{f:u_b \leftarrow u_a} u_b\right)$
Which will cause the following problems:

In step 7: $r_a \overset{u_b \neq u_z}{\not\Leftarrow} \overleftarrow{u_b, u_a}$ and there is nothing that we can relax at n_b which can help. Instead, we have to relax the filtering at p_a to indulge receiving on u_a from u_b while it was initially opened for receiving from u_z. i.e, r_a has to tolerate host change which is not satisfied by PD nor HD filtering, therefore $f(n_a) = \textsc{ei}$ is necessary, resulting into
$r_a = \left(v_a \xleftrightarrow[m:v_a \rightarrow u_z]{f:u_a \leftarrow *} u_a\right)$
In step 8: $\overrightarrow{v_a, u_b} \overset{u_b \neq u_z}{\not\Rightarrow} r_a$ and $\overrightarrow{v_a, u_b} \vDash r'_a$ where $r'_a = \left(v_a \xleftrightarrow[m:v_a \rightarrow u_b]{f:u'_a \leftarrow *} u'_a\right)$. Consequently, $r_b \overset{u_a \neq u'_a}{\not\Leftarrow} \overleftarrow{u'_a, u_b}$. To solve this, we have two solutions, the first is to let the mapping reuse r_a and not create r'_a which needs relaxing $m(n_a)$ to be EI, in which case we can keep $f(n_b)$ as restrictive. The second solution is to keep n_a as restrictive and relax $f(n_b)$ to tolerate receiving from u'_a. In the second solution, there is a minor subtlety that needs to he handled, where p_b has to be careful to keep sending to p_a on u_a despite the fact that it is receiving from u'_a. Similarly p_a should always send to p_b on u_b despite the fact it is receiving from u'_b. That is an asymmetry that is not in general needed.

□

5.6.3 Coverage of SHP

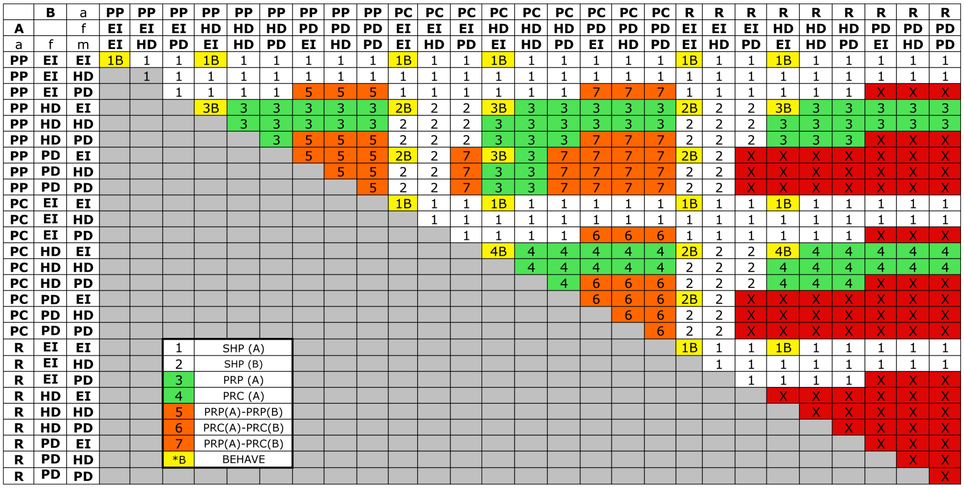
Since $|\tau| = 27$ types, we have a $\frac{27 \times 28}{2} = 378$ distinct combinations of NAT types of two peers. Using Theorem 5.6.1, we find that 186 combinations, i.e. 49.2% of the total number of possible ones are traversable using the Simple Hole Punching approach. That said, this high coverage is totally orthogonal to how often one is likely to encounter combinations in the covered set in practice, which we discuss in our evaluation (Section 5.9.1). Traversable SHP combinations are shown in Figure 5.1 with label SHP(*).

To cover the rest of the cases, we use port prediction which enables a peer to punch a hole by sending to the opposite peer instead of z, which makes it possible to tolerate more restrictive filtering and mapping policies, as explained below.

5.7 Prediction

5.7.1 Prediction using Contiguity (PRC)

The traversal process consists in the following steps:



	B	a	PP	PP	PP	PP	PP	PP	PP	PP	PP	PC	PC	PC	PC	PC	PC	PC	PC	PC	R	R	R	R	R	R	R	R	R
A		f	EI	EI	EI	HD	HD	HD	PD	PD	PD	EI	EI	EI	HD	HD	HD	PD	PD	PD	EI	EI	EI	HD	HD	HD	PD	PD	PD
a	f	m	EI	HD	PD	EI	HD	PD	EI	HD	PD	EI	HD	PD	EI	HD	PD	EI	HD	PD	EI	HD	PD	EI	HD	PD	EI	HD	PD
PP	EI	EI	1B	1	1	1B	1	1	1	1	1	1B	1	1	1B	1	1	1	1	1	1B	1	1	1B	1	1	1	1	1
PP	EI	HD		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
PP	EI	PD			1	1	1	1	5	5	5	1	1	1	1	1	1	7	7	7	1	1	1	1	1	1	X	X	X
PP	HD	EI				3B	3	3	3	3	3	2B	2	2	3B	3	3	3	3	3	2B	2	2	3B	3	3	3	3	3
PP	HD	HD					3	3	3	3	3	2	2	2	3	3	3	3	3	3	2	2	2	3	3	3	3	3	3
PP	HD	PD						3	5	5	5	2	2	2	3	3	3	7	7	7	2	2	2	3	3	3	X	X	X
PP	PD	EI							5	5	5	2B	2	7	3B	3	7	7	7	7	2B	2	X	X	X	X	X	X	X
PP	PD	HD								5	5	2	2	7	3	3	7	7	7	7	2	2	X	X	X	X	X	X	X
PP	PD	PD									5	2	2	7	3	3	7	7	7	7	2	2	X	X	X	X	X	X	X
PC	EI	EI										1B	1	1	1B	1	1	1	1	1	1B	1	1	1B	1	1	1	1	1
PC	EI	HD											1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
PC	EI	PD												1	1	1	1	6	6	6	1	1	1	1	1	1	X	X	X
PC	HD	EI													4B	4	4	4	4	4	2B	2	2	4B	4	4	4	4	4
PC	HD	HD														4	4	4	4	4	2	2	2	4	4	4	4	4	4
PC	HD	PD															4	6	6	6	2	2	2	4	4	4	X	X	X
PC	PD	EI																6	6	6	2B	2	X	X	X	X	X	X	X
PC	PD	HD																	6	6	2	2	X	X	X	X	X	X	X
PC	PD	PD																		6	2	2	X	X	X	X	X	X	X
R	EI	EI																			1B	1	1	1B	1	1	1	1	1
R	EI	HD																				1	1	1	1	1	1	1	1
R	EI	PD																					1	1	1	1	X	X	X
R	HD	EI																						X	X	X	X	X	X
R	HD	HD																							X	X	X	X	X
R	HD	PD																								X	X	X	X
R	PD	EI																									X	X	X
R	PD	HD																										X	X
R	PD	PD																											X

1	SHP (A)
2	SHP (B)
3	PRP (A)
4	PRC (A)
5	PRP(A)-PRP(B)
6	PRC(A)-PRC(B)
7	PRP(A)-PRC(B)
*B	BEHAVE

Figure 5.1: All possible distinct NAT types combinations for two peers a and b with the technique needed to traverse the combination and X for un-traversable combinations. SHP(*), PRC(*) and PRP(*) stand respectively for Simple Hole Punching, Port Prediction using Contiguity and Port Prediction using Preservation. Combinations of NAT behaviors mandated by RFC 4787 are identified by the label BEHAVE in the table's legend.

1. p_a sends two consecutive messages:
   - from some v_a to z through n_a
   - from v_a to u_b^{dum}, an arbitrary endpoint of n_b
2. n_a creates the following two rules:
   - $r'_a = (v_a \leftrightarrow u'_a)$ and forwards to z.
   - $r_a = (v_a \leftrightarrow u_a)$ and forwards to u_b^{dum}. Actually, the whole point of sending u_b^{dum} is to open u_a by sending to n_b but be able to predict it at z.
3. The messages are received as follows:
   a) z receives and consequently knows u'_a and additionally predicts $u_a = u'_a + \Delta$ where Δ is known during the discovery process.
   b) n_b drops the message since no endpoint u_b^{dum} was ever bound.
4. z informs p_b about u_a (Out-of-Band).
5. Steps 5 – 9 follow the same scheme as in simple hole punching.

Port scanning. The process is susceptible to failure if another peer p'_a happens by coincidence to send a packet between the two consecutive packets. For that, a technique called port scanning [6] is used such that when p_b tries to connect to u_a, p_b will try $u_a + \Delta, u_a + 2\Delta, u_a + 3\Delta$, etc.. until a reply is received. Some gateways might identify this as a malicious UDP port scan and block it as is the case in some corporate firewalls. Port scanning might be used only when p_b connecting to p_a where $a(n_a) = PC$ has $m(n_b) < PD$, as shown by[6].

5.7.2 Prediction using Preservation (PRP)

Another technique is to exploit the port-preservation allocation policy. However, to do that, we assume that when a peer with port-preservation policy registers at z, the peer supplies a pool of free candidate ports to z. The main point here is to avoid conflicts with ports of other peers behind the same NAT. The rendez-vous server z is stateful regarding which ports are bound by each NAT and chooses from the pool of the ports supplied by the peer a port which is not already bound.

1. z chooses some arbitrary port ρ and tells p_a (Out-of-Band) to bind ρ
2. p_a sends from v_a where $p(v_a) = \rho$ to u_b^{dum} through n_a.
3. n_a creates a new rule $r_a = (v_a \leftrightarrow u_a)u_b^{dum}$ and forwards to u_b^{dum} and since $a(p_a) = \text{PP}$, $p(u_a) = p(v_a) = \rho$.
4. z informs p_b about u_a (Out-of-Band).
5. Steps 5-9 follow the same scheme as in SHP.

Note that the process is shorter than prediction by contiguity and z chooses the port for the peer behind NAT instead of the NAT of the peer deciding it and z observing it. However, for the sake of reasoning, the two are equivalent because what matters is what happens after the opposite peer learns about the punched port irrespective of how the port was predicted.

5.7.3 Prediction-on-a-Single-Side Feasibility

Theorem 5.7.1. *Prediction using contiguity or preservation on a single side is feasible for establishing direct communication between two peers p_a and p_b respectively behind n_a and n_b if:*

- *Condition 1:* $\exists n_x \in \{n_a, n_b\}$ *s.th.* $a(n_x) < \text{RD}$ *and* $f(n_x) < \text{PD}$
- *Condition 2: Either* $m(n_x) < \text{PD}$ *or* $m(n_x) = \text{PD}$ *and* $f(n_{x' \neq x}) < \text{PD}$.

Proof. Similar to theorem 5.6.1, we start with the most restrictive policies and we relax until prediction is feasible. The allocation policy of the side to be predicted (n_a in Section 5.7.2,5.7.1) can not be random, because the whole idea of prediction relies on a predictable allocation policy, thus the needed relaxation is $a(n_a) < \text{RD}$.

In both prediction techniques, the dummy packet from p_a punches a hole by sending to p_b, in contrast to SHP which punches by sending to z. nevertheless, it is sent to a dummy port of p_b. After steps 5, 6:

$$r_a = \left(v_a \xleftrightarrow[m:v_a \rightarrow u_p^{dum}]{f:u_a \leftarrow u_b^{dum}} u_a \right), r_b = \left(v_b \xleftrightarrow[m:v_b \rightarrow u_a]{f:u_b \leftarrow u_a} u_b \right)$$

In step 7: $r_a \overset{u_b \neq u_b^{dum}}{\not\Leftarrow} \overleftarrow{u_b, u_a}$, we have to relax the filtering at p_a to indulge the port difference from u_b, but we tolerate host sensitivity. The needed relaxation is: $f(n_a) < \text{PD}$ resulting into:

$$r_a = \left(v_a \xleftrightarrow[m:v_a \rightarrow u_b^{dum}]{f:u_a \leftarrow (n_b, *)} u_a \right)$$

In step 8: the reasoning about relaxing the mapping on p_a or the filtering of p_b is identical to Theorem 5.6.1 except that host-sensitivity is tolerable and thus either $m(n_a) < \text{PD}$ or is kept $m(n_a) = \text{PD}$ and in that case, the needed relaxation is $f(n_b) < \text{PD}$. □

5.7.4 Coverage of PRP & PRC

PRP and PRC together cover another 18% of the combinations. That said, we can say that PRP is as good as SHP in terms of traversal time and success rate (see Section 5.9), which means in addition to the cases where PRP on a single side is used in Figure 5.1, we can also use PRP instead of SHP when the allocation policy is port preservation.

5.8 Interleaved Prediction on Two Sides

. The remaining combinations are these not covered by SHP nor prediction. The final stretch to go is to do simultaneous prediction on both sides. However, it is a seemingly tricky deadlock situation because every peer needs to know the port that will be opened by the other peer without the other peer sending anything. Which we solve as follows.

Interleaved PRP-PRP. In this case actually double prediction is very simple because the rendez-vouz server can pick a port for each side and instruct the involved peers to simultaneously bind it and start the communication process.

Interleaved PRP-PRC This case is also easily solvable thanks to preservation. Because z can inform the peer with a port contiguity allocation policy about the specific endpoint of the opposite peer. The latter in turn will run a port prediction process using the obtained endpoint in the second consecutive message.

Interleaved PRC-PRC This one is the trickiest and it needs a small modification in the way prediction by contiguity is done. The idea is that the two consecutive packets, the first to z and the second to the opposite peer can not be sent after each other immediately. Instead, both peers are commanded by z to send a packet to z itself. From that, z deduces the ports that will be opened on each side in the future and sends to both peers informing them about the opposite peer's predicted endpoint. Both peers in their turn send a punching packet to each other. The problem with this scheme is that there is more time between the consecutive packets which makes it more susceptible to the possibility of another peer behind any of the NATs sending a packet in in between. Like

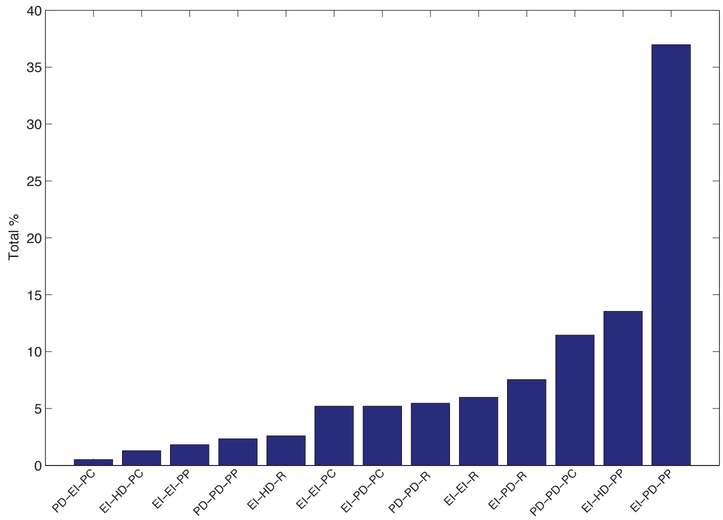


Figure 5.2: Distribution of encountered NAT types in τ as (m, f, a)

the case in single PRC, port scanning is the only resort, but in general this combination has lower success rate compared to single PRC (see Section5.9).

For our reasoning, we will work on the last one (PRC-PRC), since it is a general harder case of the first two.

5.8.1 Traversal Process

1. z tells p_a & p_b to start prediction (Out-of-Band)

2. p_a & p_b both send to z through n_a & n_b respectively resulting in the new rules $r'_a = (v_a \leftrightarrow u'_a), r'_b = (v_b \leftrightarrow u'_b)$

3. z receives from p_a & p_b, observing u'_a & u'_b and deducing $u_a = u'_a + \Delta$ & $u_b = u'_b + \Delta$

4. z informs p_a & p_b about u_b & u_a respectively (Out-of-Band)

5. p_a sends to u_b through n_a and p_b sends to u_a through n_b

6. n_b receives and forwards to v_b and n_a receives and forwards to v_a

A race condition where step 6 for one of the peers happens before the opposite peer starts to run step 5 can take place resulting into a packet drop. However, the dropped packet opens the hole for the opposite peer, and retrying sending is enough to take care of this issue.

Mapping	EI	HD	PD
	80.21%	0%	19.79%
Filtering	EI	HD	PD
	13.54%	17.45%	69.01%
Allocation	PP	PC	RD
	54.69%	23.7%	21.61%

Table 5.1: Distribution of encountered NAT policies

5.8.2 Interleaved Prediction Feasibility

Theorem 5.8.1. *Interleaved Prediction is feasible for establishing direct communication between two peers p_a and p_b respectively behind n_a and n_b if both $a(n_b)$ and $a(n_b)$ are $< RD$*

Proof. Similar to theorem 5.6.1, we start with the most restrictive policies and we relax until prediction is feasible. Since we need to predict both sides we need $a(n_a) <$ RD & $a(n_b) <$ RD. After step 5 in Section 5.8.1, we have:

$r_a = \left(v_a \xleftrightarrow[m:v_a \rightarrow u_b]{f:u_a \leftarrow u_b} u_a \right), r_b = \left(v_b \xleftrightarrow[m:v_b \rightarrow u_a]{f:u_b \leftarrow u_a} u_b \right)$

In step 6, we have $r_a \Leftarrow \overleftarrow{u_a, u_b}$ and $r_b \Leftarrow \overleftarrow{u_b, u_a}$ without the need for any relaxations on the filtering nor the mapping of either sides. □

5.8.3 Interleaved Prediction Coverage

The interleaved prediction covers another 11.9% of the combinations, namely the ones shown in Figure 5.1 leaving 20.6% of the cases untraversable. That is, approximately 79.4% of all NAT type combinations are traversable and for each combination, we know which technique to use. The more important thing is that not all of them have the same likelihood of being encountered which we discuss in the next section. That said, it worth mentioning that there is a technique in [9] which performs a brute-force search on all possible ports after reducing the search space using the birthday paradox, which we ignored due to low success probability, high traffic and long time requirements.

5.9 Evaluation

Apart from the reasoning above, we have done a sanity check on our logic using our emulation platform [10]. That is, we wrote our own NAT boxes, which behave according to the semantics defined in Section 5.3. We also implemented the Rendez-Vous server and the nodes that are capable of performing all the traversal techniques in Section 5.5. For each case in Figure 5.1, we ran the suggested traversal technique and we made sure direct communication is indeed achievable. Real-life evaluation was needed to gain insights on other aspects like probability of encountering a given type, success rates of traversal techniques and time needed for the traversal process to complete.

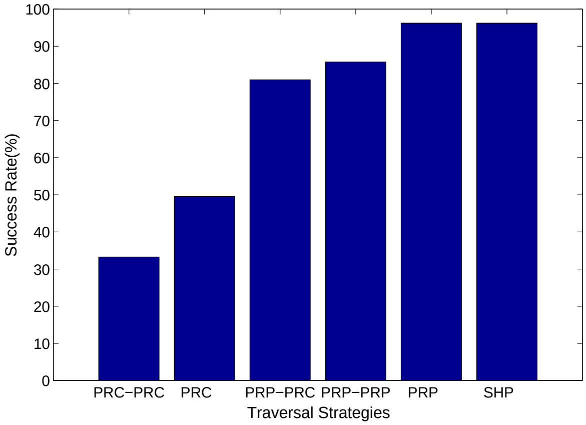


Figure 5.3: Success rate of each technique averaged over all applicable combinations.

5.9.1 Distribution of Types

We wanted to know how likely is it to encounter each of the types in τ. We have collected cumulative results for peers who have joined our network over time. As shown in Figure 5.2: i) we encountered 13 out of the 27 possible types; ii) we found that ($m = \text{EI}, f = \text{PD}, a = \text{PP}$) is a rather popular type (approx. 37%) of all encountered types, which is fortunate because port preservation is quite friendly to deal with and it is with a very relaxed mapping; iii) about 11% are the worst kind to encounter, because when two peers of this type need to talk, interleaved prediction is needed with a shaky success probability.

5.9.2 Adoption of BEHAVE RFC

By looking at each policy alone, we can see to what extent the recommendations of the BEHAVE RFC [8] ($f = \text{EI}/\text{HD}, m = \text{EI}$) are adopted. As shown in Table 5.1, for filtering, the majority are adopting the policy discouraged by the RFC, while for mapping the majority were following the recommendation. For allocation, the RFC did not make any specific relevant recommendation. The percentage of NATs following *both* recommendations was 30%.

5.9.3 Success Rate

Given the set of peers present in the network at one point in time, we conduct a connectivity test where all peers try to connect to each other. We group the result by traversal techniques, e.g. SHP is applicable for 186 combinations, so we average the success rate over all combinations and the whole process is repeated a number of times, we have found (Figure 5.3) as expected that SHP is rather dependable as it succeeds 96% of the

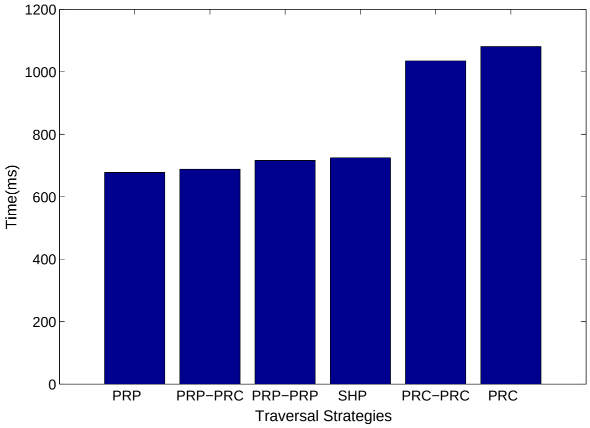


Figure 5.4: Time taken (in msec) for the traversal process to complete.

time. We also found that PRP is as good as SHP, which is quite positive given that we found that the probability of occurrence of preservation is quite high in the last section. Interleaved PRP-PRP is also rather good with slightly worse success rate. The three remaining techniques involving PRC in a way or the other are causing the success rate to drop significantly especially for PRC-PRC mainly because of the additional delay for interleaving.

5.9.4 Time to traverse

When it comes to the time needed for the traversal process to complete (Figure 5.4), we find two main classes, SHP and PRP in one class and PRC in another class, even when we do PRC-PRP, it is faster than PRC alone because the number of messages is less.

5.10 Conclusion & Future Work

In this paper, we have presented our experience with trying to find a comprehensive analysis of what combinations of NAT types are traversable. We have shown that using a semi-formal reasoning that covers all cases and we provided a slightly augmented versions of the well-known traversal techniques and shown which ones are applicable for which combinations.We have shown that about 80% of all possible combinations are traversable.

Using our deployment base for P2P live streaming, we have shown that only 50% fo all possible types are encounterable. We have also reported our findings on the success probability and time of traversing the different combinations.

For future work: *a*) Modeling: we would like to enrich the model to make it capture real-life aspects like expiration of NAT rules, multiple levels of NAT, subtleties of conflicts between many peers behind the same NAT, NATs that use different policies in different situations, and support for uPnP and TCP; *b*) Real-life Evaluation: more insight into the trade-off between success probability and timing, preventing the techniques as being identified as malicious actions in some corporate firewalls; *c*) Dissemination: releasing our library and simulator as open-source for third-party improvement and evaluation.

5.11 Acknowledgments

We would like to thank all members of Peerialism's development team for the help and collaboration on the implementation of our NAT traversal techniques, in particular Magnus Hedbeck for his patience and valuable feedback. The anonymous reviewers of ICCCN have provided a really inspiring set of comments that helped us to improve the quality of this paper.

5.12 References

[1] Bryan Ford, Pyda Srisuresh, and Dan Kegel. Peer-to-peer communication across network address translators. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 13–13, Berkeley, CA, USA, 2005. USENIX Association.

[2] P. Srisuresh, B. Ford, and D. Kegel. State of Peer-to-Peer (P2P) Communication across Network Address Translators (NATs). RFC 5128 (Informational), March 2008. URL `http://www.ietf.org/rfc/rfc5128.txt`.

[3] J. Rosenberg, J. Weinberger, C. Huitema, and R. Mahy. STUN - Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs). RFC 3489 (Proposed Standard), March 2003. URL `http://www.ietf.org/rfc/rfc3489.txt`. Obsoleted by RFC 5389.

[4] C. Huitema J. Rosenberg, R. Mahy. Traversal using relays around nat (turn): Relay extensions to session traversal utilities for nat (stun). Internet draft, November 2008. URL `http://tools.ietf.org/html/draft-ietf-behave-turn-14`.

[5] J. Rosenberg. Interactive connectivity establishment (ice): A protocol for network address translator (nat) traversal for offer/answer protocols. Internet draft, October 2007. URL `http://tools.ietf.org/html/draft-ietf-mmusic-ice-19`.

[6] Y. Takeda. Symmetric nat traversal using stun. Internet draft, June 2003. URL `http://tools.ietf.org/html/draft-takeda-symmetric-nat-traversal-00`.

[7] D. Thaler. Teredo extensions. Internet draft, March 2009. URL `http://tools.ietf.org/html/draft-thaler-v6ops-teredo-extensions-03`.

[8] F. Audet and C. Jennings. Network Address Translation (NAT) Behavioral Requirements for Unicast UDP. RFC 4787 (Best Current Practice), January 2007. URL `http://www.ietf.org/rfc/rfc4787.txt`.

[9] Andrew Biggadike, Daniel Ferullo, Geoffrey Wilson, and Adrian Perrig. NATBLASTER: Establishing TCP connections between hosts behind NATs. In *Proceedings of ACM SIGCOMM ASIA Workshop*, April 2005.

[10] Roberto Roverso, Mohammed Al-Aggan, Amgad Naiem, Andreas Dahlstrom, Sameh El-Ansary, Mohammed El-Beltagy, and Seif Haridi. Myp2pworld: Highly reproducible application-level emulation of p2p systems. In *Decentralized Self Management for Grid, P2P, User Communities workshop, SASO 2008*, 2008.

CHAPTER 6

DTL: Dynamic Transport Library for Peer-To-Peer Applications

DTL: Dynamic Transport Library for Peer-To-Peer Applications

Riccardo Reale[1], Roberto Roverso[1,2], Sameh El-Ansary[1] and Seif Haridi[2]

[1]Peerialism AB
Stockholm, Sweden
[roberto,sameh]@peerialism.com

[2]KTH - Royal Institute of Technology
Stockholm, Sweden
[haridi]@kth.se

Abstract

This paper presents the design and implementation of the Dynamic Transport Library (DTL), a UDP-based reliable transport library, initially designed for - but not limited to - peer-to-peer applications. DTL combines many features not simultaneously offered by any other transport library including: i) Wide scope of congestion control levels starting from less-than-best-effort to high-priority, ii) Prioritization of traffic relative to other non-DTL traffic, iii) Prioritization of traffic between DTL connections, iv) NAT-friendliness, v) Portability, and vi) Application level implementation. Moreover, DTL has a novel feature, namely, the ability to change the level of aggressiveness of a certain connection at run-time. All the features of the DTL were validated using a controlled environment as well as the Planet Lab testbed.

6.1 Introduction

Looking at the rich and diverse requirements of applications in a quickly emerging field like P2P computing, we find that these needs have driven innovation in Internet transport protocols. In the past few years, it has been more and more common that a P2P application develops its own congestion control algorithm, because using out-of-the box TCP congestion control did not suffice. To name a few examples, for a voice and video conferencing application like Skype, a steady low-jitter flow of packets is required. On top of that, due to its real-time nature, Skype traffic must benefit from higher priority with respect to any other application's transfer. Thus, Skype developed an application-level proprietary congestion control algorithm [1] known to be very aggressive towards other applications. On the other end of the spectrum, a content distribution application like Bittorrent, started initially by using TCP but then switched to LEDBAT, in order to be polite as much as possible towards other applications, while saturating the link capacity. Politeness was critical to eliminate the reputation of Bittorrent as a protocol which totally hogs the bandwidth and makes all other applications starve. Between extreme politeness and aggressiveness, many other applications can settle for traditional fair contention over the bandwidth.

The collective state-of-the-art in congestion control algorithms has already addressed most of the diverse needs of P2P applications. The serious shortcoming is the that the best known techniques are scattered around in different libraries. This makes it rather hard for anyone developing a new P2P application to benefit from the progress in the field. This scattering of ideas in addition to some practical and algorithmic issues that we faced while studying other algorithms like LEDBAT, MulTCP and MulTFRC motivated our work on the DTL transport library. The idea is to have one single library that can practically serve as single one-stop-shop for any P2P application.

The following is a list of the target requirements that we realize in DTL:

Priority Levels Supported. The library has to support all levels of traffic prioritization that already exist in the congestion control state-of-the-art.

Inter-Protocol Prioritization. The library has to give control to the application on how polite or aggressive it wants to be against other applications.

Intra-Protocol Prioritization. The library has to give control to the application on how polite or aggressive each individual connection is with respect to other connection within the application. A P2P storage application might need, for instance, to have low-priority background transfers for backup purposes and high-priority transfer to fetch a file that has just been shared by another peer.

In addition to that, there are some features that have to be there for practical purposes:

NAT-Friendliness. The library has to be based on UDP. This makes it easier to circumvent NAT constraints as UDP NAT traversal procedures are known to be more effective than TCP ones [2][3].

Application-level implementation. The library has to be implemented in user space, because for all practical purposes, a kernel-level implementation would hinder any successful wide use. A feature lacking from a rather flexible congestion control library like MulTCP.

Portability. The library shall be available on all platforms, again, that if we want it to be widely-adopted.

Run-Time Dynamic Prioritization. One final feature which we have not previously found in any other library is the ability to not only specify different priorities for different connection but to be able to change the priority of a particular connection at run-time. The need for this feature has arisen, when our team was working on a P2P video streaming application and we needed to tweak the priority of incoming transfers to achieve the necessary playback rate; this by progressively increasing the aggressiveness towards other traffic in the network. An additional advantage of on-the-fly priority

tuning is that it prevents disruptions of existing connections and avoids the need of connection re-establishments for transitioning from one priority level to the other or from a congestion control algorithm to the other. In fact, it is known that connection establishments procedures are usually very costly, on the order of several seconds [2], due to expensive peer-to-peer NAT Traversal and authentication procedures.
Finally, the library should support TCP-like reliability and flow control for ease of use.

In this paper, we present how we achieved meeting all aforementioned requirements in the software we call *Dynamic Transport Library* or *DTL*. The paper is organized as follows: in Section 6.2 we present the state of the art which constitutes the starting point of our effort. With respect to that, we detail the contribution of our work in Section 6.3. In Section 6.4 we explain the design and implementation of the library, while in Section 6.5, we present our library's evaluation results. We then conclude with some final considerations and future work in Section 6.6.

6.2 Related Work

LEDBAT is widely accepted as an effective solution to provide a less-than-best-effort data transfer service. Initially implemented in the μTorrent BitTorrent client and now separately under discussion as an IETF draft [4], LEDBAT is a delay-based congestion control mechanism which aims at saturating the bottleneck link while throttling back transfers in the presence of flows created by other applications, such as games or VoIP applications. The yielding procedure is engineered to avoid disruptions to other traffic in the network, and it is based on the assumption that growing one-way delays are symptoms of network congestion. With respect to classical TCP, this allows for earlier congestion detection. LEDBAT was inspired by previous efforts like TCP-Nice [5] and TCP-LP [6] which are based on the same idea.

Our initial goal was to test LEDBAT for background data transfers and later try to tweak its parameters to obtain increased aggressiveness and thus higher transfer priority. Although positively impressed by the ability of LEDBAT to minimize the latency introduced on the network, we found out that tweaking gain and target delay metrics does not lead to a corresponding degree of aggressiveness, as also stated by [7]. As a consequence, we started to investigate other possible solutions for obtaining tunable aggressiveness transfers. MulTCP [8] provides a mechanism for changing the increment and decrement parameters of the normal TCP's AIMD [9] algorithm to emulate the behavior of a fixed number N of TCP flows in a single transfer. The idea of multiple flow virtualization is very promising, however the protocol has been experimented only as a kernel module and it is unclear how an application-level implementation would perform.

MulTFRC [10] extends a previous protocol called TCP Friendly Rate Control (TFRC) [11] to achieve variable transfer aggressiveness. The core idea of TFRC is to achieve TCP-friendliness by explicitly calculating an equation which approximates the steady-state throughput of TCP. MulTFRC modifies the TFRC congestion equation, resulting in a tunable version of the rate control which emulates the behavior of N TFRC flows

	Inter-protocol prioritization	Intra-protocol prioritization	Application level	Portable	NAT-friendly	Runtime Dynamic Tuning
LEDBAT	L-T-B-E		X		UDP	
MulTCP	L-H, CONT	X			TCP	
MulTFRC	L-H, CONT	X	X		UDP	
DTL	L-T-B-E & L-H, CONT	X	X	X	UDP	X

Table 6.1: Comparison between protocols. L.T.B.E. = Less-Than-Best-Effort, CONT = Continuous Range, L-H. = Low to High

while maintaining a smooth sending rate. It has been shown that MulTFRC gives better bandwidth utilization than other protocols, such as the Coordination Protocol (CP), by better approximating the steady-state throughput of N virtualized TCP flows [10]. MulTFRC differs from MulTCP in the way that it provides a smoother sending rate, making it particularly suitable for multimedia applications. Unfortunately, MulTFRC does not allow for the runtime modification of the N parameter and thus, it is not suitable for our goals [12].

Since MulTCP and MulTFRC are both packet-based congestion control algorithms, if configured to simulate less than one TCP flow, they are likely to be more aggressive than other less-than best effort alternatives. For simulating multiple flows instead, both MulTCP and MulTFRC perform reasonably well with values up to $N = 10$, but only MulTFRC increases linearly for higher values, as mentioned in [10].

6.3 Contribution

In this paper, we present the design and implementation of an application-level library which provides TCP-like reliability and flow control while implementing variable and configurable on-the-fly traffic prioritization through different congestion control techniques. To the best of our knowledge, this library is first of its kind given the aforementioned characteristics.

The library implements two state of the art congestion controls algorithms: LEDBAT and MulTCP. The two mechanisms have been combined to achieve traffic prioritization which cover a range of levels which start from *less-than-best-effort*, where transfers totally yield to other traffic, up to *high*, where transfers try to reclaim bandwidth from both other intra- and extra-protocol transfers. The priority level can be adjusted using a unique configurable parameter named $priority$. The parameter can be changed at runtime without causing the flow of data to be disrupted and without the need of connection re-establishments. For presentation's purpose, the traffic levels are classified in two operational modes, according to which congestion control algorithm is used:

$$Mode = \begin{cases} Polite, & if priority = 0 (LEDBAT) \\ Variable, & if priority > 0 (MULTCP) \end{cases} \tag{6.1}$$

As a further contribution, the library is implemented in Java as an attempt to make the software portable between different platforms. We are unaware of any other effort to fully implement LEDBAT, MulTCP or any relevant congestion control mechanism for that matter on an application-level library in Java. A feature-wise comparison of the state-of-the art against DTL is shown in Table 6.1.

6.4 Dynamic Transport Library (DTL)

We implemented *DTL* using Java NIO over UDP. We drew inspiration for its design from TCP with SACK (Selective Acknowledgment Options). For this reason, we provide an application level TCP-like header in order to enable reliability, congestion control and flow control over UDP in the same way TCP does. In our implementation, the header is appended to each datagram together with the data to be transmitted, and it is encoded/decoded by the sender and receiver modules at the respective ends of the connection. The base header carries the receiver's advertised window, the sequence number and the acknowledge number. The packet size is dynamically chosen. By default, *DTL* uses large packet sizes of 1500 bytes, while at slow rates the size can be adjusted down to 300 bytes.

The sender's module which initiates a transfer maintains three variables: (1) the congestion window $cwnd$, meant as a sender-side limit, (2) the receiver's advertised window, as receiver-side limit, and(3) the slow start threshold.

End-to-end flow control, used to avoid the sender transmitting data too quickly to a possible slow receiver, is obtained using a sliding window buffer. In every acknowledgement packet, the receiver advertises its receive window as the amount of data that it is able to buffer for the current connection. At each point in time, the sender is allowed to transmit only a number of bytes defined as follows:

$$allowed = min(rcv\_win, cwnd) - inflight \tag{6.2}$$

where rcv_win is the advertised receive window, $inflight$ is the amount of data in transit on the network and not yet acknowledged, while $cwnd$ is the congestion window size.

Reliability is ensured by tagging each packet with a sequence number which corresponds to the the amount of bytes sent up to that point in time. Using the sequence number, the receiver is able to understand the ordering of packets and identify losses. For each received packet, the receiver sends back an acknowledgement(ack) with the amount of bytes it has received up till that point. The sender can detect packet loss in two ways: when a packet times out, or when the receiver notifies the sender with a special format's selective acknowledgement. The packet timeout value is correlated with the estimated RTT. The latter is updated the same way TCP does: using the Karn/Partridge algorithm [13].

The Selective Acknowledgement is generated by the receiver after more than three out-of-order packets, and contains the information about all successfully received segments. Consequently, the sender can retransmit only the segments which have actually been lost. The library's behaviour in case of packet loss or ack reception is defined at the server-side by the chosen congestion control mechanism.

This is because, while in our library flow control and reliability designs are both directly derived from TCP, we provided a different implementation of congestion control according to the transfer prioritization policy which needs to be enforced.

In general terms, in order to control the aggressiveness of the flow, we provide the applications with *priority* parameter for each socket, intended as a positive floating point number. The correlation between the priority parameter and the service mode has been previously defined in Equation 6.1.

In the $polite$ mode, we make use of the LEDBAT algorithm. In the $variable$ mode instead, we provide our implementation of MulTCP, parametrized with a priority value corresponding to the number $N == priority$ of flows to virtualize. The choice of MulTCP is motivated by the fact that, while MulTFRC might provide a small improvement in transfer stability and bandwidth utilization [12], the number of virtualized flows cannot be changed at runtime. In the following two sections, we will detail our implementation of both $polite$ and $variable$ modes, describing the implementation of the congestion control algorithms.

6.4.1 Polite Mode

Although LEDBAT is not the only congestion control providing less-than-best-effort transfers, it is the only one which actually tries to control the latency introduced on the network by the transfer itself. Other similar algorithms, like TCP-Nice [5] and TCP-LP [6] use the increasing delay as an indicator of imminent congestion as LEDBAT does, but they try to react in a more conservative manner, or simply by backing off earlier than TCP. LEDBAT instead opts to keep the delay under a certain threshold, reason for which it is able to achieve a yielding factor that is higher than other congestion controls, as explained in [7].

Since the main congestion indicator in LEDBAT is the one-way delay variation, in order to react earlier than TCP to congestion events, we added a timestamp and delay header fields to compute its value. For every packet sent, the sender appends to the packet its own timestamp, while the receiver sends back an acknowledgement containing that same timestamp (used also for better RTT estimation) and the measured one-way delay. The sender's congestion control module maintains a list of the minimum one-way delays observed every minute in a $BASE\_HISTORY$ queue. The smallest delay D_{min} is used to infer the amount of delay due to queuing, as we assume it represents the physical delay of the connection before any congestion happened. The mechanism of using only "recent" measurements, letting the old ones expire, results in a faster response to changes in the base delay. Consequently it also allows to correct possible errors in the measurements caused by clock skewness between sender and receiver. The last measured one-way delays are stored in a $NOISE\_FILTER$ queue. The lowest value in the queue is used to compute the queuing delay. In our implementation, the $BASE\_HISTORY$ memory time is set to 13 minutes while the $NOISE\_FILTER$ queue contains the 3 last measured delays.

The key element in the LEDBAT congestion control algorithm lies in comparing the estimated queuing delay against a fixed target delay value τ, considered as the maximum amount of delay that a flow is allowed to introduce in the queue of the bottleneck buffer. The difference $\Delta(t)$ between the queuing delay and the target is used to proportionally increase or decrease the congestion window. The original LEDBAT linear controller is defined as follows:

$$\Delta(t) = \tau - (D(t) - D_{min}) \tag{6.3}$$

$$cwnd(t+1) = cwnd(t) + \gamma\Delta(t)/cwnd(t) \tag{6.4}$$

where γ is the gain factor. The controller has a behavior similar to TCP in the way it reacts to a packet loss by halving the congestion window.

In our implementation, we set $\gamma = 1/TARGET$, so that the max ramp-up rate is the same of TCP, and $\tau = 100ms$, as specified in the BitTorrent's open source μTP implementation and later confirmed in the second version of the LEDBAT Internet draft (July 2010).

A first implementation of the standard LEDBAT linear controller confirmed the presence of intra-protocol fairness issues, known as the late-comer advantage. Primary causes of the problem are *base delay measurement errors* and a wrong *window decrement policy*. In order to overcome this issue we implemented the solution proposed by Rossi et al. [14], i.e. applying the TCP slow-start mechanism to the very beginning of LEDBAT flows. Slow-start forces a loss in the other connections active on the same bottleneck, thus allowing the new-coming flow to measure a correct base delay.

From our experiments however, we found out that even using the slow-start mechanism, slight differences in the base delay measurements might lead to significant unfairness among transfers. We identified the problem to be in the *Additive Increase/Additive Decrease (AIAD)* mechanism. Referring to the work of Chiu et Al.[15], we implemented a a simple Additive Increase/Multiplicative Decrease (AIMD) algorithm instead, which is proven to guarantee stability. As a result, we modified the original Equation 6.4 in such a way that, if the estimated queuing delay exceeds the tau value, the *cwnd* shrinks by a $\beta < 1$ factor, as described here:

$$cwnd(t+1) = \begin{cases} cwnd(t) + \gamma\Delta(t)/cwnd(t) & if \Delta(t) >= 0 \\ cwnd(t) \times \beta & if \Delta(t) < 0 \end{cases} \tag{6.5}$$

With this modification, the decrement of the congestion window is caused by the queuing delay reaching a higher value than the target. The decrement also becomes *proportional* to the sending rate itself. Flows with higher sending rate will then decrease more than others. In our implementation, we used a β factor of 0.99 as found in other implementations [16].

The validity of our considerations has been confirmed by an independent study [16], in which the authors analytically prove that the *additive-decrease* component in the LEDBAT linear controller makes the system unfair, causing transfers to fall out of Lyapunov stability. In the same study, two solutions are proposed for this problem: the first, more conservative, consists of adding a probabilistic drop to the additive increase/decrease dynamics. The other, more aggressive, directly replaces the additive decrease with a multiplicative one, thus confirming our finding.

6.4.2 Variable Mode

We implemented our variable mode transfer priority using the MulTCP algorithm. MulTCP [8] congestion control mechanism allows for a single flow to behave like an aggregate of N concurrent TCP connections, in particular from the point of view of the throughput.

The congestion control module is implemented on top of the normal TCP SACK algorithm using the *priority* value as number of virtual flows N which one transfer must virtualize. MulTCP simply provides the a and b parameters in the additive-increase/ multiplicative-decrease (AIMD) algorithm which are proportional to the number of flows to emulate. MulTCP tries to closely emulate the behavior of TCP in all of its phases, so that in both slow start and congestion avoidance, the congestion window grows as N TCP flows would. In order to avoid sending large bursts of packets if N is too large, causing packet losses, we also implemented the smooth slow start algorithm introduced in [8].

As a further improvement to the original MulTCP algorithm, we implemented the *MulTCP2* algorithm modification proposed by Nabeshima et al. [17]. The mechanism makes it possible, in case of packet loss, to achieve a better virtualized behavior than the original MutlTCP specification, considered to be too aggressive. Here we report the suggested equation implemented in our library.

First, the steady-state average congestion window of N flows is calculated as follows:

$$cwnd_N = N \times cwnd_{std} = \frac{N\sqrt{1.5}}{\sqrt{p}} \tag{6.6}$$

where p is the packet loss rate. Then we derive the $cwnd$ in function of a, AIMD's increase parameter, and b, AIMD decrease parameter:

$$cwnd = \frac{\sqrt{a(2-b)}}{\sqrt{2bp}} \tag{6.7}$$

Finally b is derived from equations 6.6 and 6.7 as:

$$b = \frac{2a}{a+3N^2} \tag{6.8}$$

We then set a to be N, the number of virtualized flows, which is also equal to our $priority$ parameter, and $b = 2/(1+3N)$.

It's easy to observe that, for $a = 1$, the parameters of the AIMD algorithm are exactly the same of the standard TCP.

6.5 Evaluation

In this chapter, we report the results gathered using DTL. Our evaluation methodology is the same used for studies conducted on the LEDBAT protocol such as [14].

We performed our tests in two different configurations:

- a controlled network environment consisting of three host machines using Ubuntu with kernel 2.6.32-22, two as traffic sources and one as traffic sink, connected by a Gigabit links. In order to emulate bottleneck network conditions, we used the *Dummynet* traffic shaper [18]. We created two pipes with RED queue and standard periodic loss, the first with symmetric capacity of $C = 1Mbps$ for the low-bandwidth scenario, and one with symmetric capacity of $C = 10Mbps$, for the high-bandwidth scenario. Both of them are configured with a delay of $20ms$. This is a common configuration setup used by other related works [10][8].

- a real-world environment using the PlanetLab testbed. The experiments are performed using the same configuration of the controlled case, i.e. two hosts, in this case PlanetLab machines, as traffic sources and one as traffic sink. Only non-congested nodes with a symmetric capacity of $C = 10Mbps$ are chosen to host the experiment.

As metrics to evaluate the performance of the DTL, we adopted:

1. the notion of *fairness* introduced by Jain's fairness index F [15], defined as:

$$F = \frac{(\sum_{i=1}^{N} X_i)^2}{N \cdot \sum_{i=1}^{N} X_i^2} \tag{6.9}$$

   where x_i is the rate of flow i and N is the number of flows sharing the bottleneck. Notice that, when N flows get the same bottleneck share, fairness is equal to 1, while it decreases to $1/N$ in the unfair case, where a single flow overtakes the other transfers and uses all available capacity.

2. the *efficiency* or link utilization η, defined as the ratio of the total link utilization normalized over the available bandwidth.

3. the *normalized throughput*, when comparing the total throughput of multiple parallel flows.

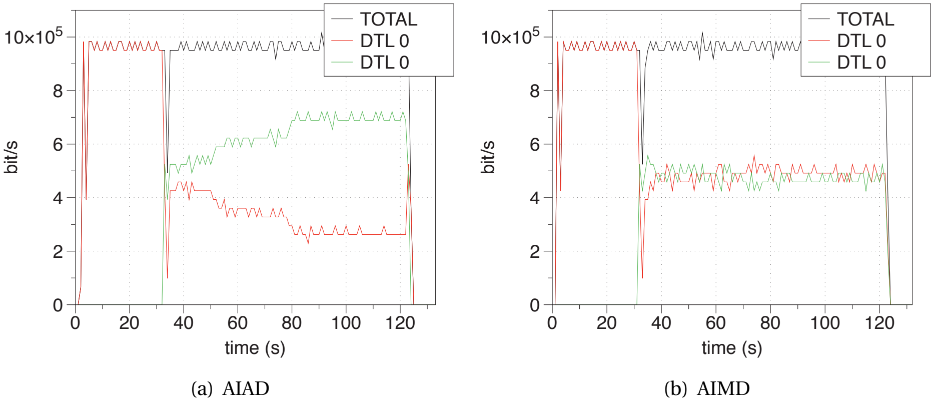
We identify flows with dynamic priority over time as DTL(0,1,2), where 0,1 and 2 are the priority values at different point in time in the experiment.

6.5.1 Polite Mode results

Early results of our initial implementation of LEDBAT, following the draft specification, with the slow start option enabled, immediately showed three fundamental characteristics:

- Capacity of the congestion control algorithm of exploiting all the available bottleneck resources, keeping the one-way delay around the queuing delay target and yielding to TCP flows.

- Fairness in sharing available bandwidth in case of multiple flows starting at the same point in time.



(a) AIAD

(b) AIMD

Figure 6.1: Comparison of Polite Mode intra-protocol fairness with AIAD (a) as opposed to with AIMD (b)

- Unfairness in sharing available bandwidth when flows start at different point in time, even when using the slow start mechanism.

Figure 6.1 shows a comparison of two intra-protocol examples. The first run (a) is executed using the original algorithm with slow-start, while the second (b) using the modified algorithm with multiplicative-decrease. The results have been obtained using the first test configuration. As shown, the original linear controller is not able to timely compensate the initial small error in the base-delay measurements, while the introduction of the non-linear decrease seems to efficiently solve the issue. We would like to underline that, from our experiments, both techniques, i.e. slow start and multiplicative decrease, should be used to guarantee intra-protocol fairness as shown on Figure 6.1(b). As opposed to another previous which claims that implementing multiplicative-decrease is enough to provide such behavior.

In Figure 6.2 we plotted the temporal evolution of the throughput of ten DTL flows with $priority = 0$ (DTL0), starting 5 seconds after each others with a duration of 300 seconds. The test was executed on PlanetLab. The low priority flows perfectly share the bottleneck using all the available resources.

We present in Figure 6.3 the temporal evolution of two polite flows (DTL0) and two TCP flows on the PlanetLab configuration. The two DTL0 flows, one started at $t = 0$ and the other at $t = 60$, equally share the available bandwidth, then yield to a TCP flow at $t = 120$, and again fill up the bottleneck link when the TCP flows, the second one having joined at $t = 180$, terminate at $t = 240$.

Finally, we examine the impact of the Multiplicative-decrease modification in terms of efficiency η, fairness F and loss rate L. We considered two low-priority flows starting at different points in time. Both flows share a common bottleneck both in low and high bandwidth configuration. Each simulation lasted 300 seconds. In Table 6.2, we report the average of multiple simulation runs executed at different point in time. The results

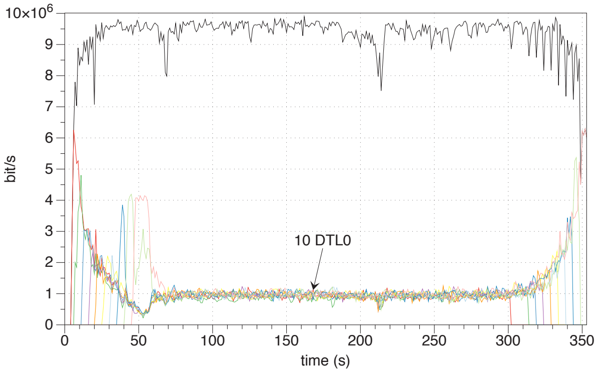


Figure 6.2: Temporal evolution of the throughput of 10 DTL 0 flows on the path planetlab1.sics.se - planet2.zib.de

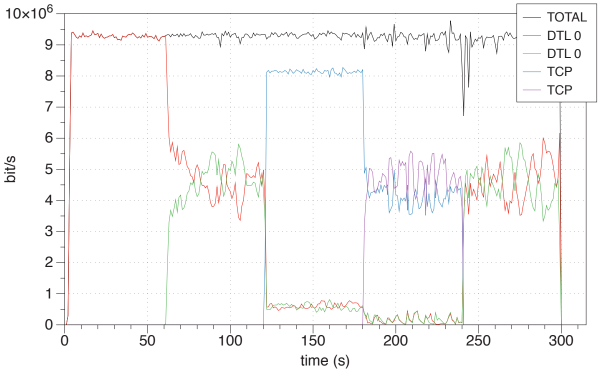


Figure 6.3: Temporal evolution of the throughput of two DTL flows $priority = 0$ (fixed) and two TCP flows on the Planetlab testing configuration

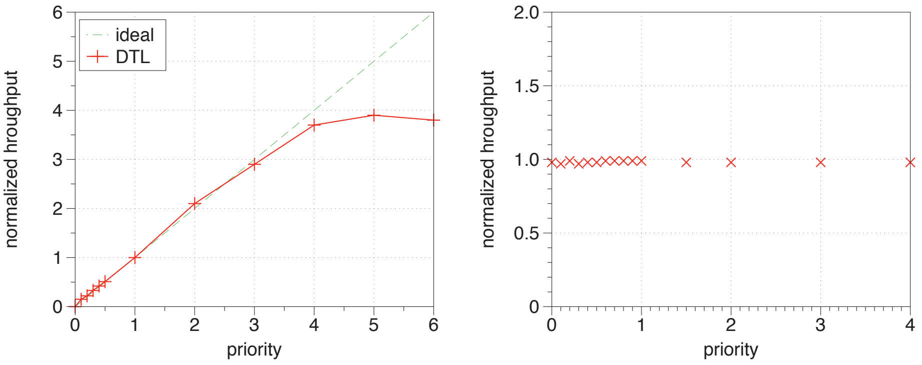
refer to the time interval where both flows are active at the same time. The gathered results clearly demonstrate the positive effect of the multiplicative-decrease modification on the intra-protocol fairness in both bottleneck configurations. As shown, the modification does not affect bandwidth usage and it causes significantly less packet loss.

6.5.2 Variable Mode results

For $priority$ parameter values greater than zero, the congestion control switches to the MulTCP algorithm, which uses packet loss as congestion detection rather than increased delay as in LEDBAT. In Figure 6.4(a), we present the normalized throughput value of one DTL flow parametrized with different priorities, ranging from 1 to 6, against nine TCP flows. In the experiment, all ten flows share a bottleneck link of capacity $C = 10Mbits$. As shown, the DTL flow reaches a throughput value of about $priority$

	C	η	F		L	
	$Mbit$	[%]	μ	σ	μ	σ
No multiplicative decrease	1	99	$5.9\cdot10^{-1}$	$1.01\cdot10^{-1}$	$1.8\cdot10^{-2}$	$5.03\cdot10^{-3}$
	10	98	$6.68\cdot10^{-1}$	$1.87\cdot10^{-1}$	$1.68\cdot10^{-2}$	$6.20\cdot10^{-3}$
With multiplicative decrease	1	98	$9.88\cdot10^{-1}$	$1.03\cdot10^{-2}$	$5.4\cdot10^{-3}$	$1.2\cdot10^{-4}$
	10	98	$9.92\cdot10^{-1}$	$4.90\cdot10^{-3}$	$8.08\cdot10^{-3}$	$7.60\cdot10^{-5}$

Table 6.2: Polite mode $priority = 0$ (fixed): results showing the Link utilization average η, together with the observed Fairness F and packet Loss Rate L, both considering their mean μ and standard deviation σ values



(a) Relative Throughput

(b) Intra-protocol Fairness

Figure 6.4: Plot of the relative Throughput (a) and the Intra-protocol Fairness (b) as a function of $priority$

times the one of a single TCP flow up to $priority$ value of 4, which leads to the best trade-off in terms of throughput. This effectively means that, for $priority = 4$, the DTL flow appears as 4 TCP flows combined, leaving to each of the other nine TCP flows a share of 1/13 of the bandwidth. However, for values of $priority$ greater than 4, the link gets congested and no more improvements are possible due to the high rate of packet loss. In Figure 6.4(b) instead, we compare the normalized throughput of one DTL flow with respect to another DTL flow of same $priority$, for values of $priority$ ranging from 1 to 4. This in order to show how the intra-protocol fairness is maintained for all priorities, as the normalized throughput remains 1.0 for all $priority$ parameters.

Figure 6.5 presents the same PlanetLab test scenario as in Figure 6.3, but this time setting $priority = 1$. As expected, the DTL flows share in a fair manner the bandwidth resources with the two new-coming TCP flows.

Similarly to the polite mode, we decided to examine with more accuracy the DTL behavior for $priority = 1$ in terms of efficiency η, fairness F and $Normalized\ Throughput$. We considered two DTL1 flows for the intra-protocol configuration, and a single DTL1 flow against a TCP flow for the inter-protocol one. Each simulation has a duration of 300 seconds. In Table 6.3, we report the average and standard deviation of multiple runs. The results confirm the ability of DTL1 to correctly share the bandwidth capac-

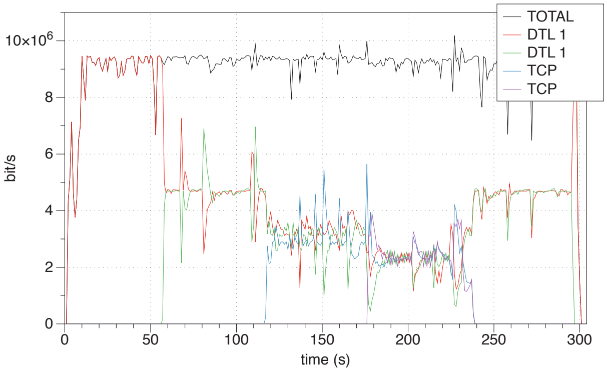


Figure 6.5: Temporal evolution of the throughput of two DTL flows $priority = 1$ (fixed) and two TCP flows on the Planetlab testing configuration

	C	η	F		$Normalized\ Throughput$	
	$Mbit$	[%]	μ	σ	μ	σ
Intra-protocol	1	97	$9.93 \cdot 10^{-1}$	$3.7 \cdot 10^{-3}$	$9.92 \cdot 10^{-1}$	$8.56 \cdot 10^{-2}$
	10	98	$9.96 \cdot 10^{-1}$	$9.3 \cdot 10^{-4}$	$9.84 \cdot 10^{-1}$	$2.3 \cdot 10^{-2}$
Inter-protocol	1	97	$9.83 \cdot 10^{-1}$	$3.5 \cdot 10^{-3}$	$9.84 \cdot 10^{-1}$	$4.47 \cdot 10^{-2}$
	10	98	$9.52 \cdot 10^{-1}$	$8.6 \cdot 10^{-3}$	$9.85 \cdot 10^{-1}$	$2.15 \cdot 10^{-2}$

Table 6.3: Variable mode $priority = 1$(fixed): results showing the Link utilization average η, together with the observed Fairness F and $Normalized\ Throughput$

ity with other TCP flows competing on the same bottleneck under the same conditions, MTU- and RTT-wise.

We then present an experiment in Figure 6.6, using the first test configuration, where DTL flow's priority value changes with time. The simulation lasts 480 seconds, the bottleneck is set to 5Mbit with RTT = 25ms. In order to emphasize the different degrees of aggressiveness, we introduce a TCP transfer on the same bottleneck. The DTL flow starts in polite mode and completely yields to the TCP flow until second 240 when its priority is updated to $priority = 1$, forcing it to switch from LEDBAT to MulTCP. Just after that point, the DTL flow starts to increase its congestion window and share in a fair manner the bandwidth with TCP. Finally, at second 360, we increase the priority up to 2. The congestion window grows now twice as fast. However, in case of packet loss, the same windows is decreased by a value of less than a half, depending on the value of b. The result is a throughput twice as large as in TCP.

In Figure 6.7, we show a plot, produced in our first configuration, of five DTL flows with $priority$ value which varies in time. At first, all flows start with $priority = 0$, then at second 120 four of them get updated to $priority = 1$. We can clearly observe that the four DTL1 flows share the bandwidth equally, while the only DTL0 flow yields to them. At second 230 two of the flows get upgraded to $priority = 2$ and the others keep their previous priority value. As a result, the DTL2 flows consume equally more bandwidth

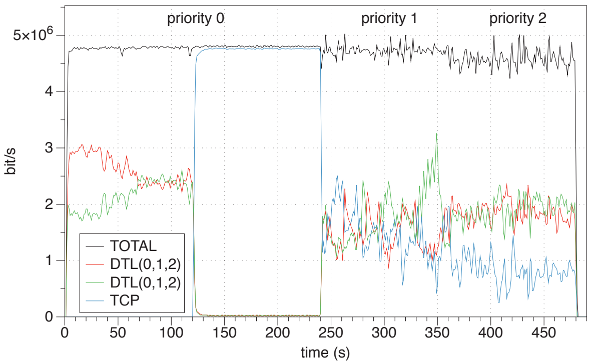


Figure 6.6: Temporal evolution of the throughput of two dynamic priority DTL(0,1,2) flows which share the bottleneck link with a TCP flow

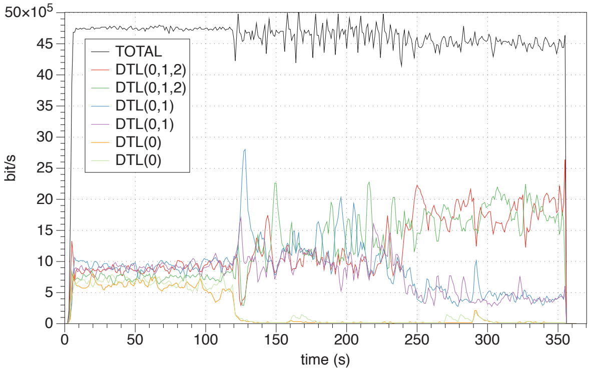


Figure 6.7: Temporal evolution of the throughput of six DTL flows with varying $priority$ value

than the other two DTL1 flows, while the DTL0 still yields to all of them. These results confirm our expectations of multiple priorities intra-protocol fairness.

6.6 Conclusion

In this paper we presented the design and implementation of the DTL application-level library, which is a reliable, variable priority transfer library developed in the Java language, using NIO over UDP. In order to provide different transfer prioritization policies, we implemented two state-of-the art congestion control control mechanisms: LEDBAT and MulTCP. We motivated our choices in using the aforementioned algorithms for the case of configurable priority. As an important achievement, we showed in our results that the library performs on-the-fly transfer priority changes as required, while avoiding connection termination or transfer rate fluctuations. On top of that, our results obtained both using a controlled environment and the Planet Lab testbed show that the library meets all necessary fairness and throughput requirements under the implemented priority levels.

As future work, we would like to provide a more extensive evaluation of our library in a deployed peer-to-peer system. We also would like to investigate the possibility of modifying the MulTFRC algorithm, which provides a more stable approximation of multiple TCP flows behavior than MultTCP, to support variable priority. We plan to make DTL available as an open source project in the near future.

6.7 References

[1] Luca Cicco, Saverio Mascolo, and Vittorio Palmisano. An experimental investigation of the congestion control used by skype voip. In *Proceedings of the 5th international conference on Wired/Wireless Internet Communications*, WWIC '07, pages 153–164, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-72694-4.

[2] Roberto Roverso, Sameh El-Ansary, and Seif Haridi. Natcracker: Nat combinations matter. In *Proceedings of the 2009 Proceedings of 18th International Conference on Computer Communications and Networks*, ICCCN '09, pages 1–7, Washington, DC, USA, 2009. IEEE Computer Society. ISBN Bad - remove. URL `http://dx.doi.org/10.1109/ICCCN.2009.5235278`.

[3] Saikat Guha and Paul Francis. Characterization and measurement of tcp traversal through nats and firewalls. In *Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement*, IMC '05, pages 18–18, Berkeley, CA, USA, 2005. USENIX Association. URL `http://portal.acm.org/citation.cfm?id=1251086.1251104`.

[4] Ledbat ietf draft. `http://tools.ietf.org/html/draft-ietf-ledbat-congestion-03`, July 2010.

[5] Arun Venkataramani, Ravi Kokku, and Michael Dahlin. Tcp nice: A mechanism for background transfers. In *OSDI*, 2002.

[6] Aleksandar Kuzmanovic and Edward W. Knightly. Tcp-lp: low-priority service via end-point congestion control. *IEEE/ACM Trans. Netw.*, 14(4):739–752, 2006.

[7] Giovanna Carofiglio, Luca Muscariello, Dario Rossi, and Claudio Testa. A hands-on assessment of transport protocols with lower than best effort priority. *CoRR*, abs/1006.3017, 2010.

[8] Jon Crowcroft and Philippe Oechslin. Differentiated end-to-end internet services using a weighted proportional fair sharing tcp. *CoRR*, cs.NI/9808004, 1998.

[9] M. Allman, V. Paxson, and E. Blanton. Tcp congestion control, 9 2009. URL `http://www.ietf.org/rfc/rfc5681.txt`.

[10] Dragana Damjanovic and Michael Welzl. Multfrc: providing weighted fairness for multimediaapplications (and others too!). *Computer Communication Review*, 39 (3):5–12, 2009.

[11] M. Handley, S. Floyd, J. Padhye, and J. Widmer. Tcp friendly rate control (tfrc): Protocol specification, 2003.

[12] Multfrc ietf draft. `http://tools.ietf.org/html/draft-irtf-iccrg-multfrc-01`, July 2010.

[13] V. Paxson and M. Allman. Computing tcp's retransmission timer, 2000.

[14] Dario Rossi, Claudio Testa, Silvio Valenti, Paolo Veglia, and Luca Muscariello. News from the internet congestion control world. *CoRR*, abs/0908.0812, 2009.

[15] Dah-Ming Chiu and Raj Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Computer Networks*, 17:1–14, 1989.

[16] Giovanna Carofiglio, Luca Muscariello, Dario Rossi, and Silvio Valenti. The quest for ledbat fairness. *CoRR*, abs/1006.3018, 2010.

[17] Masayoshi Nabeshima. Performance evaluation of multcp in high-speed wide area networks. *IEICE Transactions*, 88-B(1):392–396, 2005.

[18] Marta Carbone and Luigi Rizzo. Dummynet revisited. *Computer Communication Review*, 40(2):12–20, 2010.

PART IV

Peer Sampling

CHAPTER 7

Through the Wormhole: Low Cost, Fresh Peer Sampling for the Internet

Through the Wormhole: Low Cost, Fresh Peer Sampling for the Internet

Roberto Roverso[1,2], Jim Dowling[2] and Mark Jelacity[3]

[1]Peerialism AB
Stockholm, Sweden
[roberto,sameh]@peerialism.com

[2]KTH - Royal Institute of Technology
Stockholm, Sweden
haridi@kth.se [3]University of Szeged, and

Hungarian Academy of Sciences
jelasity@inf.u-szeged.hu

Abstract

State of the art gossip protocols for the Internet are based on the assumption that connection establishment between peers comes at negligible cost. Our experience with commercially deployed P2P systems has shown that this cost is much higher than generally assumed. As such, peer sampling services often cannot provide fresh samples because the service would require too high a connection establishment rate. In this paper, we present the wormhole-based peer sampling service (WPSS). WPSS overcomes the limitations of existing protocols by executing short random walks over a stable topology and by using shortcuts (wormholes), thus limiting the rate of connection establishments and guaranteeing freshness of samples, respectively. We show that our approach can decrease the connection establishment rate by one order of magnitude compared to the state of the art while providing the same levels of freshness of samples. This, without sacrificing the desirable properties of a PSS for the Internet, such as robustness to churn and NAT-friendliness. We support our claims with a thorough measurement study in our deployed commercial system as well as in simulation.

7.1 Introduction

A peer sampling service (PSS) provides nodes in a distributed system with a uniform random sample of live nodes from all nodes in the system, where the sample size is typically much smaller than the system size. PSSes are widely used by peer-to-peer (P2P) applications to periodically discover new peers in a system and to calculate system statistics. A PSS can be implemented as a centralized service [1], using gossip protocols [2] or random walks [3]. Gossip-based PSSes have been the most widely adopted solution, as centralized PSSes are expensive to run reliably, and random walks are only suitable for stable networks, i.e. with very low levels of churn [3].

Classical gossip-based PSSes [2] assume that all nodes can communicate directly with one another, but these protocols break down on the open Internet [4], where a large majority of nodes do not support direct connectivity as they reside behind Network Address Translation Gateways (NATs) and firewalls. To overcome this problem, a new class of NAT-aware gossip-based PSSes have appeared that are able to generate uniformly random node samples even for systems with a high percentage of *private nodes*, that is, nodes that reside behind a NAT and/or firewall [4, 5, 6].

State of the art NAT-aware gossip protocols, such as Gozar [4] and Croupier [5], require peers to frequently establish network connections and exchange messages with *public nodes*, nodes that support direct connectivity, in order to build a view of the overlay network. These designs are based on two assumptions: i) connection establishment from a private to a public peer comes at negligible cost, and ii) the connection setup time is short and predictable. However, these assumptions do not hold for many classes of P2P applications. In particular, in commercial P2P applications such as Spotify [1], P2P-Skype [7], and Google's WebRTC [8] establishing a connection is a relatively complex and costly procedure. This is primarily because security is a concern. All new connections require peers to authenticate the other party with a trusted source, typically a secure server, and to setup an encrypted channel. Another reason is that establishing a new connection may involve coordination by a helper service, for instance, to work around connectivity limitations that are not captured by NAT detection algorithms or that are caused by faulty network configurations. To put this in the perspective of our P2P live streaming system [9], which we believe is representative of many commercial P2P systems, all these factors combined produce connection setup times which can range from few tenths of a second up to a few seconds, depending on network latencies, congestion, and the complexity of the connection establishment procedure. In addition to these factors, public nodes are vulnerable to denial-of-service attacks, as there exists an upper bound on the rate of new connections that peers are able to establish in a certain period of time.

It is, therefore, preferable in our application to build a PSS over a more stable topology than the constantly changing topologies built by continuous gossiping exchanges. An instance of such a stable topology might be an overlay network where connections between peers are maintained over time and node degree is kept constant by replacing failed connections. Random walks (RWs) over such a stable overlay are potentially an alternative to NAT-resilient gossip protocols. However, we are not aware of any work which addresses the problem of random walks over the Internet in presence of NATs. That said, RW methods over other stable networks that are not the Internet[10] are not practical in our case because of the high level of churn experienced in P2P systems, which causes the PSS' quality of service to degrade by interrupting or delaying the RWs. Furthermore, RWs are not able to provide fresh-enough samples because a random walk has to complete a large number of hops (depending on the topology of the network) to collect or deposit a sample.

In this paper, we present an alternative approach where the PSS creates enough new connections to ensure the PSS' quality of service in the face of peer churn, but not too many as to exceed peers' upper bounds on connection establishment rate. We call our system a *wormhole*-based PSS (WPSS). WPSS can tune the number of new stable network connections established per sample to match application-level requirements and constraints. We show that our system can provide the same level of freshness of samples as the state of the art in NAT-aware PSSes [5] but with a connection establishment rate that is one order of magnitude lower. This, without sacrificing the desirable properties of a PSS for the Internet, such as robustness to churn, NAT-friendliness and local randomness of samples.

The main idea behind WPSS is that we separate the service into two layers. The bottom layer consists of a stable base overlay network that should be NAT-friendly, with private nodes connecting to public nodes, while public nodes connect to one another. On top of this overlay, every node periodically connects to a random public node selected from the base overlay (not necessarily a neighbor in the base overlay). Using the random public node, each node systematically places samples of itself on nodes in the neighborhood of this random public node. We call these links to random public nodes *wormholes*. That is, a wormhole is a link to a public node that is selected uniformly and independently at random. We do not require hole-punching or relaying to private nodes in this paper, although those techniques can be used as well if there are not enough public nodes.

In addition to explaining the WPSS algorithm (Section 7.4), our contributions also include a analytical comparison between WPSS and related work (Section 7.5), and a thorough evaluation of the protocol in both simulation and our deployed system (Section 7.6). The latter experiments include a comparison with the state of the art NAT-resilient PSS, Croupier.

7.2 System Model

We model a distributed system as a network of autonomous nodes that exchange messages and execute the same protocol. Nodes join and leave the system continuously. We consider large-scale networked systems with limited connectivity, for example, where a large majority of nodes reside behind NAT devices or firewalls. A public node has a globally reachable IP address; this includes nodes that use IP addresses allocated by UPnP Internet Gateway Devices. A private node does not support direct connectivity, typically, because it is behind a firewall or a NAT.

Each node discovers its NAT type (public or private) at bootstrap-time and also when its IP address changes using a NAT-type identification protocol. We also assume that a bootstrap service provides newly joined nodes with a small number of node descriptors for live nodes in the system. Each node separately maintains open network connections to a small, bounded number of randomly selected public nodes in a *stable* base overlay. The node degree on the base overlay is kept constant over time by replacing connections to failed nodes with new ones.

7.3 Background and Related Work

The first generation of peer sampling services were not designed to account for NATs [2, 11, 12]. In recent years, researchers have worked on NAT-aware gossip protocols [13], and the first NAT-aware PSSes were Nylon [6] and Gozar [4]. They enabled gossiping with a private node by relaying a message via an existing node in the system that had already successfully communicated with the private node. Nylon routes packets to a private node using routing tables maintained at all nodes in the system. In contrast, Gozar routes packets to a private node using an existing public node in the system. Gozar does not require routing tables as the address of a private node includes the addresses of j public nodes that can act as relays to it. To improve Gozar's reliability, gossip messages can be sent to a private node in parallel via its j relays, where j is a system parameter. Parallelizing relay messages also reduces the latency of gossip messages, but at the cost of an increase in protocol overhead. Both Nylon and Gozar require that private nodes refresh their NAT bindings by periodically pinging their neighbors.

Croupier [5] provided an alternative NAT-aware PSS that removed the need for relaying gossip messages to private nodes. Instead of routing gossip messages to private nodes, gossip requests are only sent to public nodes that act as croupiers, shuffling node descriptors on behalf of both public and private nodes. Public nodes are able to send response messages through a private node's NAT, as the shuffle request from the private node created a NAT binding rule that is subsequently used to forward the response to the private node. Two similarities between Croupier and WPSS are that nodes need to know whether they are public or private, and that the stable topology created by WPSS is similar to the dynamic topology maintained by Croupier, where both private and public nodes only connect to public nodes. In contrast to WPSS, Croupier provides a decentralized NAT-type identification protocol to discover a node's NAT type (public or private), while our system provides an infrastructure-supported service based on [14].

One general difference between gossip-based approaches and WPSS is that a gossip message combines both the advertisement of the source node's descriptor as well as a set of other node descriptors for dissemination [2][11], while WPSS messages only contain an advertisement of the initiating node's descriptor, making the message size somewhat smaller.

PSSes have also been implemented using independent RWs on stable, and even very slowly changing, dynamic graphs [3]. Intuitively, RWs work by repeatedly injecting randomness at each step until the initiator node is forgotten. RW sampling can be classified

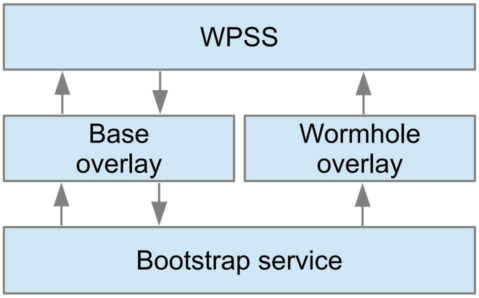


Figure 7.1: WPSS layered architecture.

as either *push-based* or *pull-based*. In push-based RWs, the initiating node advertises its descriptor as a random sample at the terminating node, where the RW completes. In pull-based RWs, the final node's descriptor is advertised as a random sample at the initiating node. An extension to these methods allows a new direct link (DL) to be created by the final node to fetch (pull) or send (push) a fresher sample. Depending on the network topology, RWs may require a large number of hops over unreliable links with highly varying latencies before they reach good mixing and complete, thus samples can be relatively old on arrival at final nodes. Direct links provide more recent samples at the cost of creating a new network connection. However, on the open Internet, DLs will not work if the initiator is a private node and there is no support for NAT traversal. To the best of our knowledge, there has been no previous work on making RW-based PSSes NAT-aware.

7.4 Wormhole Peer Sampling Service

The core of our idea is that nodes disseminate (push) advertisements of themselves over a stable base overlay using short walks, that also utilize *wormhole* links. That is, the node that initiates the advertisement sends its own descriptor over a small number of hops in the base overlay and places it at the node where this (typically short) walk terminates. The short length of the walks guarantees the freshness of the samples. However, to be able to provide good random samples, despite completing only a relatively few hops, our advertisements always traverse a wormhole link.

A wormhole link (or wormhole, for short) points to a public node in the system that is selected independently at random. In our WPSS protocol, every node discovers such a random public node to act as its wormhole, and every node only has one wormhole active at any given point in time. The distribution of the wormholes does not need to be uniform in the system, for example, we restrict the wormholes to be public nodes. However, all the nodes must sample their wormhole from the same distribution independently. This guarantees that for each node it is true that any other node will pick it as a wormhole with the same probability.

A new network connection will be created only when a wormhole is traversed for the first time by an advertisement. The wormhole is then reused for a few subsequent advertisements from the same initiator node, in which case no new network connection needs to be established. This makes it possible to decrease the number of new links we establish.

The very first time an advertisement traverses the wormhole, it can be considered to have reached a random public node. Thus, the advertisement can now be placed at the public node as a new sample. However, if the wormhole has already been used, or the public node already has a sample from the initiator node, then the advertisement will start a random walk over the base overlay until it either (1) reaches a node that does not already have a sample from the initiator node or (2) it reaches a given time-to-live (TTL) that guarantees good quality sampling. Clearly, as a wormhole creates a link to a public node, advertisements through it will place a sample first at that public node unless it already has an advertisement from the initiator. However, the reuse of the wormhole causes advertisements to continue, allowing them to also finish at private nodes, as private nodes are connected to public nodes over the base overlay.

When a wormhole is reused by a advertisement, the expected number of hops the advertisement will have to take increases, as it needs to reach a node that does not already have a sample from the initiator node. To counteract this, new wormholes are created. WPSS defines a wormhole renewal period as a parameter for creating new wormholes enabling users to control how frequently new network connections will be created. If a new wormhole is created for each advertisement, then we get a protocol very similar to RW-push-DL (see Section 7.5). If wormholes are never updated, the behavior converges to that of RW-push, eventually. In between, there is a range of interesting protocols some of which—as we will argue—achieve the goals we set.

We illustrate the interaction between the wormholes and the base overlay in Figure 7.2. In the figure, the bottom layer is the base overlay, while the upper layer shows the wormhole overlay, containing a single wormhole. The upper layer also shows two advertisements over the wormhole that follow links in the base overlay to place samples in the vicinity of the wormhole.

7.4.1 WPSS Architecture

We implemented WPSS in a modular, layered architecture, illustrated in Figure 7.1. WPSS requires a stable base overlay and a set of wormhole links. It is very important that the wormhole links are drawn independently from an identical (but not necessarily uniform) distribution. Since the random samples generated using wormholes do not guarantee inter-node independence, these samples cannot be used to generate new wormhole links. So, another PSS is required that provides independent samples of public nodes. We call this the bootstrap PSS. The bootstrap PSS will have relatively low load, since wormholes are refreshed rarely, given that WPSS is designed to reduce the number of new network connections created. For this reason, the bootstrap PSS can be implemented even as a central server. Alternatively, public nodes can generate independent samples by periodically starting random walks that continue until they reach their TTL.

In our architecture, the base overlay can be any connected overlay, as long as the

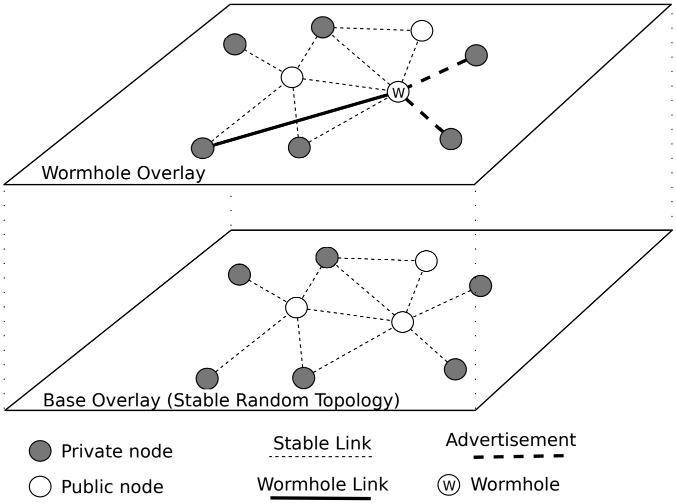


Figure 7.2: The Base Overlay contains stable links between nodes (bottom layer). Wormholes (the thick line on the upper layer) are created to public nodes. The upper layer also illustrates a private node placing two advertisements at neighboring nodes to the wormhole.

TTL of the advertisements is set to make sure the random walks are long enough to provide high quality samples. It is beneficial, however, to maintain a random base overlay due to its low mixing time [15, 16]. Therefore, we construct a stable undirected overlay that is random, but where private nodes are connected to a random set of public nodes, thus avoiding the need for NAT traversal. This results in a stable topology with a low clustering coefficient, a constant degree for private nodes, and a narrow (roughly binomial) degree distribution for public nodes. This is similar to the topology maintained by Croupier [5], although Croupier's topology is dynamic.

The base overlay also needs a PSS in order to replace broken links with new random neighbors. The samples used to replace random links can be obtained either from the bootstrap PSS or from WPSS itself, with the choice depending on application requirements. For example, with relatively low churn rates the bootstrap PSS might be a better choice. However, under high churn, when many samples are needed, the cheaper WPSS samples are preferable despite the lack of inter-node independence that might result in higher clustering. This potentially higher clustering is not a problem as long as the network maintains good enough mixing properties.

The base overlay service strives to keep the overlay connected by keeping and repairing in case of failures a fixed number of outgoing links. In order to detect failures of base topology links, we implement a simple failure detector based on timeouts.

Algorithm 5 Wormhole peer sampling

```
procedure onWormholeFailure ⟨⟩
    wormhole ← getNewWormhole()
end procedure

procedure onWormholeTimeout ⟨⟩                                ▷ Every $\Delta_{wh}$ time units
    wormhole ← getNewWormhole()
end procedure

procedure onAdTimeout ⟨⟩                                      ▷ Every $\Delta$ time units
    ad ← createAd()
    hops ← 1
    sendAd(wormhole, ad, hops)
end procedure

procedure onReceiveAd ⟨ad, hops⟩
    if hops == getTTL() || acceptAd(ad) then
        view.addAd(ad)
    else
        j ← getMetropolisHastingsNeighbor(baseOverlay)
        sendAd(j, ad, hops+1)
    end if
end procedure
```

7.4.2 Wormhole Peer Sampling Skeleton

Algorithm 5 contains the pseudocode of WPSS that implements the ideas described above. The algorithm contains a number of abstract methods, implementations of which will be discussed in the subsequent sections. The algorithm is formulated as a set of event-handlers that are in place on every node in the system. The events in the system are classified into three types: failures, timeouts and advertisements.

Failure events are generated by detecting failing neighboring nodes in the two topology layers in Figure 7.1. We deal with these failure events by picking a new neighbor using the appropriate PSS through the abstract methods GetNewLink and GetNewWormhole. Timeout events are generated by two local timers with a tunable period. The periods of these timers are protocol parameters that are the same at all nodes. One of the timers, the wormhole timer, has a period of Δ_{wh}. This timer triggers the generation of new wormholes.

The other timer defines the rate at which advertisements are published by all the nodes. Clearly, this rate is the same as the average rate of receiving new random samples, so the period of this timer must be Δ at all nodes. This timer triggers the sending of one advertisement over the local wormhole. Finally, the event of receiving an advertisement is handled by first checking whether the node is willing to add the given sample to its set of samples or View (that is, whether it will consume the advertisement).

This check is performed by the AcceptAd method. The AcceptAd method consumes an advertisement only if its sample is not already contained in the node's view, thus promoting diversity of samples. On top of that, AcceptAd makes sure that every node consumes advertisements at the same rate, namely one advertisement in each Δ time period. We implement this rate control mechanism only at the public nodes; if the acceptance rate on the public nodes is $1/\Delta$, then private nodes are guaranteed to observe the same rate on average due to the advertisement generation rate being $1/\Delta$ at all the nodes. To control the rate, we need to approximate the period of receiving advertisements. To do this, we calculate the running average and the average deviation of the delays between consecutive acceptance events. If the approximated period, increased by mean deviation, is higher than Δ then the advertisement is accepted and the approximation of the period is updated. We show later in Section 7.6 that the AcceptAd method successfully balances advertisements over public and private nodes.

If the node does not consume the advertisement, it sends it on to another node using the Metropolis-Hastings transition probabilities over the (random, stable) base network which results in a uniform stationary distribution [17]. Let d_i denote the degree of node i, that is, the number of neighbors of node i. Note that the graph is undirected. The implementation of GetMetropolisHastingsNeighbor works as follows. First, we select a neighbor j with uniform probability, that is, with probability $1/d_i$. Then, we return j with probability $\min(d_i/d_j, 1)$, otherwise we return node i itself, that is, the advertisement will visit i again.

Note that the node will consume the advertisement also if the TTL of the advertisement is reached, since at that point it can be considered a uniform random sample. The TTL is returned by GetTTL, that can be implemented in many different ways depending on the system properties. We simply set a constant TTL in our experiments.

7.5 Analytical comparison

Here, we provide an analytical comparison between related work and WPSS. In order to do that, we develop a number of metrics and criteria. As a general assumption, we assume that the considered PSSes provide a continuous stream of random node descriptors at every node in the network. To be more precise, we require each node to receive one sample every Δ time units on average, where Δ is the *sampling period.*

7.5.1 Metrics of Quality of Service and Cost

The minimum that we assume about any PSS is that the stream of samples at a given fixed node is *unbiased* (that is, the samples received by a node are independent of the node) and that samples are *uniform* (that is, any fixed individual sample has a uniform distribution). These properties hold for WPSS. This is almost trivial and can be formally proven using the facts that WPSS is blind to the content of node descriptors and wormholes are random. The technical proof is omitted due to lack of space.

Apart from this, we characterize the quality of service using the properties of *freshness* and *independence.* Freshness is the age of the sample, that is, the delay between the recording of the sample at the source node and its delivery at the terminating node.

	Quality of service		Cost per sample	
Algorithm	Freshness	Indep.	# New links	Bandwidth
RW-push	TTL$\cdot\delta_{hop}$	yes	0	TTL
RW-push-DL	δ_{hop}	yes	1	TTL+1
RW-pull	2$\cdot$TTL$\cdot\delta_{hop}$	yes	0	2$\cdot$TTL
RW-pull-DL	δ_{hop}	yes	1	TTL+1
gossip healer	$O(k\log k)\Delta$	no	$1/k$	1
gossip swapper	$O(k^2)\Delta$	no	$1/k$	1
WPSS	$h(\Delta_{wh}/\Delta)\delta_{hop}$	no	Δ/Δ_{wh}	$h(\Delta_{wh}/\Delta)+\Delta TTL/\Delta_{wh}$

Table 7.1: Analytical comparison of different PSS implementations.

Obviously, fresher samples are always preferable. Independence is a property of the random samples that states whether the stream of samples generated at any two nodes are statistically independent of one another or not. Independence is not a hard requirement for many applications. Having unbiased and uniform streams (that are not necessarily independent) is often sufficient.

We focus now on two costs for the PSSes: *bandwidth* and *link creation rate*. Bandwidth is the amount of data transferred per received sample. The link creation rate is the number of new network connections (links) that are created (or in other words, the number of new network connections established) per sample. This new measure is motivated by our experience with deployed commercial P2P streaming.

7.5.2 Analysis of Existing PSS Algorithms

In Table 7.1, we summarize the quality of service and costs of the PSS algorithm space for existing RW and gossiping approaches, where δ_{hop} is the average latency for a hop in a RW. The example given here results in a connection establishment that is tolerable for our application and good enough level of freshness for samples.

In the case of RW-push, the initiating node's descriptor will be added to the final node's sample set (the node where the RW terminates). In the case of RW-pull, the final node's descriptor is added to the sample set of the initiating node. In addition, if the physical network supports efficient routing then after the RW terminates a new direct link (DL) can be created by the final node to fetch (push) or send (pull) a fresh sample as well. Clearly, using direct links one can achieve a higher freshness at the cost of creating new links.

For the gossip-based protocols, we consider an idealized gossip protocol that has a view size of $2k$, and a gossip period of $k\Delta$. This setting makes sure that a node gets one new sample per sampling period (Δ) on average, since, in a typical gossip-based PSS, nodes refresh half of their views in each round. This is an optimistic assumption, because k is an upper bound on the independent samples.

We consider the two classical gossip protocols: *healer* and *swapper* [2]. Let us provide a quick intuitive analysis of the freshness for these protocols. In the case of healer, fresher descriptors always have preference when spreading. This results in an exponential initial spreading of newly inserted descriptors, until their age reaches about the logarithm of the view size. At that point the fresher descriptors take over. This results in an average age that is equal to $O(\log k)$, where age is measured in gossip rounds. So, freshness equals $O(k \log k)\Delta$. In the case of swapper, descriptors perform batched RWs, during which no descriptor gets replicated. In every round, every node removes one random descriptor and replaces it with its own new descriptor. For this reason, the age of a descriptor follows a geometric distribution, where the expected age (measured in number of rounds) is proportional to the view size: $O(k)$. So the average freshness equals $O(k^2)\Delta$.

Let us now discuss the NAT-aware gossip protocols. Croupier has the same costs and freshness as *swapper* (although node-level load is higher at public than private nodes, as they provide shuffling services). In Gozar, public nodes create the same number of new links and generate the same bandwidth as *healer*. Private nodes, on the other hand, generate $j * 2$ times more bandwidth, as messages are sent over 2 links: first to the j relays and then to the private nodes. Gozar also creates and maintains an extra constant number of links to relays ($j * N$, where N is system size). Nylon has unbounded costs, as routing paths to private nodes can be arbitrarily long, and for that reason it is not considered here.

For WPSS, the most important parameter is Δ_{wh}, the wormhole renewal period. One important value that is determined by Δ_{wh}/Δ is the average number of hops that an advertisement makes before being accepted by a node. Let us denote this value by $h(\Delta_{wh}/\Delta)$. Now, the freshness of samples can be calculated using $h(\Delta_{wh}/\Delta)\delta_{hops}$. The number of newly created links per sample is Δ/Δ_{wh}, since the only new links are the wormholes. Finally, the bandwidth utilization per sample is given by $h(\Delta_{wh}/\Delta) + \Delta TTL/\Delta_{wh}$ where the second addend accounts for the cost of the bootstrap PSS, assuming it is implemented as a RW over the stable random topology.

Clearly, since Δ is a constant given by the application requirements, and Δ_{wh} (and thus Δ/Δ_{wh}) is a parameter of the protocol, the most important component is $h(\Delta_{wh}/\Delta)$. We know from the definition of the protocol that $h(1) = 1$ (one hop through the wormhole), and $\lim_{x\to\infty} h(x) = TTL$. They key question is whether $h(\Delta_{wh}/\Delta)$ grows slowly. In Section 7.6, we show that this is indeed the case.

We calculate the properties of the different protocols using a typical practical parameter setting, with the results summarized in Table 7.2. For this example, we set $k = 10$, and $\delta_{hop} = 0.1$ seconds. We assume that $TTL = 100$. During the experimental evaluation in Section 7.6 we apply the same setting, and we present a justification for it as well.

For WPSS, we set $\Delta_{wh} = 10\Delta$, which is shown in Section 7.6 to provide a good tradeoff between reducing the number of new connections created per round and keeping samples fresh. For $\Delta_{wh} = 10\Delta$, the freshness is evaluated in our experiments in Section 7.6 as $h(10) \approx 3$, and, thus, this value is given in the table.

Note that freshness depends on the sampling period Δ only in the case of the gossip protocols. Typical settings for Δ range from 1 to 10 seconds. Besides, it is not meaningful to set $\Delta < \delta_{hop}$, so we can conclude that WPSS provides samples that are fresher by an

Algorithm	Freshness	# New links	Bandwidth
RW-push	10	0	100
RW-push-DL	0.1	1	101
RW-pull	20	0	200
RW-pull-DL	0.1	1	101
gossip healer	33Δ	0.1	1
gossip swapper	100Δ	0.1	1
WPSS	**0.3**	**0.1**	**13**

Table 7.2: Example comparison of PSS implementations. The example uses $k = 10$, $\delta_{hop} = 0.1$, TTL=100, and $\Delta_{wh} = 10\Delta$.

order of magnitude than those provided by the fastest possible gossip protocol.

We would like to point out that gossip protocols will use a lot less bandwidth compared to the other protocols under most parameter settings, assuming our invariant (which fixes the same rate of receiving new samples). If we allow for the same bandwidth for gossip protocols and WPSS by speeding up gossip then gossip samples will get proportionally fresher, besides, proportionally more samples will arrive in a unit time as well. However, freshness will still depend on the length of the (now shorter) gossip cycle. In addition, most importantly, the number of new links will increase proportionally since all the connections in gossip protocols are new connections. Reducing the number of new links is one of our main motivations.

Gossip protocols have a further advantage due to batching $2k$ advertisements in a single message. This could additionally reduce bandwidth costs, as the relative amount of meta-data per packet sent is lower if the network packets from gossip messages are closer in size to the network's maximum transmission unit (MTU) than WPSS messages. However, in our target application the size of an advertisement can be large, which justifies our focus on bandwidth as a measure of cost as opposed to the number of messages sent.

Based on the above cost analysis of existing PSS algorithms, our main observation is that no single method other than WPSS offers a combination of three desirable properties for a PSS: fresh samples, a low number of new links, and low bandwidth overhead. Apart from being vulnerable to failures, RW methods can offer fresh samples with an extremely high number of new links (one per each sample) or relatively old samples without creating new links; in both cases with a relatively high bandwidth. Gossip based methods provide even older samples than RW methods.

7.6 Evaluation

We now evaluate the performance of WPSS in simulation as well as in our deployed system. The experiments on the deployed system show that our protocol provides the desirable PSS properties of fresh samples, randomness, and low cost in a real environment. The simulation experiments, in contrast, test robustness in scenarios that are difficult or impossible to reproduce in deployment, such as different churn levels.

Our implementation follows the structure outlined in Figure 7.1. The bootstrap service component provides addresses of random public nodes that are used by the upper layer to build the base overlay and to create wormholes. In our implementation, the bootstrap service initiates RWs from the public nodes in the base overlay using the same transition probabilities as used by WPSS. The rate of starting these walks is double the rate of wormhole renewal to account for failure events in the base overlay and the wormhole overlay. If the bootstrap service runs out of local random samples generated by RWs (because of massive join or failure events), a central bootstrap server is contacted for random samples.

We set $TTL = 100$. After a RW of this length, the distribution is very close to uniform in the base overlays we apply here. More precisely, its total variational distance [16] from the uniform distribution is less than 10^{-6} in all base overlays in this section. In a fully public network—but with the same number of nodes and random links—the same quality can be reached with $TTL = 16$. This means that in our environment, independent RW methods have a further disadvantage due to the constrained topology. By increasing the network size, this difference becomes larger. As we will see, WPSS walks will always terminate much sooner in most practical settings, so the TTL is not a critical parameter from that point of view.

The base overlay service strives to keep the overlay connected by identifying and repairing broken links, thus, maintaining a fixed number of outgoing links at each node. In order to detect failures of base topology links and the wormhole, we implement a simple failure detector based on timeouts. Every node maintains 20 links to random public nodes in the base overlay. However, these links are bidirectional, so the effective average degree is 40 as a result.

7.6.1 Experimental Setup: Simulation

For simulations, we implemented WPSS on the Kompics platform [18]. Kompics provides a framework for building P2P systems and a discrete event simulator for evaluating those systems under different churn, latency and bandwidth scenarios. All experiments are averaged over 6 runs. Unless stated otherwise, we applied the following settings. The view size (the number of freshest random samples a node remembers) was 50 and we set $\Delta = 1$ second. For all simulation experiments, we use a scenario of $N = 1000$ nodes that join following a Poisson distribution with a mean inter-arrival time of 100 milliseconds. In all simulations 20% of the nodes were public and 80% were private, which reflects the distribution observed in the commercial deployments of our P2P application.

7.6.2 Experimental Setup: Deployment

In order to test WPSS in a real environment, we implemented the protocol using [19], a production-quality framework for building event-based distributed applications. The

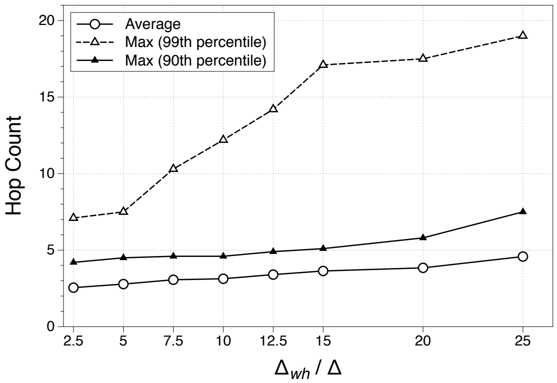


Figure 7.3: Average hop count in simulation, with real deployment input data

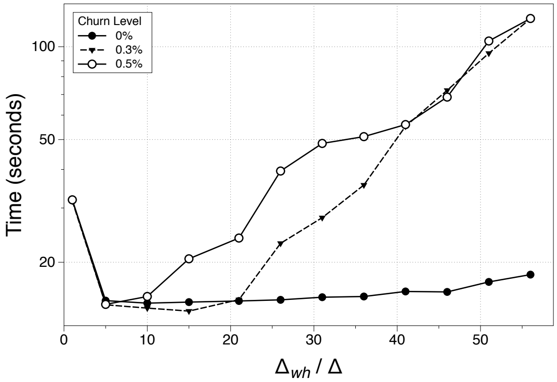
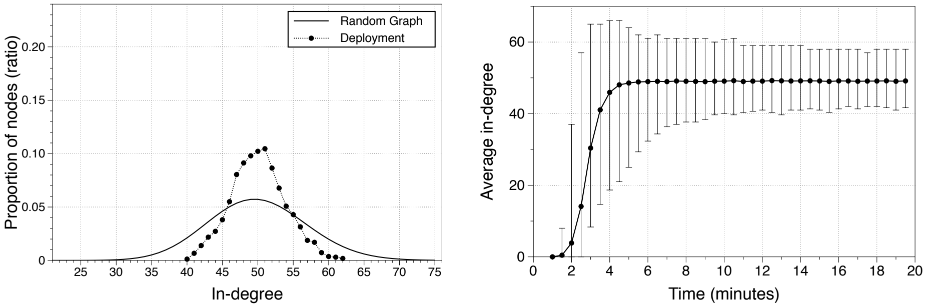


Figure 7.4: Time to publish 20 ads per peer

framework utilizes a UDP-based transport library that implements the same reliability and flow control mechanisms as TCP [20].

We tested WPSS in our test network, where volunteers give us permission to conduct experiments on their machines using a remotely-controlled test agent. The test network contains around 12000 installations. The network included nodes mostly from Sweden (89%) but also some from Europe (6%) and USA (4%). For connectivity, 76% of the nodes were behind NATs or firewalls and 19.2% were public nodes, while the rest (4.8%) could not be determined by our NAT-type identification protocol.

Each experiment is run in wall-clock time and thus it is subject to fluctuations in network conditions and in the number of nodes involved. In order to keep a good level of reproducibility, we selected a subset of 2000 of the more stable nodes, out of an average of 6200 online, which are representative of both our test network and the Internet in Sweden, in general. For each data point, the deployment experiments were repeated a



(a) Converged distribution

(b) Evolution over time with error bars

Figure 7.5: In-degree measurements

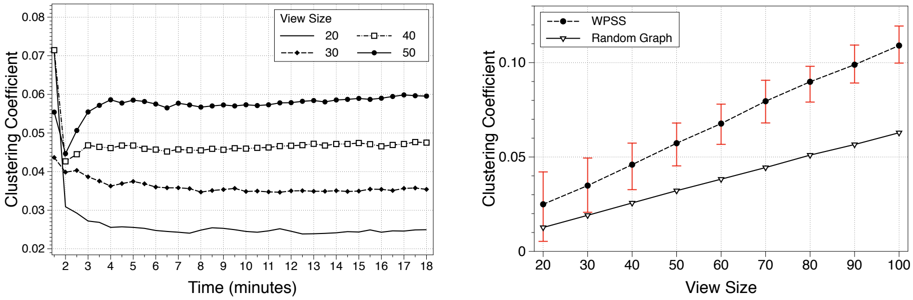
number of times varying from 3 to 10 runs, depending on the variance of the results, and every run lasted 20 minutes. In all deployment experiments, the nodes join at a uniform random point in time within the first 2 minutes from the start of the test. Unless stated otherwise, we set the a view size to 50, and $\Delta = 2$ seconds.

7.6.3 Freshness

In this set of experiments, we measure the freshness of samples on our deployed system using the average hop count as well as the $90th$ and $99th$ percentiles. As we can see in Figure 7.3, both the average hop count ($h(\Delta_{wh}/\Delta)$) and the 90th percentile grow slowly with increasing Δ_{wh} (wormhole renewal period), while the TTL is never reached by any advertisement. The $99th$ percentile, however, grows more quickly when Δ_{wh}/Δ exceeds 5 seconds as a small number of public nodes have under-performing or too few connections, meaning that advertisements that arrive at them via wormholes have to take more hops to complete.

In simulation, we now measure the average time required for a node to publish 20 consecutive advertisements. This characterizes the average freshness of the advertisements also taking into account all possible delays, not only hop count. We examine scenarios with and without churn. The total time we measure includes the time required for an advertisement to be accepted by a node ($h(\Delta_{wh}/\Delta)\delta_{hops}$), and the amount of time devoted to opening new wormhole links, if applicable. It also includes the retransmission time in case of failures.

We note that the average time required to establish a network connection to a public node in our system and was measured to be $1250\,ms$ in deployment. In Figure 7.4, we can observe the freshness of advertisements under three different scenarios: with no churn, and with a churn level of 0.3% or 0.5% of nodes failing and joining every 10 seconds. Recall that in simulation we had $\Delta = 1$ second. We consider these churn figures to be representative of the deployments of our commercial P2P application. The required time to publish an advertisement increases with Δ_{wh}/Δ, especially in the scenarios with



(a) Evolution for different view sizes

(b) Converged value for increasing view sizes

Figure 7.6: Clustering coefficient measurements

churn. This is due to both a higher average hop count, as shown later in the evaluation, and to retransmissions caused by failures along the advertisement path.

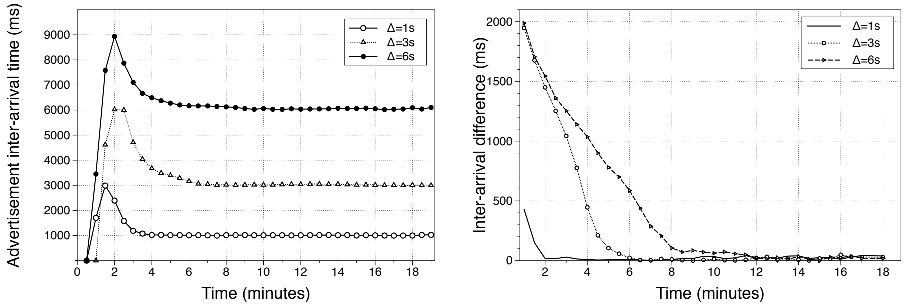
The most interesting result here is that the performance is optimal for Δ_{wh}/Δ between about 5 and 10 even under high churn. Given this finding, we set $\Delta_{wh} = 10\Delta$ for the remaining experiments, as this value has good freshness (even considering the $99th$ percentile), while the number of new links required is relatively low.

7.6.4 Randomness

Similar to [2, 4, 5], we evaluate here the global randomness properties of our deployed system by measuring properties of the WPSS overlay topology (that is, not the base overlay, but the overlay that is defined by the samples stored at nodes). In this set of experiments, we measure the indegree distribution of the WPSS overlay network, its convergence time for different view sizes, and finally its clustering coefficient for different view sizes.

With samples drawn uniformly at random, we expect that the in-degree to follow the binomial distribution. In Figure 7.5(a), we can see that WPSS actually results in a distribution that is even narrower than what is predicted by the uniform random case, which suggests good load balancing. In Figure 7.5(b) we can also see that the average indegree converges after around 4 minutes and the variance after around 8 minutes, respectively; an acceptable convergence time for our system. Since in deployment we had $\Delta = 2$ seconds, this translates to 120Δ and 240Δ, respectively.

In Figure 7.6(a), we can see that the clustering coefficient converges at roughly the same rate as the indegree distribution. Figure 7.6(b) indicates that the clustering coefficient is higher than that of the random graph by a constant factor, which is due to the fact that WPSS does not guarantee independent samples at nodes that are close in the stable base overlay, as we explained previously. As we increase the view size, the clustering coefficient increases simply due to the larger chance of triangles; this is true for the random graph as well.



(a) Average values for samples (terminating advertisements) at all peers

(b) Difference values between public and private peers

Figure 7.7: Inter-arrival time measurements

7.6.5 Inter-arrival Time of Advertisements

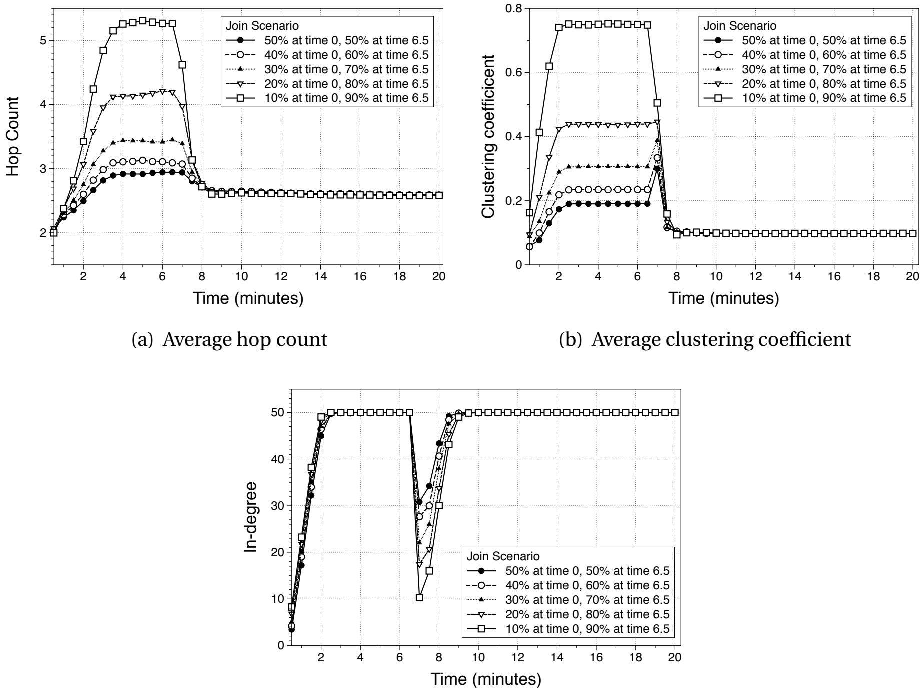
These experiments were again conducted in the deployed system. Figure 7.7(a) shows that the average inter-arrival times for samples (advertisements) converges to the respective advertisement period after roughly 120Δ seconds. As public nodes can be wormhole exits, they receive samples at a higher rate than private nodes. But, as Figure 7.7(b) shows, the method *acceptAd* (see Section 7.4) successfully balances samples (advertisements) across public and private nodes.

7.6.6 Robustness to Different Churn Patterns

We experiment with several churn patterns in simulation. Figure 7.8 shows our results with flash crowd scenarios, where we progressively decrease the number of peers that join at the beginning of the simulation while increasing the number of those that join at a later point in time. For instance, for a flash crowd scenario of 70% of the nodes, 300 nodes start to join at time 0 and the other 700 nodes start to join the system at minute 6.5. The number of nodes involved in all experiments stays constant at 1000. As we can observe in Figure 7.8(a), the hop count not only stabilizes quickly after the flash crowd terminates, but it also converges to the same value for all flash crowd scenarios.

The clustering coefficient exhibits a similar behavior, that is, it stabilizes quickly after the flash crowd to the same value for all flash crowd sizes. The converged clustering coefficient value (0.0975) is almost identical to the converged value of *gossip healer* (0.0960) [2] in a scenario with no churn. Figure 7.8(c) shows that the average indegree drops significantly during the flash crowd but it recovers quickly after that.

In Figure 7.9, we also show that the protocol is robust to catastrophic failures, that is, when a large number of peers leaves the system at a single instant in time. In this scenario, we wait for the overlay to stabilize after all the nodes have joined, then we fail a percentage of peers drawn uniformly from the set of all peers at time 5 minutes. It is



(a) Average hop count

(b) Average clustering coefficient

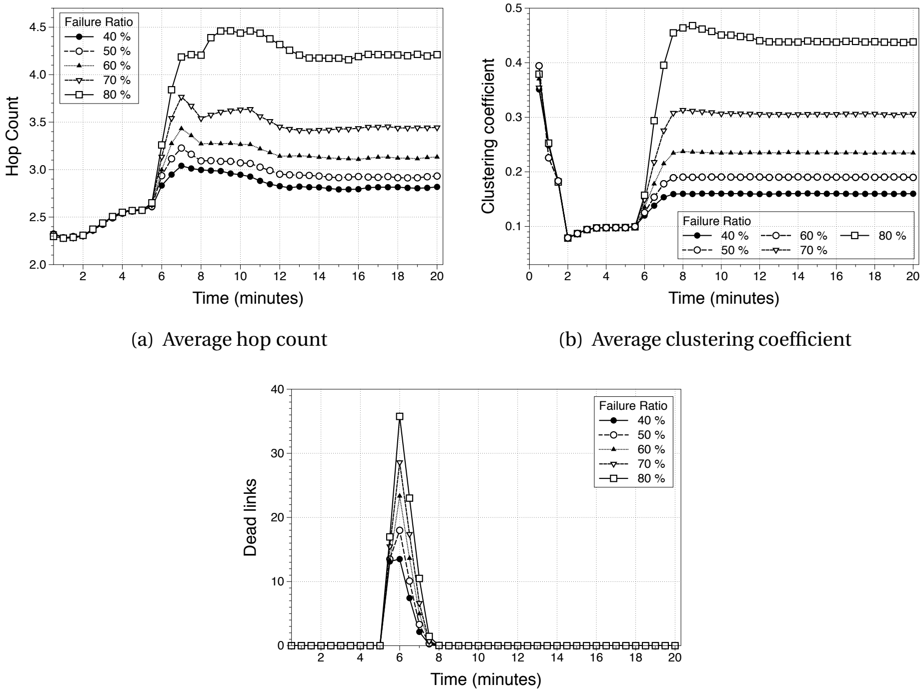
(c) Average in-degree

Figure 7.8: *Flash crowd scenario for different sizes of flash crowd.*

important to note that the overlay remains connected even after the failure of 80% of the nodes.

As we can observe in Figure 7.9(a), the average hop count stabilizes more slowly for higher percentages of failed nodes. This is expected given the large number of broken links to detect and repair for both the base and wormhole overlays. The clustering coefficient converges somewhat quicker. Note that the clustering coefficient stabilizes on higher values, because the number of remaining nodes is lower after the failure. More precisely, for both clustering coefficient and hop count, the converged values after the mass failure are the same as they would have been in a network that hadn't experienced mass failure but had the same size as our system had size after the mass failure. Two minutes after the mass failures, in all scenarios, dead links have been expelled from the view, as shown in Figure 7.10(c).

In Figure 7.10, we also show that the protocol is robust to different levels of steady churn, with up to one percent of the nodes failing and joining every wormhole refresh period (10 seconds). Figure 7.10(a) shows that for a level of churn of 0.1%, the hop count takes a longer time to stabilize compared to a scenario without churn, but it converges



(a) Average hop count

(b) Average clustering coefficient

(c) Average number of dead links in view

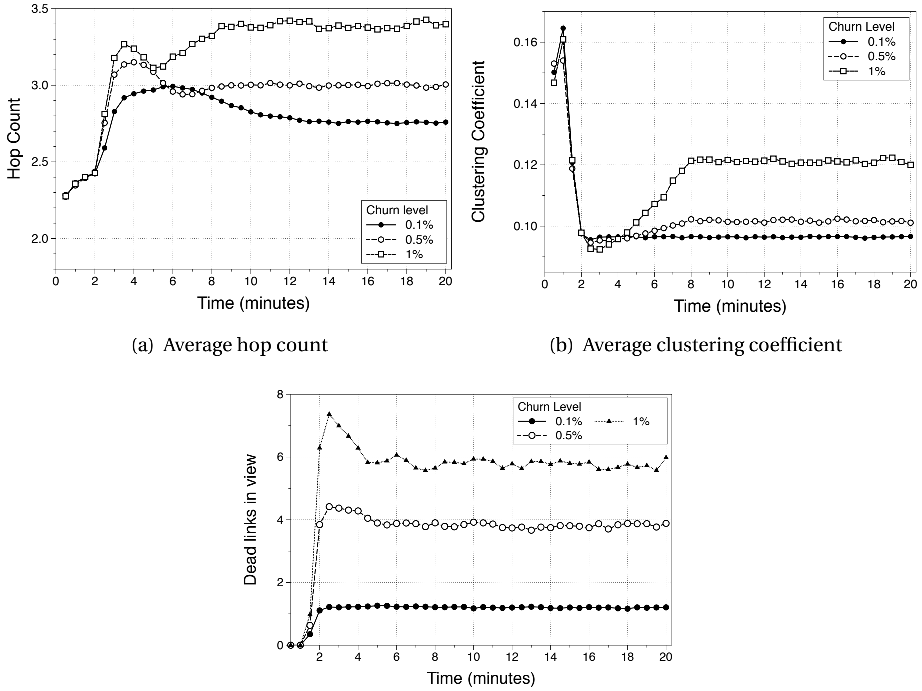
Figure 7.9: *Catastrophic failure scenarios in WPSS for different ratios of failed nodes.*

nevertheless to the same value as in the scenario without churn. For levels of churn of 0.5% and 1%, the average hop count increases by 7% and 21%, respectively, which is still acceptable in terms of freshness. Another effect of churn is the steady increase of the clustering coefficient, as shown in Figure 7.10(b).

In order to measure the extent of damage of continuous churn, we also show in Figure 7.10(c) the average number of dead links in a node's view. From the figure, it is clear that it is proportional to the level of churn.

7.6.7 Comparison with Croupier

We conclude our evaluation with a comparison with the state of the art NAT-aware PSS, Croupier. We use the same experiment setup in Croupier as in WPSS (the same number of nodes, ratio of public/private nodes and join distribution) with a view size of 50 in Croupier. We compare WPSS and Croupier using well-established properties for good PSSes, and we show that WPSS has better results for global randomness (through the network properties of the clustering coefficient, and the in-degree distribution of graph of samples), the freshness of samples, and how many hops node descriptors have to



(a) Average hop count

(b) Average clustering coefficient

(c) Average number of dead links in view

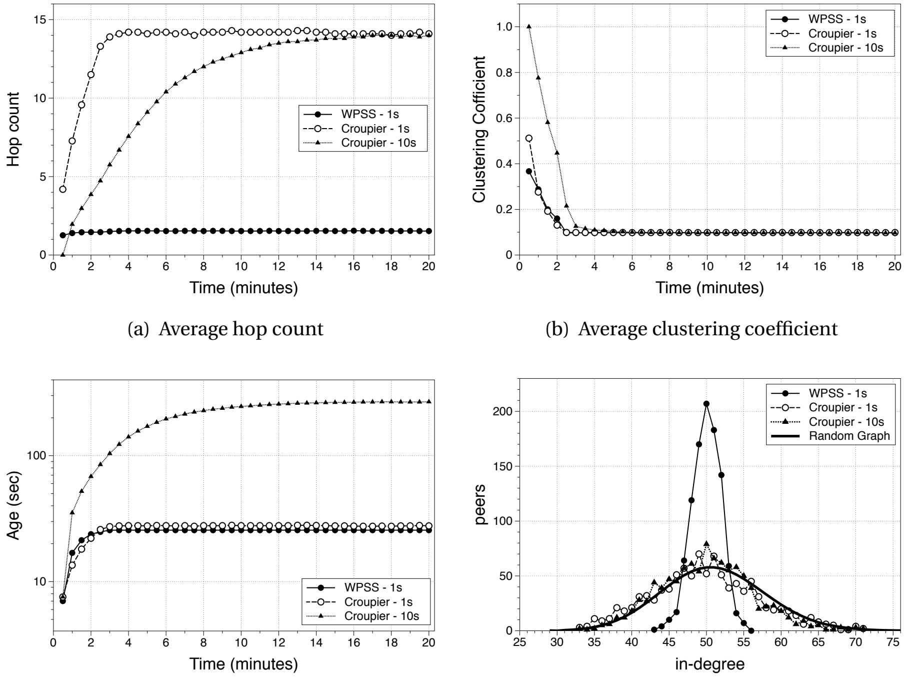
Figure 7.10: *WPSS under churn scenarios with varying levels of churn.*

traverse before being placed as a sample.

For a fair comparison with Croupier, we include two settings, one that creates new network connections at the same rate as WPSS (Croupier-10s) with the Croupier gossip round time is set to 10 seconds, and another setting that has the same round time as WPSS (Croupier-1s), but, for this setting, the network connection establishment rate is 10 times higher than WPSS. In Figure 7.11(b), we can see that the clustering coefficient of WPSS converges more quickly than in Croupier-10s, but at only a slightly faster rate than Croupier-1s. Both protocols have low clustering coefficients, close to random graphs. In Figure 7.11(d) we can see that WPSS has a much narrower in-degree distribution than Croupier around the expected in-degree, indicating better load balancing of samples around all the nodes.

In Figure 7.11(a), we can see that average hop count in WPSS is stable and low over time, while in Croupier the average hop count increases as node descriptors spread throughout the system until they finally are expired.

In Figure 7.11(c), we can see the average age (freshness) of samples generated by WPSS is roughly the same as Croupier-1s at 11.5 seconds, but much better than Croupier-10s. As Croupier is based on the swapper policy, from our earlier Table 7.2, we can see



(a) Average hop count

(b) Average clustering coefficient

(c) Average age of samples

(d) In-degree distribution

Figure 7.11: *Comparison between WPSS and Croupier.*

that the average freshness of its samples are close to Swapper's expected value of 10s, with the extra second resulting from the private nodes having to use an extra hop to public nodes who shuffle descriptors on behalf of private nodes.

7.7 Conclusions and Future Work

In this paper, we presented WPSS, a peer sampling service to meet the requirements of commercially deployed P2P systems on the Internet. WPSS executes short random walks over a stable topology and by using shortcuts (wormholes). WPSS provides the same level of sample freshness as other state-of-the-art protocols, but achieves that with a connection establishment rate that is one order of magnitude lower.

In addition, the connection establishment rate of WPSS can be tuned from zero per sample to up one per sample, according to the application requirements on freshness and cost. We showed in our deployed live-streaming P2P system that the lowest cost connection creation rate lies between these two extremes. On top top of that, we experimentally demonstrated that our system has the randomness properties required of a peer sampling service, while it is robust to churn and large-scale failure scenarios.

While we have designed WPSS for the Internet, we believe it is general enough to work over any type of stable base overlay (given a high enough TTL for RWs) and any subset of nodes can act as wormholes. As part of our future work, we will consider applying WPSS to mobile and sensor networks, and overlay networks without NATs or firewalls.

7.8 References

[1] Gunnar Kreitz and Fredrik Niemela. Spotify – large scale, low latency, P2P Music-on-Demand streaming. In *Tenth IEEE International Conference on Peer-to-Peer Computing (P2P'10)*. IEEE, August 2010. ISBN 978-1-4244-7140-9.

[2] Márk Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. Gossip-based peer sampling. *ACM Transactions on Computer Systems*, 25(3):8, August 2007.

[3] Daniel Stutzbach, Reza Rejaie, Nick Duffield, Subhabrata Sen, and Walter Willinger. On unbiased sampling for unstructured peer-to-peer networks. *IEEE/ACM Transactions on Networking*, 17(2):377–390, April 2009.

[4] Amir H. Payberah, Jim Dowling, and Seif Haridi. Gozar: Nat-friendly peer sampling with one-hop distributed nat traversal. In Pascal Felber and Romain Rouvoy, editors, *DAIS*, Lecture Notes in Computer Science, 2011.

[5] Jim Dowling and Amir H. Payberah. Shuffling with a croupier: Nat-aware peer-sampling. *ICDCS*, 0:102–111, 2012. ISSN 1063-6927.

[6] Anne-Marie Kermarrec, Alessio Pace, Vivien Quema, and Valerio Schiavoni. NAT-resilient gossip peer sampling. In *Proc. 29th IEEE Intl. Conf. on Distributed Computing Systems*, pages 360–367. IEEE Comp. Soc., 2009.

[7] Saikat Guha, Neil Daswani, and Ravi Jain. An Experimental Study of the Skype Peer-to-Peer VoIP System. In *Proceedings of IPTPS*, pages 1–6, Santa Barbara, CA, February 2006.

[8] A. Bergkvist, D.C. Burnett, C. Jennings, and A. Narayanan. Webrtc 1.0: Real-time communication between browsers. In *W3C Working Draft 21 August 2012*, 2012. URL `http://www.w3.org/TR/webrtc/`.

[9] Roberto Roverso, Sameh El-Ansary, and Seif Haridi. Smoothcache: Http-live streaming goes peer-to-peer. *Lecture Notes in Computer Science*, 7290:29–43, 2012.

[10] Ittay Eyal, Idit Keidar, and Raphael Rom. Limosense – live monitoring in dynamic sensor networks. In *Algorithms for Sensor Systems*, volume 7111 of *LNCS*, pages 72–85. Springer, 2012.

[11] Spyros Voulgaris, Daniela Gavidia, and Maarten Van Steen. Cyclon: Inexpensive membership management for unstructured p2p overlays. *Journal of Network and Systems Management*, 13:2005, 2005.

[12] Ayalvadi J. Ganesh, Anne-Marie Kermarrec, and Laurent Massoulié. Peer-to-peer membership management for gossip-based protocols. *IEEE Transactions on Computers*, 52(2), February 2003.

[13] J. Leitão, R. van Renesse, and L. Rodrigues. Balancing gossip exchanges in networks with firewalls. In Michael J. Freedman and Arvind Krishnamurthy, editors, *Proceedings of the 9th International Workshop on Peer-to-Peer Systems (IPTPS '10)*, page 7. USENIX, 2010.

[14] Roberto Roverso, Sameh El-Ansary, and Seif Haridi. NATCracker: NAT Combinations Matter. In *Proc. of 18th International Conference on Computer Communications and Networks*, ICCCN '09, SF, CA, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-4581-3.

[15] Daniel Stutzbach, Reza Rejaie, Nick Duffield, Subhabrata Sen, and Walter Willinger. On unbiased sampling for unstructured peer-to-peer networks. *IEEE/ACM Trans. Netw.*, 17(2):377–390, April 2009. ISSN 1063-6692.

[16] László Lovász and Peter Winkler. Mixing of random walks and other diffusions on a graph. In P. Rowlinson, editor, *Surveys in Combinatorics*, volume 218 of *London Math. Soc. Lecture Notes Series*, pages 119–154. Cambridge University Press, 1995. URL `http://research.microsoft.com/users/lovasz/survey.htm`.

[17] Siddhartha Chib and Edward Greenberg. Understanding the metropolis-hastings algorithm. *The American Statistician*, 49(4):327–335.

[18] Cosmin Arad, Jim Dowling, and Seif Haridi. Message-passing concurrency for scalable, stateful, reconfigurable middleware. In *ACM/IFIP/USENIX International Middleware Conference*, 2012.

[19] Roberto Roverso, Sameh El-Ansary, Alexandros Gkogkas, and Seif Haridi. Mesmerizer: a effective tool for a complete peer-to-peer software development life-cycle. In *SIMUTools '11*, 2011. ISBN 978-1-936968-00-8.

[20] Riccardo Reale, Roberto Roverso, Sameh El-Ansary, and Seif Haridi. DTL: Dynamic Transport Library for Peer-To-Peer Applications. In *In Proc. of the 12th International Conference on Distributed Computing and Networking*, ICDCN, Jan 2012.

PART V

Peer-To-Peer Streaming Agent Design

CHAPTER 8

SmoothCache: HTTP-Live Streaming Goes Peer-To-Peer

SmoothCache: HTTP-Live Streaming Goes Peer-To-Peer

Roberto Roverso[1,2], Sameh El-Ansary[1] and Seif Haridi[2]

[1]Peerialism AB
Stockholm, Sweden
[roberto,sameh]@peerialism.com

[2]KTH - Royal Institute of Technology
Stockholm, Sweden
[haridi]@kth.se

Abstract

In this paper, we present SmoothCache, a peer-to-peer live video streaming (P2PLS) system. The novelty of SmoothCache is threefold: i) It is the first P2PLS system that is built to support the relatively-new approach of using HTTP as the transport protocol for live content, ii) The system supports both single and multi-bitrate streaming modes of operation, and iii) In Smoothcache, we make use of recent advances in application-layer dynamic congestion control to manage priorities of transfers according to their urgency. We start by explaining why the HTTP live streaming semantics render many of the existing assumptions used in P2PLS protocols obsolete. Afterwards, we present our design starting with a baseline P2P caching model. We, then, show a number of optimizations related to aspects such as neighborhood management, uploader selection and proactive caching. Finally, we present our evaluation conducted on a real yet instrumented test network. Our results show that we can achieve substantial traffic savings on the source of the stream without major degradation in user experience.

8.1 Introduction

Peer-to-peer live streaming (P2PLS) is a problem in the Peer-To-Peer (P2P) networking field that has been tackled for quite some time on both the academic and industrial fronts. The typical goal is to utilize the upload bandwidth of hosts consuming a certain live content to offload the bandwidth of the broadcasting origin. On the industrial front, we find successful large deployments where knowledge about their technical approaches is rather limited. Exceptions include systems described by their authors like Coolstreaming [1] or inferred by reverse engineering like PPlive [2] and TVAnts [3]. On the academic front, there have been several attempts to try to estimate theoretical limits in terms of optimality of bandwidth utilization [4][5] or delay [6].

Traditionally, HTTP has been utilized for progressive download streaming, championed by popular Video-On-Demand (VoD) solutions such as Netflix [7] and Apple's iTunes movie store. However, lately, adaptive HTTP-based streaming protocols became the main technology for live streaming as well. All companies who have a major say in the market including Microsoft, Adobe and Apple have adopted HTTP-streaming as the main approach for live broadcasting. This shift to HTTP has been driven by a number of advantages such as the following: i) Routers and firewalls are more permissive to HTTP traffic compared to the RTSP/RTP ii) HTTP caching for real-time generated media is straight-forward like any traditional web-content iii) The Content Distribution Networks (CDNs) business is much cheaper when dealing with HTTP downloads [8].

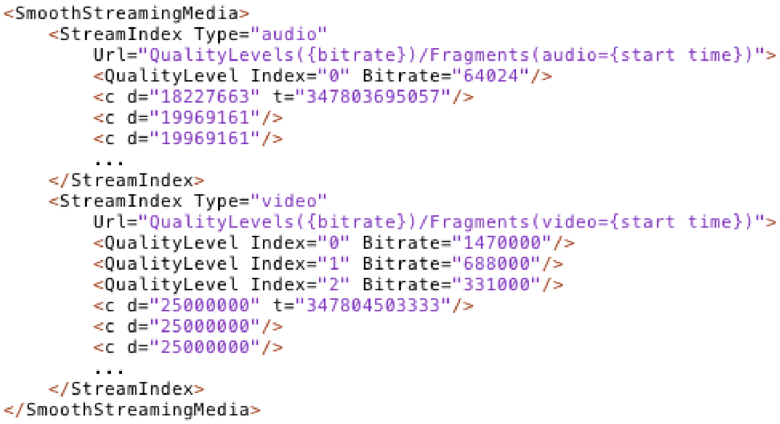
The first goal of this paper is to describe the shift from the RTSP/RTP model to the HTTP-live model (Section 8.2). This, in order to detail the impact of the same on the design of P2P live streaming protocols (Section 8.3). A point which we find rather neglected in the research community (Section 8.4). We argue that this shift has rendered many of the classical assumptions made in the current state-of-the-art literature obsolete. For all practical purposes, any new P2PLS algorithm irrespective of its theoretical soundness, won't be deployable if it does not take into account the realities of the mainstream broadcasting ecosystem around it. The issue becomes even more topical as we observe a trend in standardizing HTTP live [9] streaming and embedding it in all browsers together with HTML5, which is already the case in browsers like Apple's Safari.

The second goal of this paper is to present a P2PLS protocol that is compatible with HTTP live streaming for not only one bitrate but that is fully compatible with the concept of adaptive bitrate, where a stream is broadcast with multiple bitrates simultaneously to make it available for a range of viewers with variable download capacities (Section 8.5).

The last goal of this paper is to describe a number of optimizations of our P2PLS protocol concerning neighborhood management, uploader selection and peer transfer which can deliver a significant amount of traffic savings on the source of the stream (Section 8.6 and 8.7). Experimental results of our approach show that this result comes at almost no cost in terms of quality of user experience (Section 8.8).

8.2 The Shift from RTP/RTSP to HTTP

In the traditional RTSP/RTP model, the player uses RTSP as the signalling protocol to request the playing of the stream from a streaming server. The player enters a receive loop while the server enters a send loop where stream fragments are delivered to the receiver using the RTP protocol over UDP. The interaction between the server and player is stateful. The server makes decisions about which fragment is sent next based on ac-



```
<SmoothStreamingMedia>
    <StreamIndex Type="audio"
        Url="QualityLevels({bitrate})/Fragments(audio={start time})">
        <QualityLevel Index="0" Bitrate="64024"/>
        <c d="18227663" t="347803695057"/>
        <c d="19969161"/>
        <c d="19969161"/>
        ...
    </StreamIndex>
    <StreamIndex Type="video"
        Url="QualityLevels({bitrate})/Fragments(video={start time})">
        <QualityLevel Index="0" Bitrate="1470000"/>
        <QualityLevel Index="1" Bitrate="688000"/>
        <QualityLevel Index="2" Bitrate="331000"/>
        <c d="25000000" t="347804503333"/>
        <c d="25000000"/>
        <c d="25000000"/>
        ...
    </StreamIndex>
</SmoothStreamingMedia>
```

Figure 8.1: Sample Smooth streaming Manifest

knowledgements or error information previously sent by the client. This model makes the player rather passive, having the mere role of rendering the stream fragments which the server provides.

In the HTTP live streaming model instead, it is the player which controls the content delivery by periodically pulling from the server parts of the content at the time and pace it deems suitable. The server instead is entitled with the task of encoding the stream in real time with different encoding rates, or qualities, and storing it in data fragments which appear on the server as simple resources.

When a player first contacts the streaming server, it is presented with a metadata file (*Manifest*) containing the latest stream fragments available at the server at the time of the request. Each fragment is uniquely identified by a time-stamp and a bitrate. If a stream is available in n different bitrates, then this means that for each timestamp, there exists n versions of it, one for each bitrate.

After reading the manifest, the player starts to request fragments from the server. The burden of keeping the timeliness of the live stream is totally upon the player. The server in contrast, is stateless and merely serves fragments like any other HTTP server after encoding them in the format advertised in the manifest.

Manifest Contents. To give an example, we use Microsoft's Smooth Streaming manifest. In Figure 8.1, we show the relevant details of a manifest for a live stream with 3 video bitrates (331, 688, 1470 Kbps) and 1 audio bitrate (64 Kbps). By inspecting one of the streams, we find the first (the most recent) fragment containing a d value which is the time duration of the fragment in a unit of 100 nanoseconds and a t value which is the timestamp of the fragment. The fragment underneath (the older fragment) has only a d value because the timestamp is inferred by adding the duration to the timestamp of the one above. The streams each have a template for forming a request url for fragments of that stream. The template has place holders for substitution with an actual bitrate and timestamp. For a definition of the manifest's format, see [8].

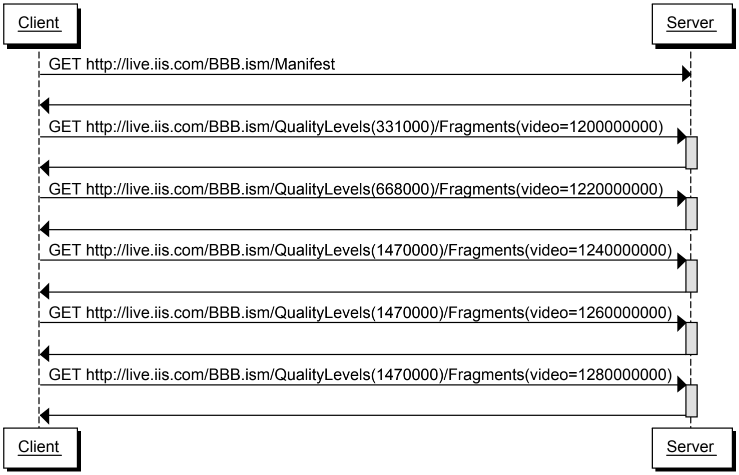

Client
Server
GET http://live.iis.com/BBB.ism/Manifest
GET http://live.iis.com/BBB.ism/QualityLevels(331000)/Fragments(video=1200000000)
GET http://live.iis.com/BBB.ism/QualityLevels(668000)/Fragments(video=1220000000)
GET http://live.iis.com/BBB.ism/QualityLevels(1470000)/Fragments(video=1240000000)
GET http://live.iis.com/BBB.ism/QualityLevels(1470000)/Fragments(video=1260000000)
GET http://live.iis.com/BBB.ism/QualityLevels(1470000)/Fragments(video=1280000000)
Client
Server

Figure 8.2: Client-Server interactions in Microsoft Smooth Streaming

Adaptive Streaming Protocol. In Figure 8.2, we show an example interaction sequence between a Smooth Streaming Client and Server [8]. The Client first issues a HTTP GET request to retrieve the manifest from the streaming server. After interpreting the manifest, the player requests a video fragment from the lowest available bitrate (331 Kbps). The timestamp of the first request is not predictable but in most cases we have observed that it is an amount equal to 10 seconds backward from the most recent fragment in the manifest. This is probably the only predictable part of the player's behavior.

In fact, without detailed knowledge of the player's internal algorithm and given that different players may implement different algorithms, it is difficult to make assumptions about the period between consecutive fragment requests, the time at which the player will switch rates, or how the audio and video are interleaved. For example, when a fragment is delayed, it could get re-requested at the same bitrate or at a lower rate. The timeout before taking such action is one thing that we found slightly more predictable and it was most of the time around 4 seconds. That is a subset of many details about the pull behavior of the player.

Implications of Unpredictability. The point of mentioning these details is to explain that the behavior of the player, how it buffers and climbs up and down the bitrates is rather unpredictable. In fact, we have seen it change in different version of the same player. Moreover, different adopters of the approach have minor variations on the interactions sequence. For instance, Apple HTTP-live [9] dictates that the player requests a manifest every time before requesting a new fragment and packs audio and video fragments together. As a consequence of what we described above, we believe that a P2PLS protocol for HTTP live streaming should operate as if receiving random requests in terms of timing and size and has to make this the main principle. This filters out the details of the different players and technologies.

8.3 Impact of the Shift on P2PLS Algorithms

Traditionally, the typical setup for a P2PLS agent is to sit between the streaming server and the player as a local proxy offering the player the same protocol as the streaming server. In such a setup, the P2PLS agent would do its best, exploiting the peer-to-peer overlay, to deliver pieces in time and in the right order for the player. Thus, the P2PLS agent is the one driving the streaming process and keeping an active state about which video or audio fragment should be delivered next, whereas the player blindly renders what it is supplied with. Given the assumption of a passive player, it is easy to envisage the P2PLS algorithm skipping for instance fragments according to the playback deadline, i.e. discarding data that comes too late for rendering. In this kind of situation, the player is expected to skip the missing data by fast-forwarding or blocking for few instants and then start the playback again. This type of behavior towards the player is an intrinsic property of many of the most mature P2PLS system designs and analyses such as [1, 6, 10].

In contrast to that, a P2PLS agent for HTTP live streaming can not rely on the same operational principles. There is no freedom in skipping pieces and deciding what is to be delivered to the player. The P2PLS agent has to obey the player's request for fragments from the P2P network and the speed at which this is accomplished affects the player's next action. From our experience, delving in the path of trying to reverse engineer the player behavior and integrating that in the P2P protocol is some kind of black art based on trial-and-error and will result into very complicated and extremely version-specific customizations. Essentially, any P2PLS scheduling algorithm that assumes that it has control over which data should be delivered to the player is rather inapplicable to HTTP live streaming.

8.4 Related Work

We are not aware of any work that has explicitly articulated the impact of the shift to HTTP on the P2P live streaming algorithms. However, a more relevant topic to look at is the behavior of the HTTP-based live players. Akhshabi et. al [11], provide a recent dissection of the behavior of three such players under different bandwidth variation scenarios. It is however clear from their analysis that the bitrate switching mechanics of the considered players are still in early stages of development. In particular, it is shown that throughput fluctuations still cause either significant buffering or unnecessary bitrate reductions. On top of that, it is shown how all the logic implemented in the HTTP-live players is tailored to TCP's behavior, as the one suggested in [12]. That in order to compensate throughput variations caused by TCP's congestion control and potentially large retransmission delays. In the case of a P2PLS agent acting as proxy, it is then of paramount importance to not interfere with such adaptation patterns.

We believe, given the presented approaches, the most related work is the P2P caching network LiveSky [13]. We share in common the fact of trying to establish a P2P CDN. However, LiveSky does not present any solution for supporting HTTP live streaming.

Strategy	**Baseline**	**Improved**
Manifest Trimming (*MT*)	Off	On
Partnership Construction (*PC*)	Random	Request-Point-aware
Partnership Maintenance (*PM*)	Random	Bitrate-aware
Uploader Selection (*US*)	Random	Throughput-based
Proactive Caching (*PR*)	Off	On

Table 8.1: Summary of baseline and improved strategies.

8.5 P2PLS as a Caching Problem

We will describe here our baseline design to tackle the new realities of the HTTP-based players. We treat the problem of reducing the load on the source of the stream the same way it would be treated by a Content Distribution Network (CDN): as a *caching problem*. The design of the streaming protocol was made such that every fragment is fetched as an independent HTTP request that could be easily scheduled on CDN nodes. The difference is that in our case, the caching nodes are consumer machines and not dedicated nodes. The player is directed to order from our local P2PLS agent which acts as an HTTP proxy. All traffic to/from the source of the stream as well as other peers passes by the agent.

Baseline Caching. The policy is as follows: any request for manifest files (metadata), is fetched from the source as is and not cached. That is due to the fact that the manifest changes over time to contain the newly generated fragments. Content fragments requested by the player are looked up in a local index of the peer which keeps track of which fragment is available on which peer. If information about the fragment is not in the index, then we are in the case of a P2P cache *miss* and we have to retrieve it from the source. In case of a cache *hit*, the fragment is requested from the P2P network and any error or slowness in the process results, again, in a fallback to the source of the content. Once a fragment is downloaded, a number of other peers are immediately informed in order for them to update their indices accordingly.

Achieving Savings. The main point is thus to increase the cache hit ratio as much as possible while the timeliness of the movie is preserved. The cache hit ratio is our main metric because it represents savings from the load on the source of the live stream. Having explained the baseline idea, we can see that, in theory, if all peers started to download the same uncached manifest simultaneously, they would also all start requesting fragments exactly at the same time in perfect alignment. This scenario would leave no time for the peers to advertise and exchange useful fragments between each others. Consequently a perfect alignment would result in no savings. In reality, we have always seen that there is an amount of intrinsic asynchrony in the streaming process that causes some peers to be ahead of others, hence making savings possible. However, the larger the number of peers, the higher the probability of more peers being aligned. We will show that, given the aforementioned asynchrony, even the previously described baseline design can achieve significant savings.

Our target savings are relative to the number of peers. That is we do not target achieving a constant load on the source of the stream irrespective of the number of users, which would lead to loss of timeliness. Instead, we aim to save a substantial percentage of all source traffic by offloading that percentage to the P2P network. The attractiveness of that model from a business perspective has been verified with content owners who nowadays buy CDN services.

8.6 Beyond Baseline Caching

We give here a description of some of the important techniques that are crucial to the operation of the P2PLS agent. For each such technique we provide what we think is the simplest way to realize it as well as improvements if we were able to identify any. The techniques are summarized in Table 1.

Manifest Manipulation. One improvement particularly applicable in Microsoft's Smooth streaming but that could be extended to all other technologies is manifest manipulation. As explained in Section 8.2, the server sends a manifest containing a list of the most recent fragments available at the streaming server. The point of that is to avail to the player some data in case the user decides to jump back in time. Minor trimming to hide the most recent fragments from some peers places them behind others. We use that technique to push peers with high upload bandwidth slightly ahead of others because they have they can be more useful to the network. We are careful not to abuse this too much, otherwise peers would suffer a high delay from live playing point, so we limit it to a maximum of 4 seconds. It is worth noting here that we do a quick bandwidth measurement for peers upon admission to the network, mainly, for statistical purposes but we do not depend on this measurement except during the optional trimming process.

Neighborhood & Partnership Construction. We use a tracker as well as gossiping for introducing peers to each other. Any two peers who can establish bi-directional communication are considered neighbors. Each peer probes his neighbors periodically to remove dead peers and update information about their last requested fragments. Neighborhood construction is in essence a process to create a random undirected graph with high node arity. A subset of the edges in the neighborhood graph is selected to form a directed subgraph to establish partnership between peers. Unlike the neighborhood graph, which is updated lazily, the edges of the partneship graph are used frequently. After each successful download of a fragment, the set of (*out-partners*) is informed about the newly downloaded fragment. From the opposite perspective, it is crucial for a peer to wisely pick his *in-partners* because they are the providers of fragments from the P2P network. For this decision, we experiment with two different strategies: i) Random picking, ii) Request-point-aware picking: where the in-partners include only peers who are relatively ahead in the stream because only such peers can have future fragments.

Partnership Maintenance. Each peer strives to continuously find *better* in-partners using periodic maintenance. The maintenance process could be limited to replacement of dead peers by randomly-picked peers from the neighborhood. Our improved maintenance strategy is to score the in-partners according to a certain metric and replace low-scoring partners with new peers from the neighborhood. The metric we use for scoring peers is a composite one based on: i) favoring the peers with higher percentage of successfully transferred data, ii) favoring peers who happen to be on the same bitrate. Note that while favoring peers on the same bitrate, having all partners from a single bitrate is very dangerous, because once a bit-rate change occurs the peer is isolated. That is, all the received updates about presence of fragments from other peers would be from the old bitrate. That is why, upon replacement, we make sure that the resulting in-partners set has all bit-rates with a gaussian distribution centered around the current bitrate. That is, most in-partners are from the current bit rate, less partners from the immediately higher and lower bit rates and much less partners from other bitrates and so forth. Once the bit-rate changes, the maintenance re-centers the distribution around the new bitrate.

Uploader Selection. In the case of a cache hit, it happens quite often that a peer finds multiple uploaders who can supply the desired fragment. In that case, we need to pick one. The simplest strategy would be to pick a random uploader. Our improved strategy here is to keep track of the observed historical throughput of the downloads and pick the fastest uploader.

Sub-fragments. Up to this point, we have always used in our explanation the fragment as advertised by the streaming server as the unit of transport for simplifying the presentation. In practice, this is not the case. The sizes of the fragment vary from one bitrate to the other. Larger fragments would result in waiting for a longer time before informing other peers which would directly entail lower savings because of the slowness of disseminating information about fragment presence in the P2P network. To handle that, our unit of transport and advertising is a sub-fragment of a fixed size. That said, the reality of the uploader selection process is that it always picks a set uploaders for each fragment rather than a single uploader. This parallelization applies for both random and throughput-based uploader selection strategies.

Fallbacks. While downloading a fragment from another peer, it is of critical importance to detect problems as soon as possible. The timeout before falling back to the source is thus one of the major parameters while tuning the system. We put an upper bound (T_{p2p}) on the time needed for any P2P operation, computed as: $T_{p2p} = T_{player} - S * T_f$ where T_{player} is the maximum amount of time after which the player considers a request for a fragment expired, S is the size of fragment and T_f is the expected time to retrieve a unit of data from the fallback. Based on our experience, T_{player} is player-specific and constant, for instance Microsoft's Smooth Streaming waits 4 seconds before timing out. A longer T_{p2p} translates in a higher P2P success transfer ratio, hence higher savings. Since T_{player} and S are outside of our control, it is extremely important to estimate T_f correctly, in particular in presence of congestion and fluctuating throughput towards the source. As a further optimization, we recalculate the timeout for a fragment while a P2P transfer is happening depending on the amount of data already downloaded, to allow more time for the outstanding part of the transfer. Finally, upon fallback, only the amount of fragment that failed to be downloaded from the overlay network is retrieved from the source, i.e. through a partial HTTP request on the range of missing data.

8.7 Proactive Caching

The baseline caching process is in essence reactive, i.e. the attempt to fetch a fragment starts *after* the player requests it. However, when a peer is informed about the presence of a fragment in the P2P network, he can trivially see that this is a future fragment that would be eventually requested. Starting to prefetch it early before it is requested, increases the utilization of the P2P network and decreases the risk of failing to fetch it in time when requested. That said, we do not guarantee that this fragment would be requested in the same bitrate, when the time comes. Therefore, we endure a bit of risk that we might have to discard it if the bitrate changes. In practice, we measured that the prefetcher successfully requests the right fragment with a 98.5% of probability.

Traffic Prioritization. To implement this proactive strategy we have taken advantage of our dynamic runtime-prioritization transport library DTL [14] which exposes to the application layer the ability to prioritize individual transfers relative to each other and to change the priority of each individual transfer at run-time. Upon starting to fetch a fragment proactively, it is assigned a very low-priority. The rationale is to avoid contending with the transfer process of fragments that are reactively requested and under a deadline both on the uploading and downloading ends.

Successful Prefetching. One possibility is that a low-priority prefetching process completes before a player's request and there is no way to deliver it to the player before that happens, the only option is to wait for a player request. More importantly, when that time comes, careful delivery from the local machine is very important because extremely fast delivery might make the adaptive streaming player mistakenly think that there is an abundance of download bandwidth and start to request the following fragments a higher bitrate beyond the actual download bandwidth of the peer. Therefore, we schedule the delivery from the local machine to be not faster than the already-observed average download rate. We have to stress here that this is not an attempt to control the player to do something in particular, we just maintain transparency by not delivering prefetched fragments faster than not prefetched ones.

Interrupted Prefetching. Another possiblity is that the prefetching process gets interrupted by the player in 3 possible ways: i) The player requests the fragment being prefetched: in that case the transport layer is dynamically instructed to raise the priority and T_{player} is set accordingly based on the remaining amount of data as described in the previous section. ii) The player requests the same fragment being prefetched but at a higher rate which means we have to discard any prefetched data and treat the request like any other reactively fetched fragment. iii) The player decides to skip some fragments to catch up and is no longer in need of the fragment being prefetched. In this case, we have to discard it as well.

8.8 Evaluation

Methodology. Due to the non-representative behaviour of Planetlab and the difficulty to do parameter exploration in publicly-deployed production network, we tried another approach which is to develop a version of our P2P agent that is remotely-controlled and ask for volunteers who are aware that we will conduct experiments on their machines. Needless to say, that this functionality is removed from any publicly-deployable version of the agent.

Test Network. The test network contained around 1350 peers. However, the maximum, minimum and average number of peers simultaneously online were 770, 620 and 680 respectively. The network included peers mostly from Sweden (89%) but also some from Europe (6%) and the US (4%). The upload bandwidth distribution of the network was as follows: 15% : $0.5Mbps$, 42% : $1Mbps$, 17% : $2.5Mbps$, 15% : $10Mbps$, 11% : $20Mbps$. In general, one can see that there is enough bandwidth capacity in the network, however the majority of the peers are on the lower end of the bandwidth distribution. For connectivity, 82% of the peers were behind NAT, and 12% were on open Internet. We have used our NAT-Cracker traversal scheme as described in [15] and were able to establish bi-directional communication between 89% of all peer pairs. The unique number of NAT types encountered were 18 types. Apart from the tracker used for introducing clients to each other, our network infrastructure contained, a logging server, a bandwidth measurement server, a STUN-like server for helping peers with NAT traversal and a controller to launch tests remotely.

Stream Properties. We used a production-quality continuous live stream with 3 video bitrates (331, 688, 1470 Kbps) and 1 audio bitrate (64 Kbps) and we let peers watch 20 minutes of this stream in each test. The stream was published using Microsoft Smooth Streaming traditional tool chain. The bandwidth of the source stream was provided by a commercial CDN and we made sure that it had enough capacity to serve the maximum number of peers in our test network. This setup gave us the ability to compare the quality of the streaming process in the presence and absence of P2P caching in order to have a fair assessment of the effect of our agent on the overall quality of user experience. We stress that, in a real deployment, P2P caching is not intended to eliminate the need for a CDN but to reduce the total amount of paid-for traffic that is provided by the CDN. One of the issues that we faced regarding realistic testing was making sure that we are using the actual player that would be used in production, in our case that was the Microsoft Silverlight player. The problem is that the normal mode of operation of all video players is through a graphical user interface. Naturally, we did not want to tell our volunteers to click the “Play” button every time we wanted to start a test. Luckily, we were able to find a rather unconventional way to run the Silverlight player in a headless mode as a background process that does not render any video and does not need any user intervention.

Reproducibility. Each test to collect one data point in the test network happens in real time and exploring all parameter combination of interest is not feasible. Therefore, we did a major parameter combinations study on our simulation platform [16] first to get a set of worth-trying experiments that we launched on the test network. Another

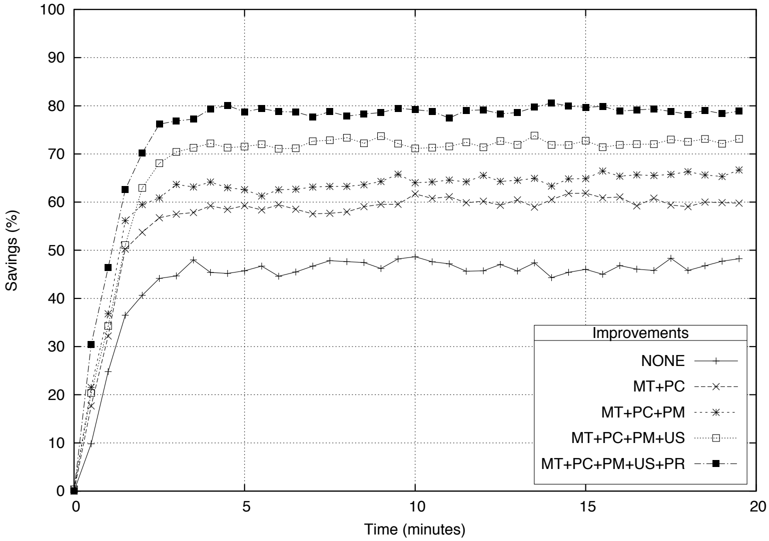


Figure 8.3: Comparison of traffic savings with different algorithm improvements

problem is the fluctuation of network conditions and number of peers. We repeated each data point a number of times before gaining confidence that this is the average performance of a certain parameter combination.

Evaluation Metrics. The main metric that we use is *traffic savings* defined as the percentage of the amount of data served from the P2P network from the total amount of data consumed by the peers. Every peer reports the amount of data served from the P2P network and streaming source every 30 seconds to the logging server. In our bookkeeping, we keep track of how much of the traffic was due to fragments of a certain bitrate. The second important metric is *buffering delay*. The Silverlight player can be instrumented to send debug information every time it goes in/out of buffering mode, i.e. whenever the player finds that the amount of internally-buffered data is not enough for playback, it sends a debug message to the server, which in our case is intercepted by the agent. Using this method, a peer can report the lengths of the periods it entered into buffering mode in the 30 seconds snapshots as well. At the end of the stream, we calculate the sum of all the periods the player of a certain peer spent buffering.

8.8.1 Deployment Results

Step-by-Step Towards Savings. The first investigation we made was to start from the baseline design with all the strategies set to the simplest possible. In fact, during the development cycle we used this baseline version repeatedly until we obtained a stable product with predictable and consistent savings level before we started to enable all the other improvements. Figure 8.3 shows the evolution of savings in time for all strategies.

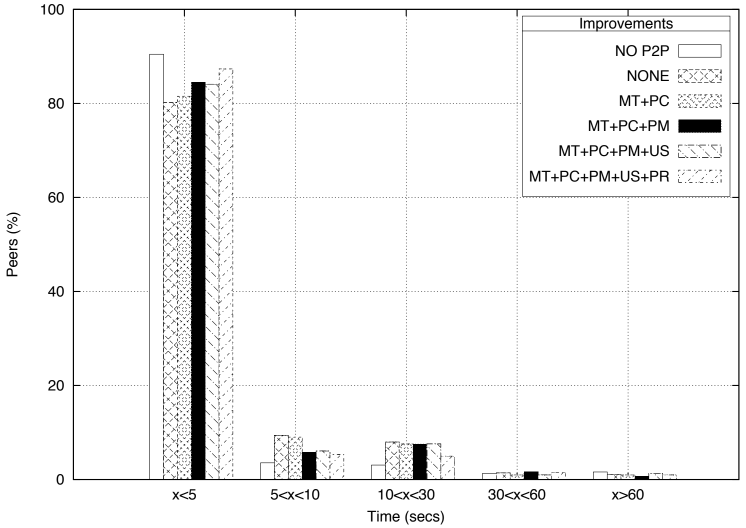


Figure 8.4: Comparison of cumulative buffering time for source only and improvements

The naive baseline caching was able to save a total of 44% of the source traffic. After that, we worked on pushing the higher-bandwidth peers ahead and making each partner select peers that are useful using the request-point-aware partnership which moved the savings to a level of 56%. So far, the partnership maintenance was random. Turning on bit-rate-aware maintenance added only another 5% of savings but we believe that this is a key strategy that deserves more focus because it directly affects the effective partnership size of other peers from each bitrate which directly affects savings. For the uploader selection, running the throughput-based picking achieved 68% of savings. Finally, we got our best savings by adding proactive caching which gave us 77% savings.

User Experience. Getting savings alone is not a good result unless we have provided a good user experience. To evaluate the user experience, we use two metrics: First, the percentage of peers who experienced a total buffering time of less than 5 seconds, i.e. they enjoyed performance that did not really deviate much from live. Second, showing that our P2P agent did not achieve this level of savings by forcing the adaptive streaming to move everyone to the lowest bitrate. For the first metric, Figure 8.4 shows that with all the improvements, we can make 87% of the network watch the stream with less than 5 seconds of buffering delay. For the second metric, Figure 8.5(a) shows also that 88% of all consumed traffic was on the highest bitrate and P2P alone shouldering 75% (Figure 8.5(b)), an indication that, for the most part, peers have seen the video at the highest bitrate with a major contribution from the P2P network.

P2P-less as a Reference. We take one more step beyond showing that the system offers substantial savings with reasonable user experience, namely to understand what would be the user experience in case all the peers streamed directly from the CDN.

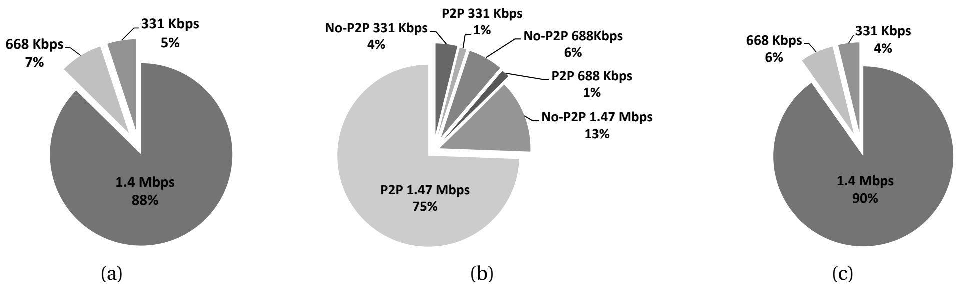


Figure 8.5: Breakdown of traffic quantities per bitrate for: (a) A network with P2P caching, Source & P2P traffic summed together. (b) The same P2P network with source & P2P traffic reported separately, and (c) A network with no P2P.

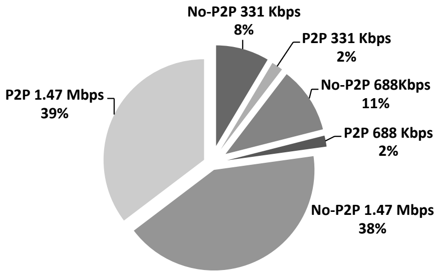


Figure 8.6: Breakdown of traffic quantities per bitrate using baseline

Therefore, we run the system with P2P caching disabled. Figure 8.4 shows that without P2P, only 3% more (90%) of all viewers would have a less than 5 seconds buffering. On top of that, Figure 8.5(c) shows that only 2% more (90%) of all consumed traffic is on the highest bitrate, that is the small price we paid for saving 77% of source traffic. Figure 8.6 instead describes the lower performance of our baseline caching scenario, which falls 13% of the P2P-less scenario (77%). This is mainly due to the lack of bit-rate-aware maintenance, which turns out to play a very significant role in terms of user experience.

Partnership Size. There are many parameters to tweak in the protocol but, in our experience, the number of in-partners is by far the parameter with the most significant effect. Throughout the evaluation presented here, we use 50 in-partners. Figure 8.7 shows that more peers result in more savings; albeit with diminishing returns. We have selected 50-peers as a high-enough number, at a point where increasing the peers does not result into much more savings.

Single Bitrate. Another evaluation worth presenting as well is the case of a single bitrate. In this experiment, we get 84%, 81% and 69% for the low, medium and high bitrate respectively (Figure 8.8). As for the user experience compared with the same single bitrates in a P2P-less test, we find that the user experience expressed as delays is much closer to the P2P-less network (Figure 8.9). We explain the relatively better experience in

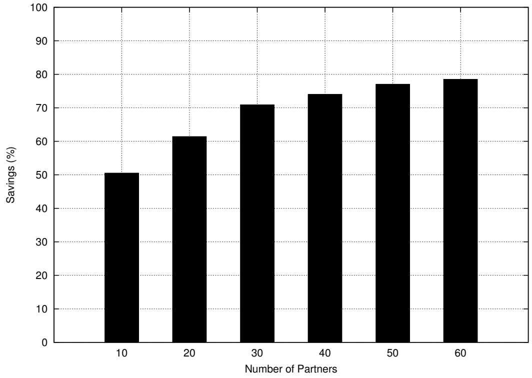


Figure 8.7: Comparison of savings between different in-partners number

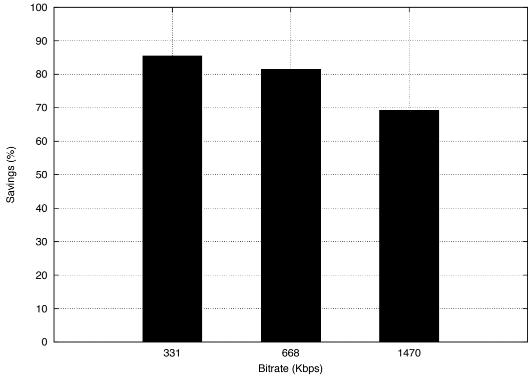


Figure 8.8: Savings for single bitrate runs

the single bitrate case by the fact that all the in-partners are from the same bitrate, while in the multi-bitrate case, each peer has in his partnership the majority of the in-partners from a single bitrate but some of them are from other bitrates which renders the effective partnership size smaller. We can also observe that the user experience improves as the bitrate becomes smaller.

8.9 Conclusion

In this paper, we have shown a novel approach in building a peer-to-peer live streaming system that is compatible with the new realities of the HTTP-live. These new realities revolve around the point that unlike RTSP/RTP streaming, the video player is driving the streaming process. The P2P agent will have a limited ability to control what gets rendered on the player and much limited ability to predict its behaviour. Our approach was to start with baseline P2P caching where a P2P agent acts as an HTTP proxy that re-

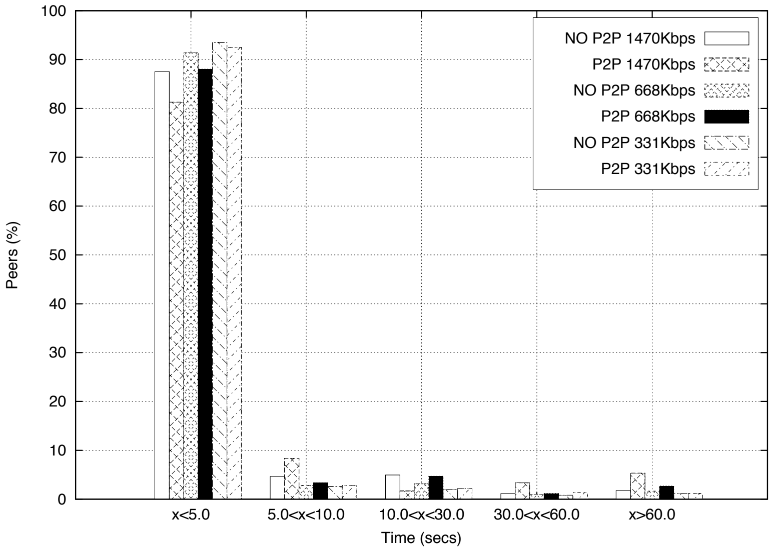


Figure 8.9: Buffering time for single bitrate runs

ceives requests from the HTTP live player and attempts to fetch it from the P2P network rather the source if it can do so in a reasonable time.

Beyond baseline caching, we presented several improvements that included: a) Request-point-aware partnership construction where peers focus on establishing relationships with peers who are ahead of them in the stream, b) Bit-rate-aware partnership maintenance through which a continuous updating of the partnership set is accomplished both favoring peers with high successful transfers rate and peers who are on the same bitrate of the maintaining peer, c) Manifest trimming which is a technique for manipulating the metadata presented to the peer at the beginning of the streaming process to push high-bandwidth peers ahead of others, d) Throughput-based uploader selection which is a policy used to pick the best uploader for a certain fragment if many exist, e) Careful timing for falling back to the source where the previous experience is used to tune timing out on P2P transfers early enough thus keeping the timeliness of the live playback.

Our most advanced optimization was the introduction of proactive caching where a peer requests fragments ahead of time. To accomplish this feature without disrupting the already-ongoing transfer, we used our application-layer congestion control [14] to make pre-fetching activities have less priority and dynamically raise this priority in case the piece being pre-fetched got requested by the player.

We evaluated our system using a test network of real volunteering clients of about 700 concurrent nodes where we instrumented the P2P agents to run tests under different configurations. The tests have shown that we could achieve around 77% savings for a multi-bitrate stream with around 87% of the peers experiencing a total buffering delay of less than 5 seconds and almost all of the peers watched the data on the highest bitrate. We compared these results with the same network operating in P2P-less mode and found that only 3% of the viewers had a better experience without P2P which we judge as a very limited degradation in quality compared to the substantial amount of savings.

8.10 References

[1] Xinyan Zhang, Jiangchuan Liu, Bo Li, and Y. S. P. Yum. CoolStreaming/DONet: a data-driven overlay network for peer-to-peer live media streaming. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*

[2] X. Hei, C. Liang, J. Liang, Y. Liu, and K. W. Ross. Insights into PPLive: A Measurement Study of a Large-Scale P2P IPTV System. In *Proc. of IPTV Workshop, International World Wide Web Conference*, 2006.

[3] Thomas Silverston and Olivier Fourmaux. P2P IPTV measurement: a case study of TVants. In *Proceedings of the 2006 ACM CoNEXT conference*, CoNEXT '06, pages 45:1–45:2, New York, NY, USA, 2006. ACM. ISBN 1-59593-456-1. URL `http://doi.acm.org/10.1145/1368436.1368490`.

[4] Yang Guo, Chao Liang, and Yong Liu. AQCS: adaptive queue-based chunk scheduling for P2P live streaming. In *Proceedings of the 7th IFIP-TC6 NETWORKING*, 2008.

[5] L. Massoulie, A. Twigg, C. Gkantsidis, and P. Rodriguez. Randomized Decentralized Broadcasting Algorithms. In *INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE*, pages 1073 –1081, may 2007.

[6] Meng Zhang, Qian Zhang, Lifeng Sun, and Shiqiang Yang. Understanding the Power of Pull-Based Streaming Protocol: Can We Do Better? In *Selected Areas in Communications, IEEE Journal on*, volume 25, pages 1678 –1694, december 2007.

[7] Netflix inc. www.netflix.com.

[8] Microsoft Inc. Smooth Streaming. http://www.iis.net/download/SmoothStreaming.

[9] R. Pantos. HTTP Live Streaming, December 2009. URL `\http://tools.ietf.org/html/draft-pantos-http-live-streaming-01`.

[10] A. Vlavianos, M. Iliofotou, and M. Faloutsos. BiToS: Enhancing BitTorrent for Supporting Streaming Applications. In *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings*, pages 1 –6, april 2006.

[11] Saamer Akhshabi, Ali C. Begen, and Constantine Dovrolis. An experimental evaluation of rate-adaptation algorithms in adaptive streaming over HTTP. In *Proceedings of the second annual ACM conference on Multimedia systems*, MMSys, 2011.

[12] Chenghao Liu, Imed Bouazizi, and Moncef Gabbouj. Parallel Adaptive HTTP Media Streaming. In *Computer Communications and Networks (ICCCN), Proc. of 20th International Conference on*, pages 1 –6, 31 2011-aug. 4 2011.

[13] Hao Yin, Xuening Liu, Tongyu Zhan, Vyas Sekar, Feng Qiu, Chuang Lin, Hui Zhang, and Bo Li. Livesky: Enhancing cdn with p2p. *ACM Trans. Multimedia Comput. Commun. Appl.*, 6:16:1–16:19, August 2010. ISSN 1551-6857. URL `http://doi.acm.org/10.1145/1823746.1823750`.

[14] Riccardo Reale, Roberto Roverso, Sameh El-Ansary, and Seif Haridi. DTL: Dynamic Transport Library for Peer-To-Peer Applications. In *In Proc. of the 12th International Conference on Distributed Computing and Networking*, ICDCN, Jan 2012.

[15] Roberto Roverso, Sameh El-Ansary, and Seif Haridi. NATCracker: NAT Combinations Matter. In *Proc. of 18th International Conference on Computer Communications and Networks*, ICCCN '09, SF, CA, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-4581-3.

[16] Roberto Roverso, Sameh El-Ansary, Alexandros Gkogkas, and Seif Haridi. Mesmerizer: A effective tool for a complete peer-to-peer software development life-cycle. In *In Proceedings of SIMUTOOLS*, March 2011.

CHAPTER 9

SmoothCache 2.0: CDN-quality adaptive HTTP live streaming on P2P overlays

SmoothCache 2.0: the HTTP-Live peer-to-peer CDN[1]

Roberto Roverso[1,2], Riccardo Reale[1], Sameh El-Ansary[1] and Seif Haridi[2]

[1]Peerialism AB
Stockholm, Sweden
[roberto,riccardo,sameh]@peerialism.com

[2]KTH - Royal Institute of Technology
Stockholm, Sweden
[haridi]@kth.se

Abstract

In recent years, adaptive HTTP streaming protocols have become the de-facto standard in the industry for the distribution of live and video-on-demand content over the Internet. This paper presents SmoothCache 2.0, a distributed cache for adaptive HTTP live streaming content based on peer-to-peer (P2P) overlays. The contribution of this work is twofold. From a systems perspective, to the best of our knowledge, it is the only commercially deployed P2P platform which supports recent live streaming protocols based on HTTP as a transport and the concept of adaptive bitrate switching. From an algorithmic perspective, the system describes a novel set of overlay construction and prefetching techniques that realize: *i*) substantial savings in terms of the bandwidth load on the source of the stream, and *ii*) CDN-quality user experience in terms of delay from the live playing point and the mostly-watched bitrate. In order to support our claims, we conduct a methodical evaluation encompassing a subset of thousands of real customers' machines which who agreed to install an instrumented version of our software.

9.1 Introduction

Peer-to-Peer live streaming (PLS) is the problem of improving distribution efficiency for a given online live broadcast by letting viewers of the content contribute with their resources, namely bandwidth.

[1] Submitted to a conference and currently under review.

Previous work in this area has produced a lot of systems such as PPlive [1], CoolStreaming [2], T-bone [3], and more recently Bittorrent live[4]. Overlay construction for such applications falls mainly into 2 classes, tree-based and mesh-based systems. Although a consensus has not been reached on the best approach, mesh-based approaches lately became more popular as they are more robust to churn [5, 6], while tree-based overlays are more cumbersome to maintain in the same conditions [7].

Despite the increased maturity in the field, an important aspect which we find relatively under-studied is the ramification of the recent trend of using HTTP-live streaming instead of the traditional RTSP/RTP protocol for broadcast of live video content over the Internet. HTTP-live streaming consists of a set of protocols which all utilize the HTTP protocol as transport [8]. HTTP-live streaming changes the way the producer of the content (usually a CDN) and the consumer of the content (the media player) interact. Traditional streaming protocols such as the RTSP/RTP are based on a push-based model, where the player requests a certain stream and then the server pushes content over UDP to the player controlling the speed of the delivery. HTTP-live streaming protocols are instead based on a pull-model, where it is the player which requests content chunks over HTTP at the pace it deems suitable. On top of that, the HTTP-live protocols have been designed to support adaptive bitrate mode of operation, which avails the stream in a number of qualities. The choice of which quality to retrieve/reproduce is left to the player.

All major actors in the online broadcasting business, such as Microsoft, Adobe and Apple, have developed technologies which embrace HTTP-streaming and the concept of adaptive bitrate switching as the main approach for broadcasting. HTTP live has been adopted by content services and creators like Netflix, Hulu and the BBC with support across all platforms and OSs, including computers, tablets and smart phones.

The shift from the push-based RTSP/RTP protocol to the pull-based HTTP-live protocols has rendered many of the classical assumptions made in the current state-of-the-art PLS literature obsolete [9]. For all practical purposes, any new PLS algorithm, irrespective of its theoretical soundness, won't be deployable if it does not take into account the realities of the mainstream broadcasting ecosystem around it.

Being an industrial entity, our attention to this area grew as we started to find it difficult to raise any reasonable market interest while overlooking the fact that both the CDNs and the live streaming solution vendors are focusing on HTTP-based solutions. Our first take on the problem of designing a distributed caching system tailored to HTTP live streaming [9] showed that it is possible to achieve significant savings towards the source of the stream by using peer-to-peer overlays.

In this iteration, we aim higher by not only targeting high savings but also making sure viewers experience CDN-like QoE in terms of delay from the live point. In order to achieve our goal, we exploit detailed knowledge of the HTTP live streaming distribution chain. Based on that, we build a distributed caching solution which prefetches content ahead of the live deadline and distributes it to all nodes in the network such as to significantly decrease the bandwidth requirements on the source of the stream.

The distribution of data happens over a self-organizing overlay network that takes into account many factors such as upload bandwidth capacity, connectivity constraints, performance history, prefetching point and currently watched bitrate all of which work together to maximize flow of fragments in the network. In the description of our system, we focus primarily on how our prefetching techniques work and on the overlay construction heuristics. On top of that, we conduct a thorough evaluation of our overlay techniques on a real network of thousands of consumer machines and show how each of our techniques individually contributes to the performance. We also show empirically that the resulting system provides the same QoE of a standard CDN, with respect to cumulative buffering time and mostly watched bitrate.

9.2 Related Work

First, an important high-level distinguishing feature of SmoothCache from classical peer-to-peer solutions is that, in all other systems, bounded source bandwidth is assumed and the challenge is to maximize throughput, but nothing prevents the maximum delay from the live point to grow with the network size [10]. Our industrial experience has led us to believe that commercial content owners are not willing to compromise on quality of user experience. In contrast, we have designed this system to achieve a bounded delay that is no worse than a CDN. That is, we have *first* to meet the same targets of throughput and playback delay of a CDN, and *then* try to maximize savings. This concept is found both in *peer-assisted* and *cloud-assisted* systems. The former category includes systems which build their own ad-hoc infrastructure to aid the peer-to-peer overlay, while the latter category use existing infrastructure such as Content Delivery Networks for the same purpose. While the approaches may differ, both categories of systems share the same goal of enforcing a specific target quality of user experience for viewers by using the least amount of infrastructure resources.

Cloud-assisted systems, such as LiveSky [11] and CALMS [12], utilize cloud-based dynamically allocated resources to support the peer-to-peer delivery. The main drawback of cloud-assisted solutions is that they are cumbersome to deploy in existing live streaming systems that are already relying on CDN services. They in fact require a complete re-engineering of the server-side of the system to allow bidirectional feedback from the peer-to-peer overlay and a centralized coordinator which controls the cloud resources. On top of that, in cloud-assisted systems a completely new management layer must be deployed for the coordinator to enforce its cloud allocation decisions.

SmoothCache 2.0 belongs instead to the peer-assisted category, which leverages the inherent ability of CDN networks to dynamically scale up and down resources and therefore cope with variable load from the peer-to-peer overlay. Dynamic scalability is one of the main features of CDNs because in such systems it is very common to experience dramatically varying demand for video content depending on factors such as increased/decreased popularity and time of the day.

The peer-assisted approach has been applied mostly to video on demand (VoD) streaming rather than live streaming. In the industry, we find instances of peer-assisted VoD systems in Akamai's NetSession [13] and Adobe's Flash framework [14][15].

In academia instead, efforts have been directed at understanding in general, and without considering the details of streaming protocols, what are the best prefetch policies for peers to download VoD content ahead of playback such that the load on the source of the stream is minimized [16][17]. Recent work instead proposed a practical solution to enable peer-assisted distribution of VoD content with adaptive bitrate HTTP streaming protocols using plugin-less browser-based technologies with Maygh [18]. Authors show that Maygh can save up to 75% of the load on the CDN in the considered scenarios.

We believe, given the presented approaches, that SmoothCache 2.0 is the first peer-assisted *live streaming* system to be designed with HTTP adaptive streaming in mind. Other systems have addressed the problem of adaptive streaming in the past by using Layered Video Coding (LC) [19] or Multiple Descriptor Coding (MDC) [20]. There are different LC and MDC techniques in the literature, all of which come at the expense of varying levels of added overhead, in terms of redundancy, computational complexity and reconstruction performance. To the best of our knowledge, there is no wide adoption of either technique in mainstream streaming platforms. On top of that, in SmoothCache 2.0 we show that HTTP adaptive streaming is achievable without data coding techniques and therefore without incurring the associated overhead.

Regarding live P2P streaming in general, Peerstreamer[21] is one of the actively-maintained mesh-based systems. However, it is not integrated with existing commercial live streaming protocols such as [22, 23, 24], and it does not support adaptive bitrate streaming.

In order to guide our design, we also investigated theoretical work which builds on the peer-assisted paradigm. Liu et Al. [25] assert the benefits of peer-assisted delivery for single bitrate streams by providing a theoretical framework which highlights the trade-offs between three fundamental metrics: savings towards the source of the stream, bitrate watched by peers and number of hops from the source of the stream. The model is however applicable only to tree-based overlay networks deployed on always-on set-top-boxes and therefore absent of churn. Mansy et al. [26] describe a model for designing and operating a peer-assisted system. Specifically, the model estimates the minimum amount of peers which should be served by the CDN and also which bitrate should peers download given the CDN and peers bandwidth capacity. The significant limitation of this work is that it does not model bitrate switching at the player side, which is a fundamental characteristic of adaptive HTTP streaming protocols. In the model, peers are not allowed to switch bitrate according to the observed throughput but rather it is the CDN which controls which bitrate they are watching. We are not aware of any other theoretical work which considers both the peer-assisted paradigm and adaptive HTTP streaming principles.

9.3 System Architecture

SmoothCache 2.0 is composed of a client (the PLS agent) which is installed on the viewer's machines, and by a number of helper services which represent our operational centralized infrastructure. It is important to note that our system is designed to be transparent to already-existing streaming infrastructure in order to ensure both vendor neutrality as well as simplified deployment. Therefore, we do not operate our own stream origin. We assume that there is an already-published stream, for instance from a Content Distribution Network (CDN), that can be played directly from its source and our customers would transparently use our system to minimize the load on that source. That said, we solve the problem of reducing the load on the source of the stream the same way a CDN would: by building a *caching* scheme.

9.3.1 Baseline caching

The basic idea of our system is to build a random graph between all nodes watching a stream. Upon a content fragment request from the player running on the peer, the PLS agent tries to timely retrieve the data requested from other peers in the overlay. If a fragment cannot be retrieved from any other peer on time, the agent downloads the missing portion of the fragment from the source of the stream, i.e. the CDN. By falling back to the source, we guarantee that all fragments are delivered on time even if the overlay network cannot retrieve such fragments, thereby guaranteeing the desired level of QoE.

9.3.2 PLS Agent

In the absence of our PLS agent, the HTTP live streaming process starts from the video encoder which outputs video “fragments” of constant duration δ as they are “born” in real time. Each fragment is created in multiple qualities (bitrates) simultaneously. Therefore, a fragment is uniquely identified by a timestamp and a bitrate.The fragments are published on a standard web-server. In addition to that, a constantly-changing manifest file is updated to contain the newly-born fragment.

To start playing a stream, a player makes an HTTP request to fetch the manifest, learns what is the latest fragment, then makes a second request to access the fragment with the highest timestamp (the most recent) in the manifest and continues retrieving a piece every δ seconds. Since the stream is usually CDN hosted, therefore, based on the throughput of the CDN, the player jumps to a higher bitrate or a lower bitrate starting initially from the lowest. The bitrate switches are performed to make sure the player is not drifting from the live playing point.

In the presence of our PLS agent, all player requests, are redirected to the local PLS agent instead of going directly to the source of the stream and that is the only change that needs to occur compared to normal operation without our agent.

Internally, as illustrated in Figure 9.1, the PLS agent has an *HTTP server* which receives sequential HTTP requests, detects which protocol they are made for, that being Apple’s HTTP Live Streaming (HLS)[22], Adobe’s HDS[23] or Microsoft’s Smooth Streaming (SS)[24], and forwards them to a protocol-specific component which extracts relevant information and generates protocol-neutral requests to the *Distributed Cache* component. The latter decides whether a request can be served from the source or from the P2P network using a *Fragment Index*, that keeps track of fragments available

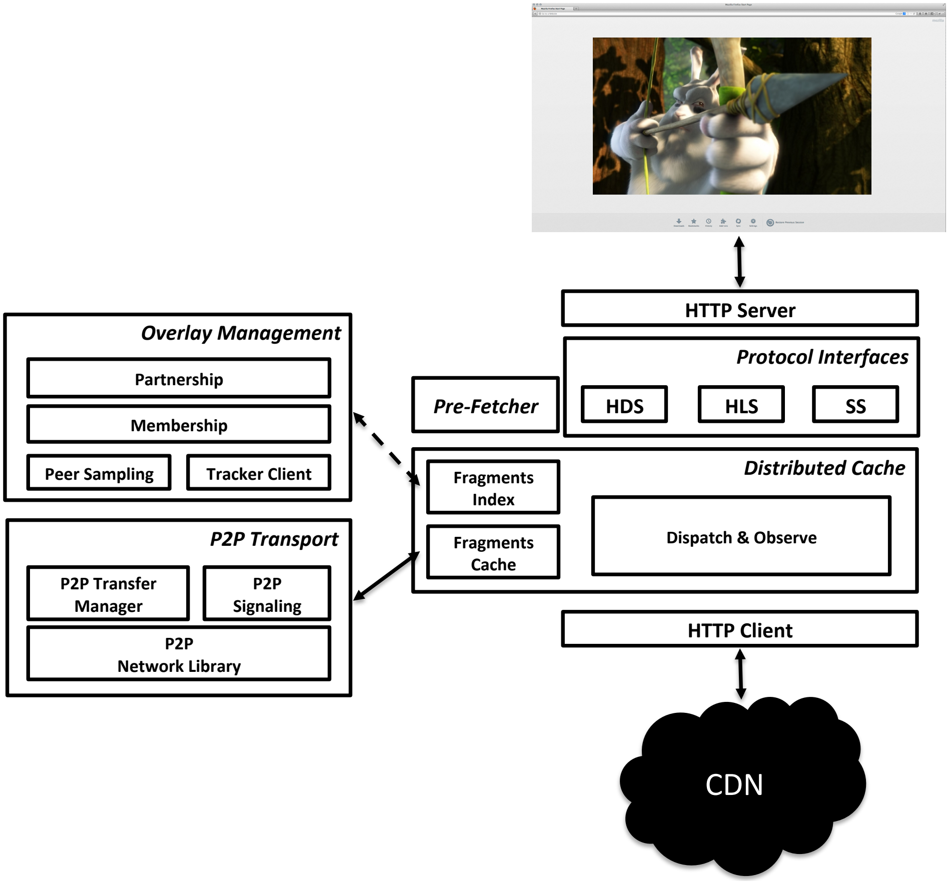


Figure 9.1: PLS agent architecture

on other peers. Manifest files are always retrieved from the source and are never cached since they are updated frequently and they contain live information about the newly released fragments.

The process described above is of a reactive nature, that is all requests to the *Distributed Cache* come from the player after protocol-specific translation. In addition to this, SmoothCache 2.0 implements a process which pro-actively downloads fragments ahead of time and before they are requested by the player. The component responsible for this operation is called the *Pre-Fetcher*.

To implement the distributed cache abstraction described above, a set of components for *Overlay management* work together to maintain the agent's neighborhood and update the fragment index. For efficient retrieval of data from other parties, an *HTTP client* maintains a keep-alive connection to the source of the stream, while a P2P transport layer utilizes state of the art NAT traversal [27] and a variable priority application-

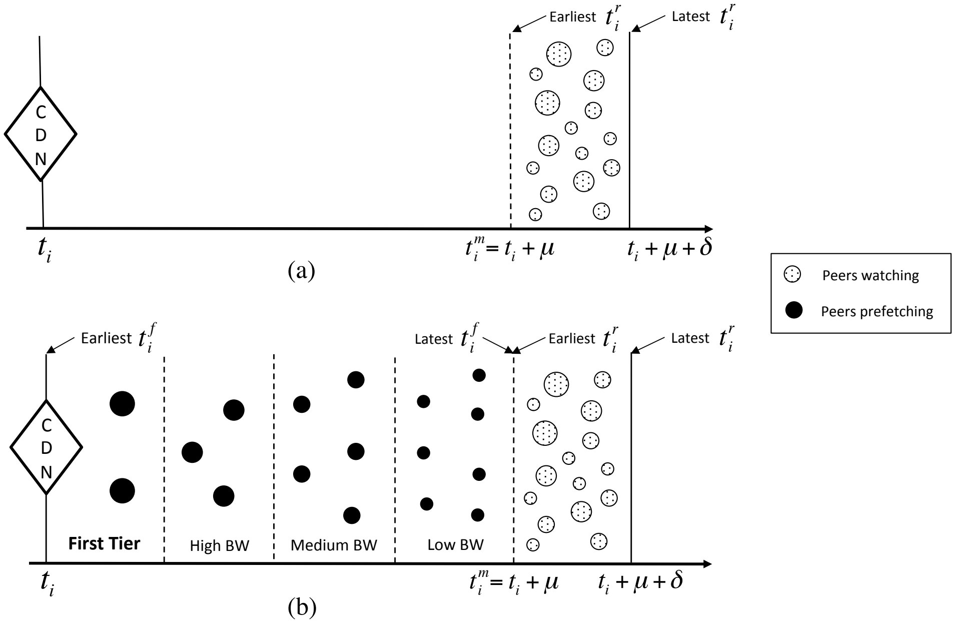


Figure 9.2: (a) All peers watching from the CDN and have access to the contents μ seconds after creation, (b) Peers of different bandwidth classes prefetching pieces in the order of their bandwidth and then viewing it at the same time as they would do from the CDN.

layer congestion control protocol [28] for transfers to and from other peers.

9.3.3 Server Infrastructure

The following is a list of services that we operate and which offer various helper functionalities to our PLS agent: *a*) *Admission:* upon entering the system, every peer is assigned a unique identifier which is a hash of it public key. Peers are admitted to the network and can communicate with the infrastructure servers and other peers after their identifiers are verified to be signed by this server, *b*) *Bandwidth Measurement:* each peer, upon startup, measures his bandwidth using this server. This measured bandwidth is used afterwards in the overlay construction algorithms, *c*) *NAT rendez-vous:* we use an augmented STUN [29] server to facilitate the traversal of network address translation (NAT) gateways. Additionally, upon startup, this server helps each agent detecting the NAT type of the gateway it is behind, according to the classification defined in [27], *d*) *Tracker:* The tracker is used to introduce peers to each others and disseminate information about them such as bandwidth capacity and NAT type. The tracker service also plays an essential role in identifying the best peers in the system for the overlay construction process.

9.4 Beyond Baseline Caching

We have, up until now, described a fully functional distributed caching system, where peers pull content from a random mesh overlay network upon player request. It is however necessary to utilize a more involved set of strategies to reach our goal of providing minimal load towards the source of the stream, while retaining quality of user experience of a hardware CDN. As a first step towards our goal, we elaborate on details regarding the functioning of production HTTP live streaming chains in order to understand how to provide the same kind of service. Then, we present a set of heuristics which leverage that knowledge to enhance the performance with respect to our baseline caching system.

9.4.1 The Window of Opportunity

In the HTTP live streaming chain, the lifetime of a given fragment i includes *three* main events: a) the time at which it is born, which is also its time stamp t_i, where the difference between any $t_{i+1} - t_i$ is a constant duration δ, b) the time at which it is published, i.e. the time t_i^m at which it gets included in the live streaming manifest resulting in peers inquiring about the latest available fragments being aware of the birth of fragment i, and c) the time t_i^r at which i is requested by the player of a certain client. The typical case for typical hardware CDNs offering HTTP live streams is to artificially introduce a delay μ between the birth of a fragment and its inclusion in the manifest i.e. $t_i^m - t_i = \mu$. This delay is to allow the propagation of the fragment to all CDN nodes. To be more specific, the born fragment is transferred to the CDN nodes as fast as possible and μ is configured to be long enough such that by the time i is included in the manifest, it is available on all CDN nodes. As shown in Figure 9.2(a) all peers are delayed μ seconds from the actual live playing point and all peers have their request time t_i^r in the period $[t_i^m, t_i^m + \delta]$, before the next piece is born.

Conversely, the time period between the generation of a fragment at the source t_i and the time at which the fragment has to be handed over to the player t_i^r constitutes the *window of opportunity* for distributing the fragment to all viewers in the system before the player requests it.

9.4.2 Exploiting the window of opportunity

We leverage this window of opportunity in two ways: first, by constructing an efficient overlay structure to enable quick dissemination of data between nodes in the system, and, second, by letting nodes prefetch data both from the source and from the other peers. The best peers in the overlay network are allowed to prefetch directly from the CDN in a rarest-first fashion, while all others prefetch from their partners in a greedy manner. That said, we have to stress that our solution does not introduce any additional delay, i.e., in the absence or presence of our distributed cache, a player will request the live content always μ seconds after its birth as in the case of a hardware CDN deployment.

Overlay construction

We strive to create a hierarchy of peers - yet without loosing the robustness of the random mesh - where peers with high upload capacity are closer to the source. We do that by creating a overlay comprising all peers in the system which internally self-sorts itself in such a way that each peer always picks uploaders with higher bandwidth than its own. This process is fully distributed and it is tailored to create an emergent behaviour which causes nodes to form a delivery mesh where peers with the best upload capacity are placed closer to the source of the stream and all others progressively further depending on their upload capacity. A number of additional factors are considered and the details on how our overlay construction heuristics are described in Section 9.4.3.

Prefetching

We implement prefetching heuristics which allow for a node to fill up its local cache with fragments that are about to be requested by the player, therefore limiting the probability of a node having to retrieve content directly from the source of the stream. Fragments are prefetched sequentially following their playback's order and only one fragment is prefetched at a time. However, sub-fragments of the same fragment may be downloaded in parallel from different sources. In our system, we implement two types of prefetching: proactive and reactive.

In *proactive prefetching*, we select a subset of peers in the system, which we call first tier peers, to behave exactly as CDN nodes would. That is, they aggressively prefetch and cache fragments as soon as they are born in order to make them available to the rest of the overlay. Proactive prefetching makes sure that the window of opportunity in our system is the same as that of a CDN, a fragment is prefetched at a point t_i^f where $t_i \leq t_i^f \leq t_i + \mu]$. The first tier peer selection process is tracker-assisted and it is tasked to identify a few peers among the ones with best upload capacity such that the their aggregate upload capacity is enough to supply the rest of the peer-to-peer.

We implement *reactive prefetching* which allows prefetching of fragments between peers as soon as they become available from their neighborhood. The download of a fragment at a peer's end is triggered by the reception of an advertisement for that same fragment from one or more neighbors. Reactive prefetching is enabled on all nodes in the system such as to maximize the whole peer-to-peer network utilization. Therefore t_i^f for a given t_i will differ depending on how close a peer is to the CDN.

In Figure 9.2(b), we visualize the timeline of a fragment's delivery in our system using our prefetching heuristics. First tier peers use proactive prefetching to request the fragment as soon as it is born. After that, the fragment is reactive prefetched from the first tier peers by the rest of the overlay. For simplicity, we classify the overlay by a peer's upload capacity and in three bandwidth classes: high, medium and low. As we can see, the fragment is first downloaded by peers in the highest bandwidth class and then by the ones in medium and in low. This is an effect of our bandwidth-ordered overlay structure.

Multi-bitrate support

Multi-bitrate support is not achieved by constructing a separate overlay for each bitrate. Instead, first tier peers prefetch more than one bitrate. A small price which we decided to pay instead of creating churn by frequent hopping between different overlays.

Regarding overlay construction, peers preferably choose neighbors that are downloading (prefetching or watching) the same bitrate as they are watching. On top of that, each peer will keep a small selection of other peers that are downloading bitrates immediately adjacent to the one it is watching. This as to quickly adapt to bitrate switches without having to fall back to the source of the stream. Note that the invariant of each peer choosing neighbors with higher upload bandwidth is kept even when watching a multi-bitrate stream.

9.4.3 Overlay Construction

The overlay construction process is divided into three sub-processes which are: *first tier construction*, *out-partners selection* and *in-partners selection*. In order to provide overlay samples as input to the latter, we also implement a distributed peer sampling service.

All peers run the same neighborhood selection process, where each peer p chooses a number of candidates out of a local sample $S \subset \mathscr{P}$ of all peers $\mathscr{P}$ in the overlay to become part of its neighborhood. A peer's neighborhood is composed of two sets, the in-partner set I_p and the out-partner set O_p. I_p contains peers that p has chosen to act as uploaders for fragments. The out-partners of a peer p are the partners that have chosen p as in-partner and p accepted them as such. Peer p accepts a maximum number of out-partners O_p^{max} depending on its maximum bandwidth capacity u_p that is estimated against our bandwidth measurement service.

9.4.4 Peer Sampling

By periodically contacting the tracker service, each peer keeps its random sample S of the overlay up-to-date. Samples are also exchanged among peers. We use our NAT-resilient peer sampling service called WPSS [30]. The service returns a uniform random sample of the nodes in the network, which is added to S. Each peer entry in it contains: a peer identifier, the peer's NAT-type and the maximum upload capacity. This information is periodically refreshed by the sample service and by the tracker service and used as input in the neighbor selection process.

9.4.5 In-partners Selection

Let I_p^{max} be the desirable size of I_p, the set of in-partners for a peer p. Every peer periodically updates its I_p set after a period of T_{repair} seconds by removing a subset of under-performing peers I_p^r. A new set of peers I_p^a is then selected from its local sample of the overlay S and added to I_p. This update happens only if p has not been getting all the content from the overlay network in the last period T_s.

The in-partner selection process makes sure that $\left|I_p\right| = \left|I_p \setminus I_p^r\right| + \left|I_p^a\right| = I^{max}$. I_p^r has size $\left|I_p^r\right| = \min(\beta * I^{max}, I^{max} - \left|I_p\right|)$ and β is a replacement factor typically set to $\beta = 0.3$.

Each peer strives to obtain a number of in-partners which is equal to I^{max} as a safety measure in order to have readily available an alternative sender if connectivity issues, congestion or churn temporarily prevent him from retrieving content from one of its partners.

Remove Filters

We here define a set of removal filters, which are applied to the current I_p, and that output the set of peers I_p^r to remove from I_p.

Prefetching point. This filter identifies and removes all the peers in I_p that are prefetching fragments later than p. We do so by estimating the prefetch indicator $pi(i) = t_i^f - t_i^r$ for each fragment i, which is the difference between the time i was prefetched and the time i was requested by the player. Note that we use t_i^r as a point of reference because we assume that the time between two fragment requests is constant and corresponds to δ. On top of that, we assume that all peers request the same fragment i in a period of time $[t_i^m, t_i^m + \delta]$, as explained in Section 9.4.1.

Peers estimate their average request indicator $\overline{pi}$ over the last downloaded fragments and then piggy-back it to all fragment advertisements. In the filter, we compare p's average request point with all of its current partners' and remove partners which have a smaller $\overline{pp}$ than peer p. The filter is defined as follows:

$$I'_p = \{\, q \mid \overline{pi}_q < \overline{pi}_p,\ q \in I_p \,\},$$

where $\overline{pi}_q$ is the average prefetch point of peer q and $\overline{pi}_p$ the one of p.

Successful transfers. The filter has the goal of removing a fixed number of underperforming peers in to order to promote discovery of new ones. We define the ordered tuple $Q = (q_0, q_1, ..., q_n)$ containing all peers in I'_p. The tuple Q is ordered in ascending order according to the number of successful fragment downloads from each peer in the set in the last period of time T_{repair}. We then select and filter the ones with the fewer number of successful downloads until we remove a minimum of peers from the set, that is $|I_p^r| = \beta * I_p^{max}$.

$$I_p^r = \{\, (q_0, q_1, ..., q_k) \mid k \le \max\{\, 0, |I'_p| - I_p^{max}(1-\beta) \,\}\}$$

Add Filters

Add filters have the purpose of selecting which peers will be added to I_p. Add filters are applied to S and produce a set of best candidates peers I_p^a.

Connectivity. We filter out any peer q which cannot establish bidirectional connectivity with p.

$$S' = \{\, q \mid imp(\tau(p), \tau(q)),\ q \in S \,\}$$

$imp(\tau(p),\tau(q))$ is a predicate that evaluates to true if bi-directional communication between p and q is not possible according to [27], where $\tau(q)$ is the type of NAT/firewall/gateway that q is behind.

Bandwidth. This filter removes all peers from S' which have lower bandwidth capacity than p. The rationale behind this is to have peers in a certain bandwidth range become partners with peers of similar or higher bandwidth. This filter together with the out-partner selection process described in the next section makes possible to construct an hierarchical overlay structure based on bandwidth capacity of peers.

$$S'' = \{\, q \mid u_p < u_q,\ q \in S' \}$$

Bitrate. The filter picks a finite subset R of size N from S' according to the bitrate peers in S' are downloading. The rationale behind the filter is to preferably choose peers from S' which are watching the same bitrate b peer p is watching and, additionally, a number of other peers which are watching the bitrates immediately higher and lower bitrate b.

We hereby describe how set R is constructed. Let S_k be the set of peers which are downloading bitrate k from S'', and let S_k^n be a set of size n_k from S_k without replacement. We define α as the percentage of R to be chosen from S_b (the bitrate p is watching). α_{b-1} and α_{b+1} are instead the percentages of the elements to be chosen from S_{b+1} and S_{b-1}, where $b+1$ and $b-1$ are the bitrates immediately lower and higher than b, respectively. The values of α_{b+1} and α_{b-1} are calculated as:

$$\alpha_{b+1} = \begin{cases} \frac{1-\alpha}{2}, & 1 < b < \mathrm{B} \\ 1-\alpha, & b = 1 \end{cases}$$

$$\alpha_{b-1} = \begin{cases} \frac{1-\alpha}{2}, & 1 < b < \mathrm{B} \\ 1-\alpha, & b = \mathrm{B} \end{cases}$$

Here B is meant as the largest bitrate of all bitrates watched by peers in S''. We then estimate n_k, the number of elements to pick from each set S_k in the following way: $n_k = \min(N * \alpha_k, |S_k|)$ for $k = b, b-1, b+1$. At this point, we can build the output of the filter I_p^a as:

$$I_p^a = \{p \in S_b^{n_b} \cup S_b^{n_{b-1}} \cup S_b^{n_{b+1}}\}$$

If $|I_p^a| = N$ then the set I_p^a is returned as is. Otherwise, a sample of $w = N - n_b - n_{b-1} - n_{b+1}$ values is randomly selected from the set $A = \bigcup_{j=1..n_b} S_j \setminus R$ and added to I_p^a.

9.4.6 Out-partners selection

The out-partner selection process is responsible for choosing, upon a partnership request from a downloader, if the request should be accepted or not. If that is the case, an existing partner may be evicted from the set of out partners. We hereby describe how this process works.

We define O_p^{max} as the maximum number of out-partners for peer p. This is calculated as $O_p^{max} = \min(u_p / b_p)$. As long as $|O_p| < O_p^{max}$, new outgoing partners that chose p as in-partner are accepted. Every newcomer is given a grace period T_{grace} where it is not considered for eviction, when T_{grace} is over and the peer has not requested any fragment from p, the peer is added to a set O_p^r. Consequently, O_p^r contains at all time the partners which may be considered for eviction. If new applicant q, i.e. a downloader requesting to become partner of p, has a maximum upload bandwidth u_q that is larger than any of the peers in O_p^r, the peer with the smallest maximum upload capacity in O_p^r is evicted and replaced by q.

The bandwidth-based out-partner selection policy makes sure that the best peers in the system, e.g. the first tier peers, accept as out-partners only the best peers in the second tier. The latter will do the same and the overlay will self-organize in an hierarchical structure where peer bandwidth's utilization is maximized at every level of the second tier. As such, we increase the efficiency of delivery in the overlay and minimize the probability of falling back to the source.

9.5 Prefetching

9.5.1 Proactive prefectching

We have designed a process for deciding on a set of peers which proactively prefetch content from the source and make it available to the rest of overlay. The process is tracker-assisted. Every peer p reports its number of out-partner slots O_p and the bitrate it is watching $b(p)$ to the tracker. Periodically, the tracker executes the first tier construction heuristic and communicates to a subset of peers to activate proactive prefetching on a specific bitrate.

The first tier construction process relies on the assumption that the overlay is organized in a bandwidth-ordered fashion, where peers with the highest upload capacity are positioned closer to the source of the stream. First tier peers are selected among those best peers as they offer the best download to upload ratio and therefore can serve a large amount of peers with relatively small load on the source.

The goal of first tier construction is to choose the least possible number of peers while maintaining two constraints: i) all peers are provided for, either from the first tier peers or from the rest of the overlay, ii) all peers retrieve fragments from the overlay before the player actually requests them, that is a period of time μ after the fragment is generated at t_i as defined in Section 9.4.1.

Intuitively, the more peers prefetch from source of the stream, the quicker the diffusion on the overlay will be and thus the lower the probability that a peer will have to request from the source. Alternatively, a small number of peers prefetching from the source would provide higher savings towards the source but also increase the probability of peers failing to retrieve fragments in their entirety from the overlay.

In order to estimate correctly the number of peers prefetching from the source, we designed a simple heuristic which models the overlay as a multi-row structure where first tier peers are positioned in the first row and all others in lower levels depending

Algorithm 6 Row construction heuristic

```
Input: $P = (p_0, p_1, ..., p_n), L$
Output: $F$
begin
  $F' \leftarrow P$
  $i \leftarrow 0$
  while $F' \neq \emptyset$ do
      $i = i + 1$
      $r = 0$
      $F = \{ p_n : n \leq i \}$
      $F' = P \setminus F$
      $upRowCap = \sum_{k=1}^{F} |O_k|$
      $curRowCap = 0$
      while $F' \neq \emptyset$ and $r \leq L$ do
          $q = q_0 \mid F' = \{q_0, q_1, ...\}$
          $F' = F' \setminus q$
          if $upRowCap - I^{max} > 0$ then
              $upRowCap = upRowCap - I^{max}$
              $curRowCap = curRowCap + O_q$
          else
              $r = r + 1$
              $upRowCap = curRowCap$
              $curRowCap = 0$
          end if
      end while
  end while
end
```

on their upload capacity, in descending order. The heuristic is a variation of the tree-construction algorithm described in our previous work [31].

We have to stress at this point that the following heuristic is only used as an approximation method to compute the number of first-tier piers while the actual construction of the overlay is achieved in a completely decentralized fashion in a mesh topology as described in Section 9.4.3.

First tier construction heuristic. The heuristic takes into account the constraints defined earlier by making sure that: i) each row r has enough upload capacity to serve the next row $r+1$, ii) the number of rows does not exceed a maximum of $L = \frac{\mu}{\delta}$, i.e. given that the unit of transfer is a fragment of duration δ, then a fragment should travel from the first to the last row in a period of length of at most μ, the CDN deadline.

The first tier construction heuristic is shown in Algorithm 6. The input is the desired maximum number of rows L and the set of peers P ordered by the number of out-partner slots, in descending order. The output is the set of first tier peers F.

The process starts by selecting a subset $F \subset P$, namely the first i peers, where $upRowCap$ is the sum of the number of their upload slots. Then, it iterates over all the non first tier peers $F' = P \setminus F$ and tries to place each peer q in the current row r.

If the capacity available $upRowCap$ is enough to accommodate q by sustaining its number of in-partners I^{max}, then the remaining $upRowCap$ is decreased and the number of p's out-partner slots added to the $curRowCap$. Otherwise, a new row is created and the current row capacity $curRowCap$ becomes the capacity available for the next row $upRowCap$. The row construction algorithm continues to allocate the peers in F' until either all peers from F' have been allocated or the number of rows has exceeded the maximum L. Reaching the first case means that F is sufficiently large. In the latter case, since not all peers have been allocated, the heuristic increases the set of first tier peers in F and retries until all peers in F' have been placed.

Besides what we described above, we also augment the heuristic to accommodate for two more factors. First, in order to become an eligible first tier candidate, a peer needs to be behind a NAT that is easily traversable or is an open Internet node. Second that peers in F can prefetch multiple bitrates which affects the calculation of $upRowCap$ to account for uploading multiple bitrates.

9.5.2 Reactive prefectching

On all peers, the peer-to-peer prefetcher runs in parallel with the player and tries to populate the fragment cache with upcoming fragments of the same bitrate the player is requesting, before they are actually requested by the latter. The peer-to-peer prefetcher retrieves fragments exclusively from the overlay and does so upon receipt of advertisements of downloaded fragments from in-partners. The idea is to increase the utilization of the P2P network and decrease the risk of failing to fetch a fragment in time when requested by the player.

Traffic Prioritization. To implement this proactive strategy we have taken advantage of our dynamic runtime-prioritization transport library DTL [28] which exposes to the application layer the ability to prioritize individual transfers relative to each other and to change the priority of each individual transfer at run-time. Upon starting to fetch a fragment proactively, it is assigned a lower-than-best-effort-priority, by using the LEDBAT congestion control implemented in DTL. The rationale is to avoid contending with the transfer process of fragments that are reactively requested and under a deadline both on the uploading and downloading ends. On top of that, as almost all the traffic in our system is LEDBAT traffic, we also avoid interfering with traffic generated by other applications. Thus meeting the requirement of politeness towards the user and host.

9.6 Evaluation

Our customer network has a total of of around 400000 peers located all over the world. In order to evaluate our application, we installed an instrumented version of our software on a subset of our customers who agreed to let us conduct experiments on their machines. The experiments are executed remotely using different sets of parameters

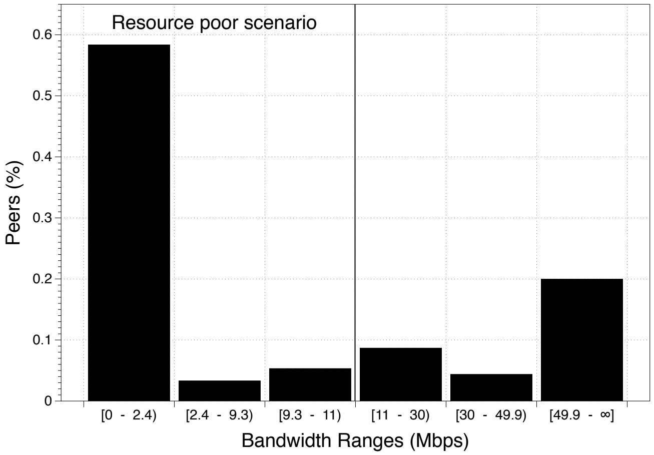


Figure 9.3: Test network bandwidth distribution

according to the needed scenario. Needless to say, these functionalities are removed from the commercial version of our software.

Th subset of instrumented nodes amounted to around 22000 machines. Nevertheless, for each data point in our results, we selected a sample of 2000 randomly-chosen peers in order to ensure that the results are not valid only for a particular set of machines. The majority of peers in the network are located in Sweden (85%), while the others are in the rest of Europe (8%) and the US (5%). In Figure 9.3, we show the distribution of upload bandwidth in our network. The measurements are made against a central server. The maximum bandwidth reported is capped to $50Mbps$. As we can observe, the majority of the peers is either on the lower end of the spectrum $(0-2.4\text{Mbps})$ or the higher ($> 49Mbps$,) while the others are almost evenly distributed across the rest of the spectrum.

Regarding connectivity, 79% of the peers were behind NAT, and 18% were on open Internet and the rest could not be determined or did not allow UDP traffic. In the experiments, we have used our state of the art traversal scheme [27] and were able to establish bi-directional communication between 84% of all peer pairs.

9.6.1 Test-bed

Stream Properties. We test our application in both single bitrate and multi-bitrate mode. We used a production-quality continuous live stream configured with different bitrates according to the scenario we wanted to present. The stream was published using Microsoft Smooth Streaming traditional tool chain. The bandwidth of the source stream was provided by a commercial streaming server and a commercial CDN for the fallback. We made sure that the CDN had enough capacity to serve the maximum number of peers for all tests we run. This setup gave us the ability to compare the quality of

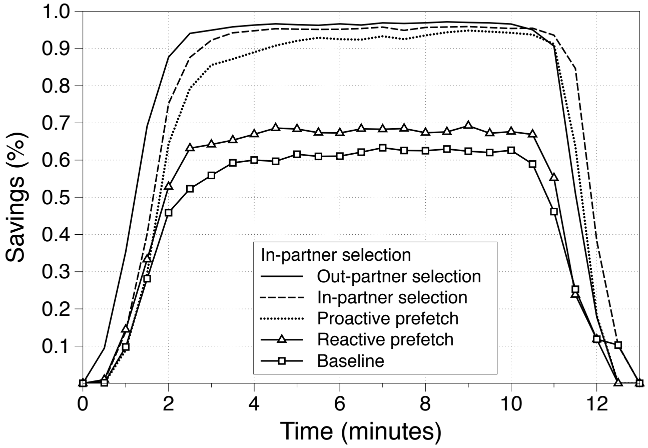


Figure 9.4: Single bitrate, resource-rich scenario

the streaming process in the presence and absence of P2P caching in order to have a fair assessment of the effect of our application on the overall quality of user experience.

In order to create the most realistic conditions for our tests, we let the peers execute the standard Silverlight player, although in headless mode, which does not render the video and does not require user intervention.

Aside the services described in Section 9.3.3, the rest of our test infrastructure comprises of a logging server and a controller to launch tests remotely.

Reproducibility. Each test to collect one data point in the test network happens in real time and exploring all parameter combination of interest is not feasible. Therefore, we did a major parameter combinations study on our simulation platform [32] first to get a set of worth-trying experiments that we launched on the test network. Another problem is the fluctuation of network conditions and number of peers. We repeated each data point a number of times before gaining confidence that this is the average performance of a certain parameter combination.

Scenarios For single bitrate tests, we define two scenarios, a *resource-rich(RR)* scenario and a *resource-poor(RP)* scenario. The resource rich scenario represents the configuration of our standard deployments, while we construct the resource-poor scenario in order to highlight some of the strengths of our heuristics. In order to estimate the resources available in each scenario we introduce the notion of resource factor, that is defined as:

$$RF = \frac{\sum_{i=1}^{N} min(u_i, O^{max} * b_i)}{N} \tag{9.1}$$

N is the number of peers in the network and u_i is the measured upload capacity of peer i. O^{max} is a global system parameter that establishes a maximum on the amount of out-partners allowed on any peer, that is for limiting the number of upload bandwidth

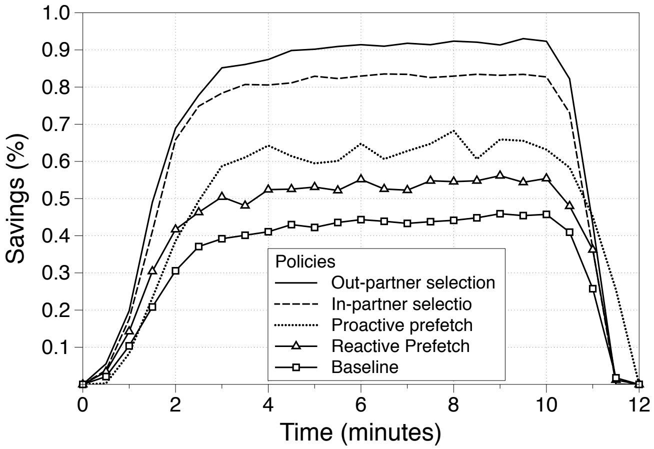


Figure 9.5: Single bitrate, resource-poor scenario

utilized on the best peers in the network and therefore promote politeness towards our customers' machines. Finally, b_i is the stream's bitrate. The resource factor represents the average number of streams that a peer is able to upload to neighbors for all peers in the network.

In the resource-rich scenario, we use a random sample of 1000 peers from the entirety of our test network, we let the peers watch a $688Kbps$ bitrate stream and we limit their O^{max} to 10, with a resulting resource factor of $RF =$ 5.

In the resource-poor scenario instead, we pick a random sample of again 1000 peers from the lower end of the bandwidth distribution, that is from 0 up to $11Mbps$. Then, we let the peers watch a stream of 1200Kbps, while the maximum number of out partners is $O^{max} = 5$. The resulting resource factor for this scenario is around 2.5. It is important to highlight that the resource factor estimation does not take into account connectivity limitations between peers, i.e. NATs/firewalls, and therefore it is safe to assume that the effective resource factor for both scenarios is actually lower.

For multi-bitrate tests, we use the same configuration of the *resource-rich* scenario but with 3 video bitrates, that is 331, 1000, 2000 Kbps. Note that O^{max} is kept to a very low number on all peers, $O^{max} = 10$ for multi-bitrate and single bitrate RR and $O_i^{max} = 5$ for single bitrate RP. This is to promote fairness and spread the load of the distribution across all peers and politeness towards the host.

9.6.2 Experiments

All experiments have a duration of 10 minutes. Peers are joined uniformly in a period of time of one minute and churn depends on the test at end.

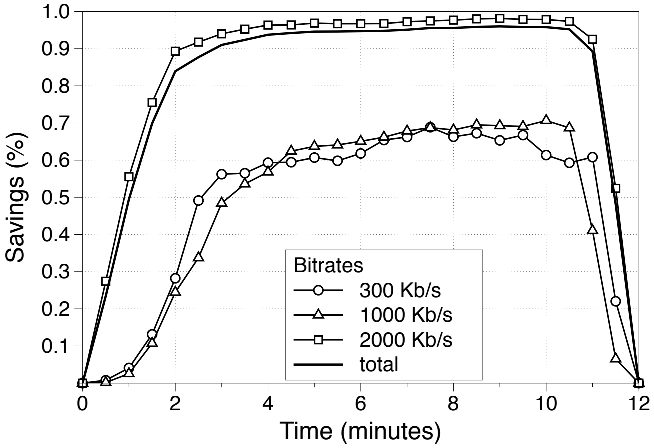


Figure 9.6: Multi-bitrate

Savings Using a single bitrate stream, we analyse the impact on savings of each of the main processes of our system: peer-to-peer pre-fetching, first tier construction and pre-fetching, in-partner selection and out-partner selection.

As reference for our experiments, we use baseline caching as described in Section 9.3.1, with all the aforementioned processes disabled. We then enable them one by one starting from peer-to-peer pre-fetching. In Figure 9.4, we show the evolution of savings in time in the single bitrate resource-rich scenario. As we can observe, peer-to-peer prefetch yields to a fairly small improvement in savings from the baseline (from 60% to 66% at steady state) because peers have short time to exchange data, as fragments become available from an uploader only when they are downloaded on response to a player request. Activating first tier construction and pre-fetching leads instead to a significant increment in performance (94%) because of the additional time μ allowed for peer-to-peer prefetching to provide to the rest of the overlay. However, enabling the overlay construction policies provides only a small increase in savings (96% at steady state). This is because of the abundance of resources in the scenario, where few peers can provide for most of the overlay network and therefore there is no need for a hierarchical structure in the overlay.

In Figure 9.5, we show the effect of the same processes but in the resource-poor scenario, always using a single bitrate. In this case, the overlay construction heuristics contribute to most of the savings, 82% at steady state, while the out-partner selection brings the savings to 92% on average. It is evident that, in the resource-poor scenario, constructing an efficient overlay where peers with higher upload capacity are arranged closer to the source is of vital importance to achieve high levels of savings.

We now consider savings in the multi-bitrate scenario, Figure 9.6 presents the evolution of savings in the system for all bitrates combined (95% at steady state) and for each bitrate separately. As we can see, the savings for low bitrates are lower (70%) than

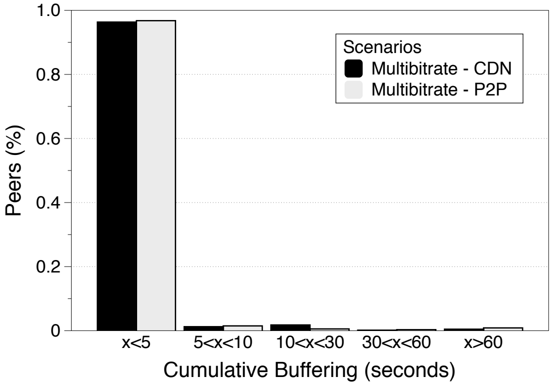


Figure 9.7: Cumulative buffering

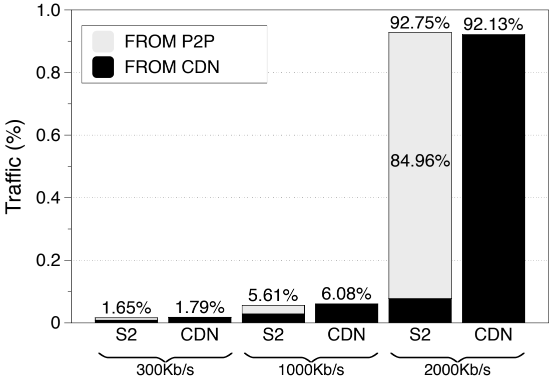


Figure 9.8: Cumulative traffic split

the ones for the highest one (98%). This is due to peers continuously switching from the lowest to the middle bitrate and vice versa, therefore not being able to stabilize their in-partner set for retrieving all the content from the overlay. The switching happens throughout the test and we attributed it to the bandwidth oscillations in the test-bed that are out of our control.

In Section 9.5.2, we described how transfers are assigned lower-than-best-effort priority while proactively pre-fetching from other peers. We have estimated the low priority traffic to be on average 97% of the cumulative peer-to-peer traffic during the aforementioned multi-bitrate tests. For the previous single bitrate experiments instead, we have achieved 98% and 95% for the resource-rich and resource-poor scenario, respectively. Consequently, in all our testing scenarios we have met the requirement of politeness towards the host and other network traffic.

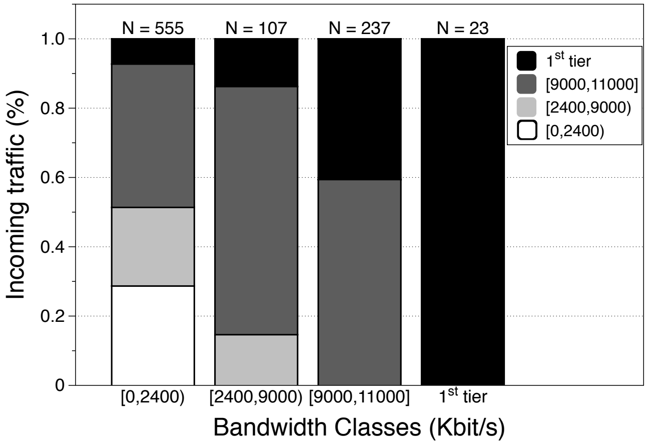


Figure 9.9: Classification of cumulative peer-to-peer transferred data per bandwidth class

Quality of user experience Getting savings alone is not a good result unless we have provided a good user experience. To evaluate the user experience, we use two metrics: First, the percentage of peers who experienced a total buffering time of less than 5 seconds, i.e. they enjoyed performance that did not really deviate much from live. Second, showing that our P2P agent did not achieve this level of savings by forcing the adaptive streaming to move everyone to the lowest bitrate. For the first metric, Figure 9.7 shows that we can make 95% of the network watch the stream with less than 5 seconds of buffering delay. For the second metric, Figure 9.8 shows also that, as indicated by bars with the SmoothCache2.0(S2) label, 92.75% of all consumed traffic was on the highest bitrate and P2P alone shouldering 84.96%, an indication that, for the most part, peers have seen the video at the highest bitrate with a major contribution from the P2P network.

P2P-less as a Reference. We take one more step beyond showing that the system offers substantial savings with reasonable user experience, namely to understand what would be the user experience in case all the peers streamed directly from the CDN. Therefore, we run the system with P2P caching disabled. Figure 9.7 shows that without P2P, only 1% less (94%) of all viewers would have a less than 5 seconds buffering. On top of that, Figure 9.8 shows that 0.62% less (92.13%) of all consumed traffic is on the highest bitrate. As shown, our solution provides the same or slightly better user experience than the one of the CDN.

Overlay organization In order to show that our heuristics indeed promote a hierarchical structure of the overlay based on bandwidth, we categorize, for one run, the cumulative peer-to-peer traffic for each bandwidth class including first tier peers in Figure 9.9. Each stacked bar shows the percentage of peer-to-peer traffic coming from different peers classes. We consider here a resource-poor scenario.

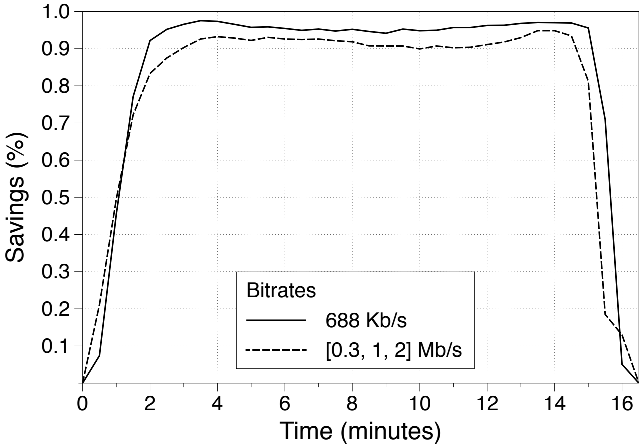


Figure 9.10: Scenario 1% churn

As we can see, all p2p traffic in the first tier comes from other first tier peers. That is not true however for the the first bandwidth class (from 9000 to 11000Kbps) on the rest of the overlay, with just 40% of the traffic coming from the first tier. We can also observe that bandwidth class [9000, 11000) provides for most of the second and third bandwidth classes in the overlay. Instead, the last class [0, 2400), that is the largest by number of peers $N = 555$, takes only half of data from peers in the first and second bandwidth class, while striving to retrieve data from the same class and the class above. For all classes, the invariant of not retrieving data from a lower bandwidth class is maintained.

Effect of churn We now explore the performance of our system under churn. Figure 9.10 shows the evolution of savings in a scenario where, after two minutes from the start of the experiment, we start failing and joining 1% of the nodes every 10 seconds until the 9^{th} minute. This is consistent with the churn experienced in our commercial deployments. In both multi-bitrate scenario (labeled as $[0.3, 1, 2]Mb/s$) and single-bitrate ($688Kb/s$, on RR configuration), the churn has only small negative impact on the savings, which is quantified to a maximum of 3% during the time the churn process is active on both scenarios.

In Figure 9.11, we create a churn scenario where 50% of the peers leaves the overlay network in one minute at time 6 minutes, which models a massive failure in the network. This churn pattern leads only to a drop in savings in the multi-bitrate scenario, from 95% to 88%, from which the system recovers after one minute the churn event stops. In the single-bitrate scenario, the drop is much smaller, from 98% to 95%.

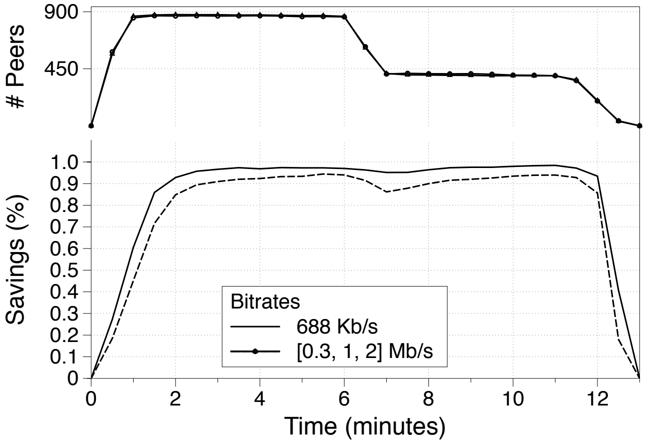


Figure 9.11: Massive failure scenario

9.7 Conclusion

In this work, we have presented the SmoothCache 2.0 system. The main motivation behind it is to cope with the recent shift in commercial live streaming protocols to use HTTP as the transport protocol instead to RTSP/RTP where timeliness is kept not by skipping video fragments but by changing bitrates. Our main target scenario is to decrease the load on a CDN broadcasting a live stream while offering the same quality and not the more typical P2P live streaming setting where a consumer machine with limited bandwidth is the source of the stream.

The approach of this work could be summarized as follows. Given the existence of an internal delay inside CDNs while releasing live video streaming fragments, we promote a small subset of powerful peers to act similarly to CDN nodes, by aggressively prefetching fragments as they are created. We make sure, then, that all prefetched data is propagated in the network within a small window of time. The goal is to make video fragments available to all peers after a delay that is not higher than they would experience by accessing the CDN directly. The fast propagation during the small window of time is made possible by employing two main techniques: *i*) a mesh-based overlay structure which enforces a hierarchy based on upload bandwidth but additionally takes into consideration other factors like connectivity constraints, performance history, prefetching point and currently watched bitrate *ii*) a mix of proactive and reactive prefetching strategies with various levels of aggressiveness using an application-layer congestion control balancing high dissemination efficiency and politeness.

Our evaluation on real consumer machines, shows that SmoothCache 2.0 delivers source bandwidth savings of up to 96% in a high bandwidth resource scenario, while it manages to save 92% in a constrained bandwidth scenario. Besides achieving high savings, we showed that our system meets the same quality of user experience as a CDN.

When all the peers watched the stream directly from the CDN 94% of the viewers experienced a maximum delay of 5 seconds, and 92% watched the highest bitrate. Both metrics were matched using P2P distribution albeit with substantial bandwidth source savings.

9.8 References

[1] X. Hei, C. Liang, J. Liang, Y. Liu, and K. W. Ross. Insights into PPLive: A Measurement Study of a Large-Scale P2P IPTV System. In *Proc. of IPTV Workshop, International World Wide Web Conference*, 2006.

[2] Xinyan Zhang, Jiangchuan Liu, Bo Li, and Y. S. P. Yum. CoolStreaming/DONet: a data-driven overlay network for peer-to-peer live media streaming. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*

[3] Feng Wang, Yongqiang Xiong, and Jiangchuan Liu. mtreebone: A hybrid tree/mesh overlay for application-layer live video multicast. In *Distributed Computing Systems, 2007. ICDCS '07. 27th International Conference on*, pages 49–49, 2007.

[4] Bittorrent live. http://live.bittorrent.com/.

[5] Amir H. Payberah, Jim Dowling, Fatemeh Rahimian, and Seif Haridi. gradientv: Market-based p2p live media streaming on the gradient overlay. In *Proc. of the 10th IFIP international conference on Distributed Applications and Interoperable Systems (DAIS'10)*, Lecture Notes in Computer Science, pages 212–225. Springer, June 2010.

[6] S. Traverso, L. Abeni, R. Birke, C. Kiraly, E. Leonardi, R. Lo Cigno, and M. Mellia. Experimental comparison of neighborhood filtering strategies in unstructured p2p-tv systems. In *Peer-to-Peer Computing (P2P), 2012 IEEE 12th International Conference on*, pages 13–24, 2012.

[7] N. Magharei, R. Rejaie, and Yang Guo. Mesh or multiple-tree: A comparative study of live p2p streaming approaches. In *INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE*, pages 1424–1432, 2007.

[8] Saamer Akhshabi, Ali C. Begen, and Constantine Dovrolis. An experimental evaluation of rate-adaptation algorithms in adaptive streaming over HTTP. In *Proc. of ACM MMSys*, 2011.

[9] Roberto Roverso, Sameh El-Ansary, and Seif Haridi. Smoothcache: Http-live streaming goes peer-to-peer. In *NETWORKING 2012*, volume 7290 of *Lecture Notes in Computer Science*, pages 29–43. Springer Berlin / Heidelberg, 2012. ISBN 978-3-642-30053-0.

[10] Bo Li, Susu Xie, Yang Qu, Gabriel Yik Keung, Chuang Lin, Jiangchuan Liu, and Xinyan Zhang. Inside the new coolstreaming: Principles, measurements and performance implications. In *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*, pages 1031–1039. IEEE, 2008.

[11] Hao Yin, Xuening Liu, Tongyu Zhan, Vyas Sekar, Feng Qiu, Chuang Lin, Hui Zhang, and Bo Li. Livesky: Enhancing cdn with p2p. *ACM Trans. Multimedia Comput. Commun. Appl.*, 6:16:1–16:19, August 2010. ISSN 1551-6857. URL `http://doi.acm.org/10.1145/1823746.1823750`.

[12] Feng Wang, Jiangchuan Liu, and Minghua Chen. Calms: Cloud-assisted live media streaming for globalized demands with time/region diversities. In *INFOCOM, 2012 Proceedings IEEE*, pages 199–207, 2012.

[13] Mingchen Zhao, Paarijaat Aditya, Ang Chen, Yin Lin, Andreas Haeberlen, Peter Druschel, Bruce Maggs, Bill Wishon, and Miroslav Ponec. Peer-assisted content distribution in akamai netsession. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 31–42. ACM, 2013.

[14] Adobe. Rtmfp for developing real-time collaboration applications, 2013. URL `http://labs.adobe.com/technologies/cirrus/`.

[15] Adobe Labs. Voddler, 2013. URL `https://www.adobe.com/content/dam/Adobe/en/casestudies/air/voddler/pdfs/voddler-casestudy.pdf`.

[16] Cheng Huang, Jin Li, and Keith W Ross. Peer-assisted vod: Making internet video distribution cheap. In *IPTPS*, 2007.

[17] Cheng Huang, Jin Li, and Keith W. Ross. Can internet video-on-demand be profitable? *SIGCOMM Comput. Commun. Rev.*, 37(4):133–144, August 2007. ISSN 0146-4833. URL `http://doi.acm.org/10.1145/1282427.1282396`.

[18] Liang Zhang, Fangfei Zhou, Alan Mislove, and Ravi Sundaram. Maygh: Building a cdn from client web browsers. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 281–294. ACM, 2013.

[19] Zhengye Liu, Yanming Shen, Shivendra S Panwar, Keith W Ross, and Yao Wang. Using layered video to provide incentives in p2p live streaming. In *Proceedings of the 2007 workshop on Peer-to-peer streaming and IP-TV*, pages 311–316. ACM, 2007.

[20] Venkata N Padmanabhan, Helen J Wang, and Philip A Chou. Resilient peer-to-peer streaming. In *Network Protocols, 2003. Proceedings. 11th IEEE International Conference on*, pages 16–27. IEEE, 2003.

[21] Luca Abeni, Arpad Bakay, Robert Birke, Gianluca Ciccarelli, Emilio Leonardi, Renato Lo Cigno, Csaba Kiraly, Marco Mellia, Saverio Niccolini, Jan Seedorf, Tivadar

Szemethy, and Giuseppe Tropea. Winestreamer(s): Flexible p2p-tv streaming applications. In *IEEE INFOCOM 2011 - IEEE Conference on Computer Communications Workshops*, Shanghai, China, 2011. URL `http://peerstreamer.org/~cskiraly/preprints/kiraly-INFOCOM2011-demo-preprint.pdf`.

[22] Http live streaming protocol. http://developer.apple.com/streaming/.

[23] Adobe http dynamic streaming protocol. http://www.adobe.com/products/hds-dynamic-streaming.html.

[24] Microsoft Inc. Smooth Streaming. http://www.iis.net/download/SmoothStreaming.

[25] Shao Liu, Rui Zhang-Shen, Wenjie Jiang, Jennifer Rexford, and Mung Chiang. Performance bounds for peer-assisted live streaming. In *Proceedings of the 2008 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS 2008, pages 313–324. ACM, 2008. ISBN 978-1-60558-005-0.

[26] A. Mansy and M. Ammar. Analysis of adaptive streaming for hybrid cdn/p2p live video systems. In *Network Protocols (ICNP), 2011 19th IEEE International Conference on*, pages 276–285, 2011.

[27] Roberto Roverso, Sameh El-Ansary, and Seif Haridi. NATCracker: NAT Combinations Matter. In *Proc. of 18th International Conference on Computer Communications and Networks*, ICCCN '09, SF, CA, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-4581-3.

[28] Riccardo Reale, Roberto Roverso, Sameh El-Ansary, and Seif Haridi. DTL: Dynamic Transport Library for Peer-To-Peer Applications. In *In Proc. of the 12th International Conference on Distributed Computing and Networking*, ICDCN, Jan 2012.

[29] J. Rosenberg, J. Weinberger, C. Huitema, and R. Mahy. STUN - simple traversal of user datagram protocol (UDP) through network address translators (NATs). RFC 3489, March 2003.

[30] R. Roverso, J. Dowling, and M. Jelasity. Through the wormhole: Low cost, fresh peer sampling for the internet. In *Peer-to-Peer Computing (P2P), 2013 IEEE 13th International Conference on*, 2013.

[31] R. Roverso, A. Naiem, M. Reda, M. El-Beltagy, S. El-Ansary, N. Franzen, and S. Haridi. On the feasibility of centrally-coordinated peer-to-peer live streaming. In *Consumer Communications and Networking Conference (CCNC), 2011 IEEE*, pages 1061–1065, 2011.

[32] Roberto Roverso, Sameh El-Ansary, Alexandros Gkogkas, and Seif Haridi. Mesmerizer: A effective tool for a complete peer-to-peer software development life-cycle. In *In Proceedings of SIMUTOOLS*, March 2011.

CHAPTER 10

SmoothCache Enterprise: Adaptive HTTP Live Streaming in Large Private Networks

SmoothCache Enterprise: adaptive HTTP live streaming in large private networks

Abstract

In this work, we present a distributed caching solution which addresses the problem of efficient delivery of HTTP live streams in large private networks. With our system, we have conducted tests on a number of pilot deployments. The largest of them, with 3000 concurrent viewers, consistently showed that our system saves more than 90% of traffic towards the source of the stream while providing the same quality of user experience of a CDN. Another result is that our solution was able to reduce the load on the bottlenecks in the network by an average of 91.6%.

A extended abstract about this work has been published as:

Roberto Roverso, Sameh El-Ansary, and Mikael Höqvist. *On HTTP live streaming in large enterprises.* In Proceedings of the ACM conference on applications, technologies, architectures, and protocols for computer communication [1]. **SIGCOMM 2013**, Hong Kong, China, 2013

10.1 Introduction

In the last years, the streaming industry has witnessed a shift from the RTP/RTSP standard towards HTTP live, a set of protocols which all utilize HTTP for delivery of live and on-demand content in IP networks [2]. Microsoft, Adobe and Apple have developed players which embrace HTTP-streaming and the concept of adaptive bitrate switching as the main approach for broadcasting. HTTP live has been adopted by content services and creators like Netflix, Hulu and the BBC with support across all platforms and OSs, including computers, tablets and smart phones.

The main challange of delivering HTTP-based live streams is the unicast nature of the HTTP protocol. This creates a potential bottleneck at the source of the stream with a linear increase in bandwidth demand as the number of viewers increases. A natural approach, which is also the primary solution to handle capacity issues for normal HTTP traffic, is to introduce caching. For HTTP live this is the only alternative since there is no multicast support.

While caching of HTTP live via Content Distribution Networks (CDNs) is common for the open Internet, it is challenging to deploy efficiently within private networks such as those operated by corporations or other entities. Larger private networks interconnect multiple network segments representing geographically distributed offices with fixed VPN links. Inside an office, the network is constructed with high capacity links and switches. Traffic from and to the public Internet is routed through a gateway link of fixed capacity or through one or more segments until a gateway link is reached.

The main bottlenecks in private networks are the VPN and gateway links which are typically not dimensioned to sustain the load of delivering one or more streams to a large amount of viewers at the same time. While a private CDN would dramatically improve the efficiency of HTTP live performance, it is hard to deploy and manage since it requires i) new or upgraded caching hardware to support HTTP live ii) that each network segment is covered to handle the bandwidth load of all potential viewers, iii) handling of heterogeneous network infrastructure resulting from network changes such as company acquisitions and mergers.

In this work, we are exploring a software-based CDN for HTTP live in private networks. We leverage the experience acquired in our previous research on improving the efficiency of delivery of HTTP streams over the Internet, where we proposed a peer-to-peer distributed caching approach [3][4]. Based on the same principles, we design an overlay which minimizes inter-segment traffic. In doing so, we enable efficient HTTP live streaming without the need of deploying and managing expensive and specialized hardware. In addition, we evaluate our solution in real-world deployments in private networks which allows us to present unique insights into the problem at hand.

It is paramount to note that the goal of this platform is actually minimizing inter-segment traffic rather than only providing savings towards the source of the stream as in our previous work [3][4].

10.2 Challenges

In this section, we identify a set of challenges and requirements which a distributed caching system must satisfy to be a viable solution for HTTP live streaming in a private network.

Locality and structure awareness. The overlay must be constructed to follow the physical structure of the private network in order to keep traffic within local segments and offload gateway and VPN links.

Bitrate switching. An HTTP live player dynamically switches between different bitrates for the same stream depending on the available bandwidth and host rendering capabilities. Our solution must therefore be able to quickly handle bitrate changes.

Vendor neutrality. Support different players and protocols such as Microsoft's Smooth Streaming, Adobe's HTTP Dynamic Streaming and the upcoming standard MPEG-DASH.

Politeness. The delivery and relaying of content should not interfere with other activities of the user or network traffic generated by critical services.

Finally, in our industrial experience, we have found that **quality of experience (QoE)** is a feature that content owners are not willing to compromise on. Our service should then strive to improve efficiency of distribution while providing QoE which matches the one of private CDNs.

10.3 System Overview

In HTTP live streaming protocol every fragment is fetched as an independent HTTP request that could be scheduled on caching servers. The difference in our solution is that the caching is done at desktop machines instead of dedicated servers. The HTTP player is directed to a local caching agent which acts as an HTTP proxy. All traffic to/from the source of the stream as well as other peers passes by the agent. Upon a content fragment request, the caching agent tries to timely retrieve the data requested by the player from other peers in the overlay. If a fragment cannot be retrieved from any other peer on time, the agent downloads the missing portion of the fragment from the source of the stream, e.g. a public CDN. By falling back to the source, we guarantee that all fragments are delivered on time even if the overlay network cannot retrieve such fragments, thereby guaranteeing the desired level of QoE. This process is engineered to make the agent totally transparent to the player and the streaming infrastructure. In this manner, our platform can support all HTTP-based live streaming protocols.

10.3.1 Overlay Construction

Our distributed caching system is implemented as a self-organizing system based on a mesh overlay network. When joining the system, peers are introduced to other participants by a tracker. After that, they build a random overlay which is used for dissemination of live peer information, e.g. throughput and playback quality. A network segment id is provided by a central registry, with a mapping provided by the network's owner. Peers choose their neighbors by sampling their local view and by ranking peers according to the aforementioned information. Peers make sure to partner with nodes which are retrieving different bitrates, in order to adapt quickly to player bitrate switches.

One or more peers in a segment are promoted to act as live caches for all others in the same segment. The promotion process is implemented either with the help of a locality-aware central service or by means of a distributed K-leader election algorithm similar to [5]. In order to determine the peers to be promoted, we utilize an absolute ranking based on metrics such as computational load, bandwidth and connectivity.

10.3.2 Delivery

Promoted peers are tasked with prefetching content ahead of all other peers in the same segment. The prefetching happens either from the source of the stream or from other nodes outside their segment. We manipulate the prefetching in a way that promoted peers retrieve the stream from the CDN only if their segment has a gateway link, as the content is typically provided by a source external to the private network.

As soon as a fragment is prefetched, other nodes in the segment start to retrieve it from the promoted peer using lower-than-best effort priority [6], as not to interfere with other critical traffic in the network.

A sample delivery overlay is shown in Figure 10.1, promoted peers are highlighted in black, while the others in orange. Arrows denote the traffic's flow across the gateway

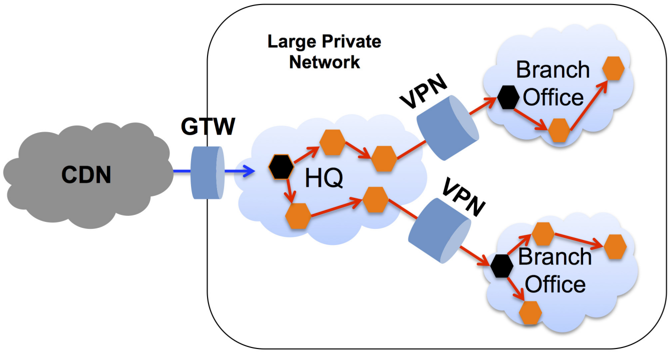


Figure 10.1: HTTP live delivery in a private network with our solution

(GTW) and VPN links, as well as across the network segments.

10.4 Differences between SmoothCache 2.0 and SmoothCache enterprise

SmoothCache 2.0, presented in Chapter 9, and SmoothCache enterprise are essentially the same system. They differ however on these three points:

- **Locality-centric overlay**. In SmoothCache enterprise, the overlay construction mechanism works similarly to SmootchCache 2.0, except that the choice of partners is strongly biased towards peers that are, first, in the same network segment and, then, in the same geographical location or Autonomous system but in other segments.
- **A first tier for each segment**. While in SmoothCache 2.0 there exists a single first tier peer set for all the overlay, in SmoothCache enterprise a different set of first tier peers is promoted for each network segment. The election is carried out in a distributed fashion inside each segment as described in Section 10.3.1.
- **Reactive Prefetching locality**. In SmoothCache enterprise, peers reactively and aggressively prefetch fragments, as in SmoothCache 2.0, but do so only from peers in the same network segment.

10.5 Preliminary results

We currently have a number of deployments in corporations with offices all around the globe. On these installations, we have conducted a series of pilot events using SmoothCache enterprise. In this section, we show a sample event in detail to prove the validity of our approach.

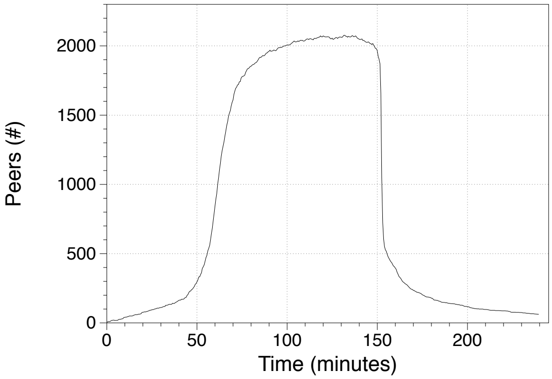


Figure 10.2: Evolution of number of viewers, Real event

10.5.1 About Deployments

Deployments of our application are done following a fairly standard process. First, we deploy our client application on all hosts of the enterprise's network. Second, we agree with the customer on a setup for the streaming chain. This includes a production-quality continuous live stream configured with a single or multi-bitrate profile. The stream is usually published using either a CDN that is located externally to the private network or a streaming server that resides inside the enterprise network. The adaptive HTTP streaming protocols that are used to transport the stream are Smooth Streaming, HLS or HDS. Once all of these components are in place, the live broadcast event can take place. During the event, the viewer is presented with a webpage with an adaptive HTTP streaming player embedded in it. The player is configured to redirect all requests to the SmoothCache proxy.

Before the event and after a series of basic tests, we run large scale tests with the SmoothCache proxy software by operating our software remotely. As part of SmoothCache's functionality, there is also the possibility of starting and stopping the exact same player that the viewers will use during the event, but without rendering of content. The streaming chain used in these instrumented tests is also the same that will be used in the live events. During instrumented tests, we let peers gather detailed statistics which include quality of user experience information, locality and overlay structure snapshots. Since these statistics require peers to send large amount of data, they are turned off during the live event as not to interfere with the broadcast. Instrumented tests are conducted on a larger amount of concurrent peers than the ones expected during the live event. The goal then is to stress test the infrastructure and verify that all potential viewers can access the stream with sufficient QoE.

In the following section, we present the data gathered during a pilot event conducted on a sample deployment in a US-based company. First, we show the statistics collected during the live event, then, we present a more detailed set of statistics obtained from an

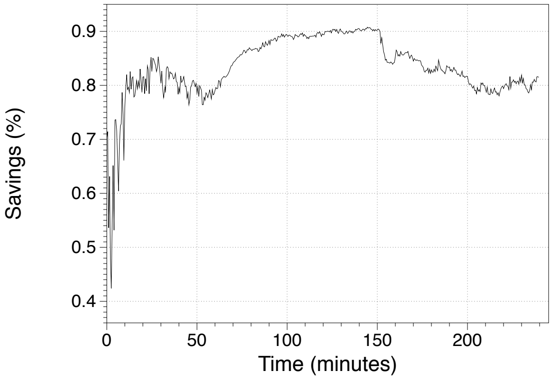


Figure 10.3: Savings towards the source of the stream, Real Event

instrumented test conducted before the event. In all graphs, each data point represents the aggregate value of the metrics in a 30 seconds period of time.

10.5.2 Pilot Event: US-based company

The US-based deployment comprised 48.000 installations on 89 network segments. The stream was broadcasted on a single bitrate ($1.4Mbps$) configuration from a CDN outside of the enterprise network using the standard Microsoft SmoothStreaming tool chain.

In Figure 10.2, we present the evolution of the amount of viewers over time. The stream started at time 0, while the event began at the 50th minute. As we can see, the peers start to join earlier than the start of the event and then the number raises exponentially to a maximum of around 2000 concurrent viewers. The number stays more or less constant until the end of the event at minute 145. We accounted for 3529 unique viewers in total located in 69 network segments.

Figure 10.3 shows the evolution of savings in time towards the CDN, that is the amount of data which the peers downloaded from the overlay network over the amount of data downloaded for playback. Since the stream was hosted externally to the private network, this metric also represents how much we offloaded the gateway links to the CDN. In total, there were 3 gateways, one for western USA, one for central USA and one for eastern USA. From the figure, it is clear that even a small amount of viewers in the beginning can lead to high savings, around 80%, while, after the event started, the savings reach and remain stable at around 90%. The total amount of data consumed by the viewers was 1720.45 gigabytes, 87.5% was served from the overlay and 12.1% from the external CDN.

Regarding quality of user experience, we present in Figure 10.4, the evolution of average buffering time for all peers over time. In this graph, each data point is the buffering time experienced by the peers in the network in a 30 seconds time period. This measure is reported directly by the SmoothStreaming player to our software. The buffering time

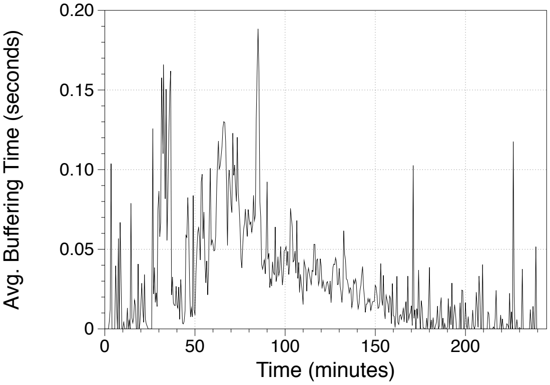


Figure 10.4: Average buffering time over all peers, Real Event

measured here includes the initial buffering time and the time the player stopped the playback because of starvation. As we can see, between time 0 and 100, when all peers join, the buffering time reaches a maximum of 0.2 seconds and then becomes much lower. The reason for that is that, in that time period, we have a large number of peers starting to watch the stream and therefore buffering before starting to render the content. This is by far the major contributor to the buffering time, while the buffering due to starvation (after time 100) is very low.

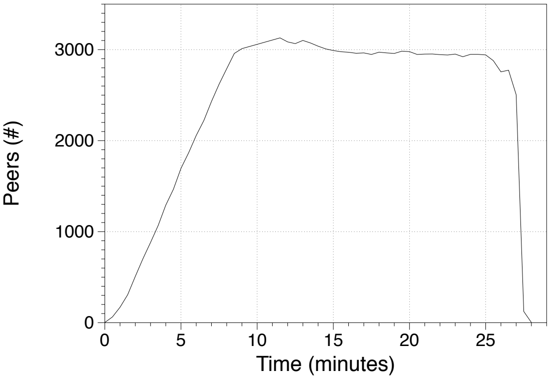


Figure 10.5: Evolution of number of viewers, Instrumented Test

Instrumented Test

We present now a test conducted before the event with around 3100 machines. In Figure 10.5 we show the evolution of number of peers running the tests over time. All peers join in the first 8 minutes of the test and then their number stays more or less constant over the duration of the test until the test finishes at time 25 minutes.

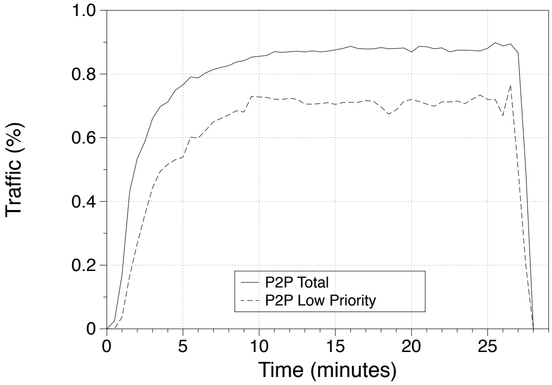


Figure 10.6: Savings towards the source of the stream, Instrumented Test

Figure 10.6 shows the savings evolution of the test as the percentage of P2P traffic (P2P Total) over the total amount of video traffic. As we can see, the savings are similar to the ones of the live event. In the same graph, we can also see the percentage of the video content that was transferred using lower than best effort priority (P2P Low Priority), as explained in 10.3.2.

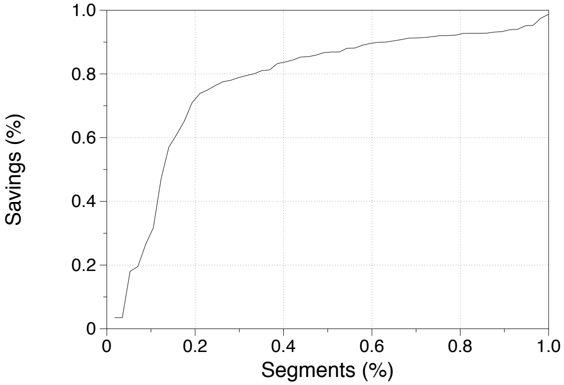


Figure 10.7: Average savings CDF for all network segments, Instrumented Test

In order to understand how the savings were distributed over the network segments, we present in Figure 10.7 the CDF distribution of average savings for all segments in the network. In the great majority of the segments (70%), the savings were higher than 80%, while in more than 50% of the segments, the savings were more than 90%.

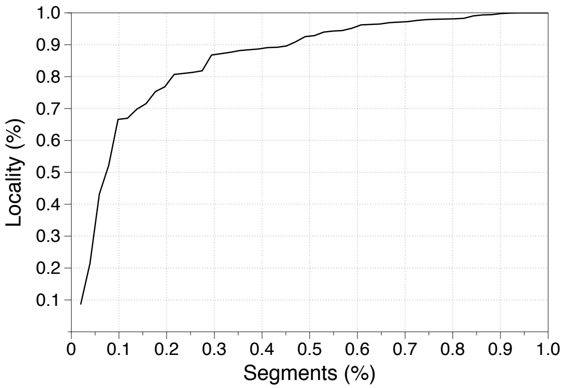


Figure 10.8: Locality percentage CDF for all network segments, Instrumented Test

We then analyze the peer-to-peer traffic by looking into the percentage of locality, that is the quantity of peer-to-peer data retrieved from peers in one segment over all the peer-to-peer data downloaded by peers in that same segment. Figure 10.8 shows the CDF of the percentage of locality for all segments. Most of the segments, that is more than 70%, have locality of over 90%, while 20% of the segments have locality close to 100%. That means that *only* the leader(s) of the segment are retrieving the content from the CDN, or alternatively from another segment, and providing for the overlay inside the segment.

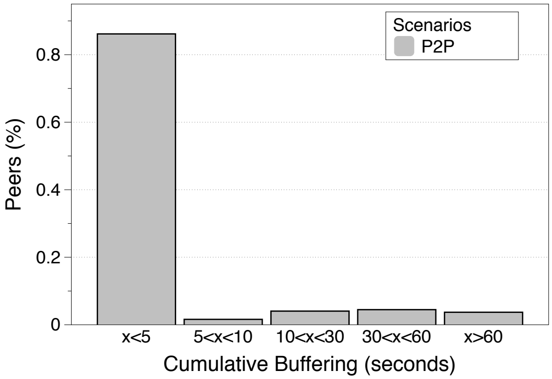


Figure 10.9: Cumulative buffering time classification, Instrumented Test

Regarding quality of user experience, we consider the cumulative delay experienced by peers during the total duration of the test. In Figure 10.9, we present the classification of the cumulative delay. Results show that 86% of the clients experienced a cumulative buffering delay of less than 5 seconds.

In Figure 10.10, we show a visual representation of the delivery overlay network for the instrumented test. The hexagons represent the network segments in the enterprise network. The size of an hexagon is proportional to the number of viewers who watched from the network segment. The gray lines represent the edges of the delivery network, that is a gray line exists if data was exchanged between two segments. The thickness of an edge represents instead the amount of data transferred, that is, the thicker the line, the larger the amount of traffic.

As we can see in the figure, there are five main clusters of network segments. The segments are clustered considering the number of transfers and the amount of data transferred between the segments. In most cases, the clustering also matches the geographical location of the segments (observed from the labels in the overlay graph). The main cluster in the center of the figure is made of offices that are located mostly in the south-eastern part of the United states. The clusters on the left side and the right-lower side of the picture contain offices from the central part of the US. Instead, the ones on the right and at the bottom of the picture contain network segments located mostly on the south and north-eastern parts of the States, respectively. We can also note that there are very few edges between different clusters, indicating that the locality policies in SmoothCache do actually work as expected when it comes to geographical location.

Finally, in the figure, we find five network segments which are not connected to any other segment. This was due to the fact that those segments were behind very restrictive firewalls which did not allow them to communicate with other branches of the company.

10.6 References

[1] Roberto Roverso, Sameh El-Ansary, and Mikael Hoeqvist. On http live streaming in large enterprises. In *Proceedings of the ACM SIGCOMM 2013 conference on Applications, technologies, architectures, and protocols for computer communication*, SIGCOMM '13, New York, NY, USA, 2013. ACM.

[2] Saamer Akhshabi, Ali C. Begen, and Constantine Dovrolis. An experimental evaluation of rate-adaptation algorithms in adaptive streaming over HTTP. In *Proc. of ACM MMSys*, 2011.

[3] Roberto Roverso, Sameh El-Ansary, and Seif Haridi. Smoothcache: Http-live streaming goes peer-to-peer. In *Proc. of IFIP NETWORKING*, 2012.

[4] Roberto Roverso, Riccardo Reale, Sameh El-Ansary, and Seif Haridi. Smoothcache 2.0: the http-live peer-to-peer cdn. In *Report*, volume 7290 of *Peerialism White Papers*, 2013.

[5] Vaskar Raychoudhury, Jiannong Cao, and Weigang Wu. Top k-leader election in wireless ad hoc networks. In *Proc. of IEEE ICCCN*, 2008.

[6] Riccardo Reale, Roberto Roverso, Sameh El-Ansary, and Seif Haridi. DTL: Dynamic Transport Library for Peer-To-Peer Applications. In *Proc. of ICDCN*, 2012.

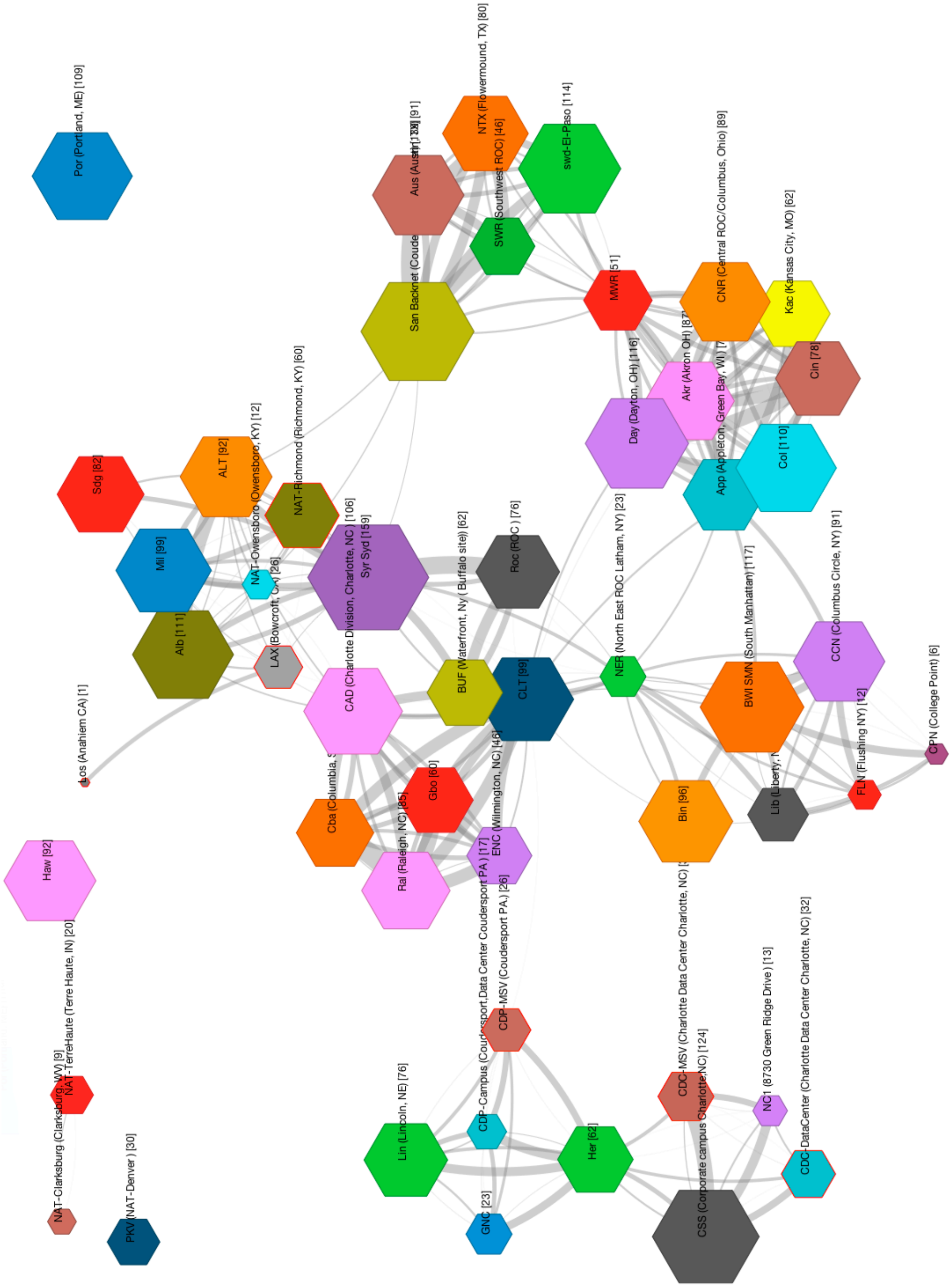

Por (Portland, ME) [109]
Aus (Austin [91]
NTX (Flowermound, TX) [80]
SWR (Southwest ROC) [46]
swd-El-Paso [114]
San Backnet (Coude
MWR [51]
CNR (Central ROC/Columbus, Ohio) [89]
Kac (Kansas City, MO) [62]
Cin [78]
Akr (Akron OH) [87
Day (Dayton, OH) [116]
App (Appleton, Green Bay, WI) [7
Col [110]
Sdg [82]
ALT [92]
NAT-Owensboro (Owensboro, KY) [12]
NAT-Richmond (Richmond, KY) [60]
Mil [99]
Alb [111]
LAX (Bowcroft, CA) [26]
CAD (Charlotte Division, Charlotte, NC) [106]
Syr Syd [159]
BUF (Waterfront, Ny (Buffalo site)) [62]
Roc (ROC) [76]
CLT [99]
NER (North East ROC Latham, NY) [23]
BWI SMN (South Manhattan) [117]
CCN (Columbus Circle, NY) [91]
CPN (College Point) [6]
FLN (Flushing NY) [12]
Lib (Liberty, N
Bin [96]
Los (Anahiem CA) [1]
Cba (Columbia, S
Gbo [60]
Ral (Raleigh, NC) [85]
ENC (Wilmington, NC) [46]
Haw [92]
CDP-Campus (Coudersport,Data Center Coudersport PA) [17]
CDP-MSV (Coudersport PA.) [26]
CDC-MSV (Charlotte Data Center Charlotte, NC) [
CSS (Corporate campus Charlotte,NC) [124]
NC1 (8730 Green Ridge Drive) [13]
CDC-DataCenter (Charlotte Data Center Charlotte, NC) [32]
NAT-TerreHaute (Terre Haute, IN) [20]
NAT-Clarksburg (Clarksburg, WV) [9]
PKV (NAT-Denver) [30]
Lin (Lincoln, NE) [76]
GNC [23]
Her [62]

Figure 10.10: Overlay visualization, Instrumented Test

CHAPTER 11

CONCLUSIONS

Video streaming accounts for a large portion of the global Internet traffic. Content providers cope with the demand for online video content by relying on Content Delivery Networks. As demand for online video is bound to increase significantly in the coming years and as competition between CDN services becomes fiercer, we argue that the CDN industry will likely turn to more cost effective solutions based on peer-to-peer overlays. In fact, we already observe some early signs of that trend for the distribution of video on demand content with companies such as Akamai pushing for adoption of NetSession [1], a peer-assisted content distribution software, and Adobe enabling support for peer-to-peer video delivery in the Flash framework [2]. These solutions also provide out-of-the-box support for adaptive HTTP streaming protocols which are quickly becoming the de-facto standard for video distribution on the Internet. All major actors in the online broadcasting business, such as Microsoft, Adobe and Apple, have developed technologies which embrace HTTP as transport protocol for streaming and the concept of adaptive bitrate switching. At the same time, most of the content services providers have switched to adaptive HTTP streaming, for instance Netflix, Hulu and the BBC, and they have also added support for the new protocols across all platforms and OSs, including desktop computers, tablets and smart phones.

In this thesis, we have explored the possibility of leveraging peer-to-peer overlays to offload CDNs when streaming live content with adaptive HTTP streaming protocols. The result of our effort is a system called SmoothCache which is the first peer-assisted solution for the distribution of adaptive bitrate HTTP live streams. Following the peer-assisted approach, our system leverages the existing Content Delivery Network infrastructure, as provided and without the need for integration, to assist the peer-to-peer delivery and provide a target quality of user experience (QoE) that is the same of a CDN.

In order to build SmoothCache, we have developed a number of components which are in their own merit novel and each of them improves on the existing state of the art in its respective research field.

We have first implemented a state-of-the art framework for the development of peer-to-peer applications called Mesmerizer [3]. The framework follows the semantics of the actor model [4], similarly to Erlang [5], Scala [6] and Kompics [7], and lets developers implement their distributed application as a set of of components which communicate using events. The framework enables the execution of the developed code both in simulation and in a real deployment. In simulation, we execute experiments on an emulated network where we accurately model multiple characteristics of physical networks, such as the presence of Network Address Translators, network delay patterns and bandwidth allocation dynamics. In order to improve scalability and accuracy of simulations in our framework, we designed a novel flow-based bandwidth emulation model [8] which combines the max-min fairness bandwidth allocation algorithm [9] with a method called affected subgraph optimization [10]. The result is a model which provides accuracy that is close to that of packet-level simulators but with much higher scalability.

For deployment of SmoothCache on real networks, we developed a novel network library called *DTL* [11] which provides reliability and on-the-fly prioritization of transfers. The library is based on the UDP protocol and enables intra- and inter-protocol prioritization by combining two state of the art congestion control algorithms: LEDBAT [12] and MulTCP [13]. DTL supports a range of priority levels, that is from less-than-best-effort (LEDBAT) priority up to many times the priority of TCP. Priority can be configured at runtime by the application without disruptions in the flow of data and without the need of connection re-establishment.

DTL also includes support for state-of-the-art Network Address Translation traversal methods, which were also studied as part of this thesis [14]. Our work on NAT Traversal relies on a detailed classification of NAT behavior which comprises of 27 different NAT types. In contrast to all previous work on the subject, we argue that it is incorrect to reason about traversing a single NAT, instead combinations must be considered. Therefore, we provide a comprehensive study which states, for every possible NAT type combination, whether direct connectivity with no relaying is feasible. On top of that, we also define which specific NAT traversal technique should be used for each traversable combination. Our approach results in a traversal success probability that is 85% on average in the tested scenarios and therefore much higher than previous approaches.

Because of the presence of NATs in the network, distributed discovery of new peers and collection of statistics on the overlay through peer sampling is problematic. That is because the probability of establishing direct connections with peers varies from one peer to the other. To work around this limitation, we have designed a novel peer sampling service, the *Wormhole peer sampling service* (WPSS) [15]. WPSS builds a slowly changing overlay network and executes short random walks on top of that overlay. A descriptor containing information on a peer is originated at a node and then carried by a random walk until it is deposited when the walk terminates. WPSS improves on the state of the art by one order of magnitude in terms of connection established per unit time while providing the same level of freshness for samples. This is achieved without sacrificing the desirable properties of a PSS for the Internet, such as robustness to churn and NAT-friendliness.

All of the aforementioned components are used as part of our SmoothCache system and are employed in commercial deployments. Regarding SmoothCache itself, we have designed our system as a distributed cache where every peer acts as a caching node for live content. The main motivation behind this design is to cope with the mode of operation of adaptive HTTP streaming where timeliness is kept not by skipping video fragments but rather by changing bitrates. For that reason, in our system, when the player requests a fragment, the data is either retrieved from the neighbors of a peer or otherwise downloaded directly from the CDN in order to guarantee its delivery. Our system is completely oblivious to the streaming protocol and, as a consequence, it never tries to manipulate the player behavior such as to skip fragments or force the playback of a specific bitrate. That is in direct contrast with classical peer-to-peer systems where the player merely renders content and the P2P agent decides which video chunks should be played and which chunks should be dropped.

Our first take on the design of the SmoothCache system, that is a distributed caching scheme for the Internet tailored to adaptive HTTP live streaming, showed that it is possible to achieve significant savings towards the source of the stream by using peer-to-peer overlays [16]. In the second iteration of our distributed caching system [17], we were able to match the same quality of user experience levels of a CDN and achieve even higher savings. This was possible by letting a small subset of powerful peers, that is first tier peers, act similarly to CDN nodes. Those nodes aggressively prefetch fragments as they are created by the streaming server. We make sure, then, that all prefetched data is propagated in the network within a small window of time. This window should not be higher than the delay that viewers would experience by accessing a CDN directly. In case some fragments are not retrieved in that time window, the nodes download the data directly from the CDN. Fast propagation in the peer-to-peer network during a small window of time is achieved in Smoothcache by employing two techniques: i) a mesh-based overlay structure which enforces a hierarchy based on upload bandwidth but additionally takes into consideration other factors like connectivity constraints, performance history and currently watched bitrate, ii) a mix of proactive and reactive prefetching strategies with various levels of aggressiveness using an application-layer congestion control for balancing high dissemination efficiency and politeness.

The performance of SmoothCache was evaluated in a thorough experimental study conducted on a subset of 22000 of the around 400000 installations of our software worldwide. The study showed that SmoothCache matches the QoE of the CDN on all of the three considered metrics: cumulative playback delay, delivered bitrate and delay from the playback point. Besides this result, SmoothCache was also able to consistently deliver up to 96% savings towards the source of the stream in single bitrate scenarios and 94% in multi-bitrate scenarios.

Finally, we were able to adapt our Smoothcache system to distribute adaptive HTTP live content in large enterprise networks [18]. In this setting, links between the branches of the enterprise and towards the rest of the Internet constitute a bottleneck during live streaming of events. We have conducted a number of pilot deployments of the resulting system and, in this work, we presented in detail one of them to prove the validity of our approach. For the considered live broadcast event, our software was installed on 48.000 machines on 89 network branches of a US-based company. The event involved 3529 unique viewers, with a maximum of around 2000 of them watching concurrently. Collected statistics show that SmoothCache was able to save up to 87.5% of the traffic towards the source of the stream and it also offloaded the majority of the enterprise branches links of more than 80%.

11.1 Future Work

As future work, we would like to replace the tracker-assisted first tier election process of SmoothCache 2.0 with a completely distributed solution. For that purpose, we intend to use an approach similar to Absolute Slicing [19]. This algorithm enables the assignment of peers to groups or *slices* of a specific size in a purely distributed manner. The assignment is maintained under churn, and membership to one or more slices is decided considering a set of predefined metrics. In SmoothCache, we would use Absolute Slicing to form a slice which contains all possible first tier candidates.Then, we would let peers in that group estimate how many of the candidates should become first tier peers.

This can be done by re-designing the estimation heuristic presented in [17] to work with statistics collected by our peer sampling layer, such as upload bandwidth distribution. Finally, the promotion of the estimated number of first tier peers would happen by creating another slice containing only the candidates which are allowed to prefetch from the source of the stream.

We believe that a natural continuation of this work would be adding support for video on demand adaptive HTTP streaming to SmoothCache, similarly to Maygh [20]. For that purpose, we would need to transform what is now a sliding window caching system into a persistent caching system. The first step of the process is to implement local persistent storage of fragments. The next step is then to build a light-weight distributed indexing mechanism for spreading information about piece availability of multiple videos. Finally, we would need to build a replication mechanism which strives to maintain enough copies of a video in order to maximize availability and performance. In line with the peer-assisted approach, we would rely on an existing infrastructure in case the content is not present in the peer-to-peer network. Once video-on-demand is in place, caching of general content might be considered as a possible extension to our platform.

Finally, it would be extremely interesting to apply the Software-defined Networking (SDN) paradigm [21] to improve efficiency in the delivery of adaptive HTTP live streaming. SDN-enabled network infrastructure decouples the the control plane, which decides on where the traffic should be sent, from the data plane, which forwards the traffic according to instructions from the control plane. Typically, the control plane does not reside on routers or switches but on a centralized controller which communicates remotely with the data plane using the OpenFlow protocol [22]. This simplifies tremendously network management since the centralized infrastructure allows for efficient monitoring and dynamic re-configuration of the data plane [23]. Recently, we have observed a new trend of applying the SDN principle to video distribution with RASP [24]. There authors utilize a mixture of SDN techniques and overlay networks to improve locality of traffic in RTSP/RTP streaming. In the near future, we would like to apply the same approach to the delivery of adaptive HTTP live streaming content, this both on the Internet and in the setting of large private networks.

11.2 References

[1] Mingchen Zhao, Paarijaat Aditya, Ang Chen, Yin Lin, Andreas Haeberlen, Peter Druschel, Bruce Maggs, Bill Wishon, and Miroslav Ponec. Peer-assisted content distribution in akamai netsession. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 31–42. ACM, 2013.

[2] Adobe. Rtmfp for developing real-time collaboration applications, 2013. URL `http://labs.adobe.com/technologies/cirrus/`.

[3] Roberto Roverso, Sameh El-Ansary, Alexandros Gkogkas, and Seif Haridi. Mesmerizer: a effective tool for a complete peer-to-peer software development life-cycle.

In *Proceedings of the 4th International ACM/ICST Conference on Simulation Tools and Techniques*, SIMUTools '11, pages 506–515, ICST, Brussels, Belgium, Belgium, 2011.

[4] Gul Abdulnabi Agha. Actors: a model of concurrent computation in distributed systems. MIT Press, 1985.

[5] Joe Armstrong. Erlang - a survey of the language and its industrial applications. In *In Proceedings of the symposium on industrial applications of Prolog (INAP96). 16 18*, 1996.

[6] Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, 410(2-3):202–220, 2009. ISSN 0304-3975.

[7] C. Arad, J. Dowling, and S. Haridi. Building and evaluating p2p systems using the kompics component framework. In *Peer-to-Peer Computing, 2009. P2P '09. IEEE Ninth International Conference on*, pages 93 –94, 2009.

[8] Alexandros Gkogkas, Roberto Roverso, and Seif Haridi. Accurate and efficient simulation of bandwidth dynamics for peer-to-peer overlay networks. In *Proceedings of the 5th International ICST Conference on Performance Evaluation Methodologies and Tools*, VALUETOOLS '11, pages 352–361, ICST, Brussels, Belgium, Belgium, 2011. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[9] Anh Tuan Nguyen and F. Eliassen. An efficient solution for max-min fair rate allocation in p2p simulation. In *ICUMT '09: Proceedings of the International Conference on Ultra Modern Telecommunications Workshops*, pages 1 –5, St. Petersburg, Russia, October 2009.

[10] F. Lo Piccolo, G. Bianchi, and S. Cassella. Efficient simulation of bandwidth allocation dynamics in p2p networks. In *GLOBECOM '06: Proceedings of the 49th Global Telecommunications Conference*, pages 1–6, San Franscisco, California, November 2006.

[11] Riccardo Reale, Roberto Roverso, Sameh El-Ansary, and Seif Haridi. Dtl: dynamic transport library for peer-to-peer applications. In *Distributed Computing and Networking*, pages 428–442. Springer, 2012.

[12] Ledbat ietf draft. `http://tools.ietf.org/html/draft-ietf-ledbat-congestion-03`, July 2010.

[13] Masayoshi Nabeshima. Performance evaluation of multcp in high-speed wide area networks. *IEICE Transactions*, 88-B(1):392–396, 2005.

[14] Roberto Roverso, Sameh El-Ansary, and Seif Haridi. NATCracker: NAT Combinations Matter. In *Proceedings of 18th International Conference on Computer Communications and Networks 2009*, ICCCN '09, pages 1–7, San Francisco, CA, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-4581-3.

[15] R. Roverso, J. Dowling, and M. Jelasity. Through the wormhole: Low cost, fresh peer sampling for the internet. In *Peer-to-Peer Computing (P2P), 2013 IEEE 13th International Conference on*, 2013.

[16] Roberto Roverso, Sameh El-Ansary, and Seif Haridi. Smoothcache: Http-live streaming goes peer-to-peer. In Robert Bestak, Lukas Kencl, LiErran Li, Joerg Widmer, and Hao Yin, editors, *NETWORKING 2012*, volume 7290 of *Lecture Notes in Computer Science*, pages 29–43. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-30053-0.

[17] Roberto Roverso, Riccardo Reale, Sameh El-Ansary, and Seif Haridi. Smoothcache 2.0: the http-live peer-to-peer cdn. In *Report*, volume 7290 of *Peerialism White Papers*, 2013.

[18] Roberto Roverso, Sameh El-Ansary, and Mikael Hoeqvist. On http live streaming in large enterprises. In *Proceedings of the ACM SIGCOMM 2013 conference on Applications, technologies, architectures, and protocols for computer communication*, SIGCOMM '13, New York, NY, USA, 2013. ACM.

[19] A. Montresor and R. Zandonati. Absolute slicing in peer-to-peer systems. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8, 2008.

[20] Liang Zhang, Fangfei Zhou, Alan Mislove, and Ravi Sundaram. Maygh: Building a cdn from client web browsers. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 281–294. ACM, 2013.

[21] Nick McKeown. Software-defined networking. *INFOCOM keynote talk, Apr*, 2009. URL `http://www.cs.rutgers.edu/~badri/552dir/papers/intro/nick09.pdf`.

[22] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38 (2):69–74, 2008.

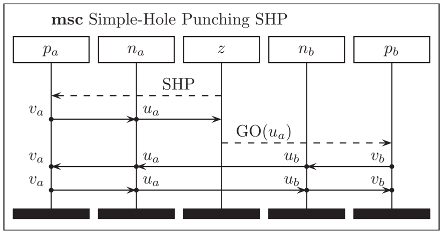
[23] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: experience with a globally-deployed software defined wan. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, SIGCOMM '13, pages 3–14, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2056-6. URL `http://doi.acm.org/10.1145/2486001.2486019`.

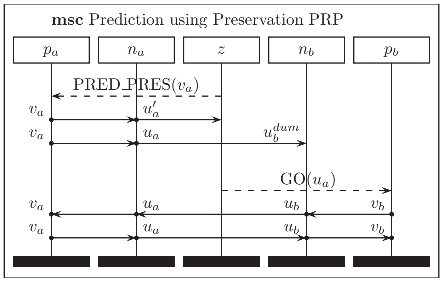
[24] David Hausheer Julius Ruckert, Jeremias Blendin. Rasp: Using openflow to push overlay streams into the underlay. In *13th IEEE International Conference on Peer-to-Peer Computing*, University of Trento, Italy, September 2013.

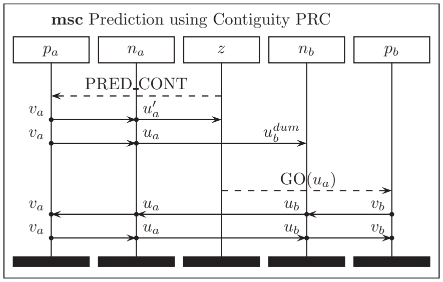
Appendices

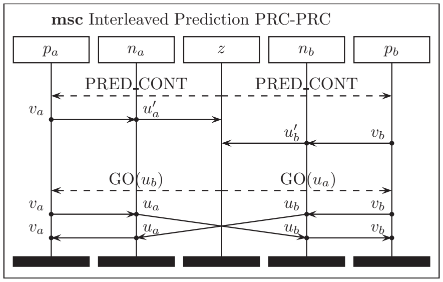
CHAPTER 12

TRAVERSAL SEQUENCE DIAGRAMS


msc Simple-Hole Punching SHP
p_a
n_a
z
n_b
p_b
SHP
v_a
u_a
GO(u_a)
v_a
u_a
u_b
v_b
v_a
u_a
u_b
v_b


msc Prediction using Preservation PRP
p_a
n_a
z
n_b
p_b
PRED_PRES(v_a)
v_a
u'_a
v_a
u_a
u_b^{dum}
GO(u_a)
v_a
u_a
u_b
v_b
v_a
u_a
u_b
v_b


msc Prediction using Contiguity PRC
p_a
n_a
z
n_b
p_b
PRED_CONT
v_a
u'_a
v_a
u_a
u_b^{dum}
GO(u_a)
v_a
u_a
u_b
v_b
v_a
u_a
u_b
v_b


msc Interleaved Prediction PRC-PRC
p_a
n_a
z
n_b
p_b
PRED_CONT
PRED_CONT
v_a
u'_a
u'_b
v_b
GO(u_b)
GO(u_a)
v_a
u_a
u_b
v_b
v_a
u_a
u_b
v_b