Efficient Content Delivery in P2P Networks

Alexander Libov
Efficient Content Delivery in P2P Networks

Research Thesis

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Alexander Libov

Submitted to the Senate of the Technion — Israel Institute of Technology
Kislev 5776 Haifa November 2015
This research was carried out under the supervision of Prof. Roy Friedman, in the Faculty of Computer Science.

Some results in this thesis have been published as articles by the author and research collaborators in conferences and journals during the course of the author’s doctoral research period, the most up-to-date versions of which being:


ACKNOWLEDGEMENTS

First, I would like to thank my advisor, Prof. Roy Friedman. Thank you for years of guidance and support. I am very glad and grateful for the opportunity to work with you under your supervision. I would also like to thank Prof. Ymir Vigfusson for the collaboration and help.

I would like to thank my friends from the faculty, you have made these years fun and unforgettable. I sincerely hope that we will be able to keep in touch when we all go on our separate ways.

Last but not least, I would like to thank my family and especially my wife for supporting me throughout my many years in the Technion and beyond.

The generous financial help of the Technion and the HPI Research School is gratefully acknowledged.
Contents

List of Figures

Abstract 1

1 Introduction 3

  2.1 Related Work 10
  2.2 Background and Model 11
  2.3 MOLSTREAM Architecture 12
  2.4 Libraries 14
    2.4.1 Overlay modules 14
    2.4.2 Streaming Protocols 16
    2.4.3 System Parameters 17
    2.4.4 Supported Performance Metrics 19
  2.5 Case Studies 20
  2.6 Discussion and Conclusion 24

3 Distilling the Ingredients of P2P Live Streaming Systems 27
  3.1 Modules and Ingredients 28
    3.1.1 Ingredients of P2P Live Streaming 28
    3.1.2 Discussion 31
    3.1.3 Ingredients in STREAMAID 31
  3.2 Expressing Design Decisions 32
    3.2.1 Player Module 32
    3.2.2 Streaming Module 33
    3.2.3 Overlay Module 37
    3.2.4 Network Module 39
  3.3 Evaluation 39
    3.3.1 Setup and Measurements 39
    3.3.2 Case Study: BitTorrent Live 41
    3.3.3 Case Study: Pull-based Streaming 42
# Chapter 4: A Dynamically Adaptive Framework for P2P Live Streaming

## 4.1 A MAB Primer

- **ε-greedy algorithm**
- **The UCB1 algorithm**

## 4.2 Applying MAB to P2P Live Streaming

- **Selecting Candidates**
- **General MAB Approach**
- **Switching Between Protocols**

## 4.3 Evaluation

- **Experimental Setup**
- **Different Algorithms Perform Better in Different Settings**
- **The Cost of Switching Algorithms**
- **Best Algorithm for Environment Conditions**
- **MAB Reward Function**
- **Convergence of MAB algorithms**

## 4.4 Related Work

## 4.5 Discussion and Conclusion

# Chapter 5: An Advertising Mechanism for P2P Networks

## 5.1 Overview

- **Goals**
- **Basic Concepts**
- **Advertiser node**
- **Advertisement serving**

## 5.2 Design Decisions

- **Payment Models**
- **Dissemination Strategies**

## 5.3 Dissemination tracking

- **The Mechanism**
- **Security Analysis**

## 5.4 Performance Evaluation

- **Model and setup**
- **Definitions**
- **Results**

## 5.5 Related Work

## 5.6 Discussion

# Chapter 6: Conclusions
List of Figures

2.1 **Architecture.** An illustration of the components present in the MOLSTREM implementation and the interfaces between them. .......................... 13
2.2 Average degree of Coolstreaming with different overlays ....................... 21
2.3 **Case study of Coolstreaming.** The continuity index and the overhead of a MOLSTREM Coolstreaming implementation protocol in the MOLSTREM compared to the original results of the Coolstreaming protocol. ............... 22
2.4 Continuity index and one SD for various sizes of networks of PlanetLab nodes 23
2.5 **Araneola overlay.** (a) and (b) Average node degree as a function of uptime of a node (c) Average latency of different $L$ settings. .......................... 23
2.6 Comparison of Coolstreaming using PeerSim simulator, Cluster and Planetlab testbed .................................................. 24
2.7 Overhead and latency of mTreebone with different stable coefficients. ....... 24

3.1 **Micro-Layers:** a model where messages pass through a stack of thin layers for transmission and reception on the destination. .......................... 28
3.2 **STREAMAID Module:** A module is composed of a set of ingredients in STREAMAID. ................................................ 29
3.3 **STREAMAID Architecture:** Each independent module contains a queue of ingredients. ................................................ 30
3.4 We test our implementation of the BitTorrent Live protocol with startup buffer of 2 sec. Figure 3.4(b) shows different missing chunk handling methods. In Figure 3.4(a) we use the skip ingredient for handling missing chunks and test various download connection ranges. In Figure 3.4(c) we use the skip ingredient for handling missing chunks, 2-3 parent nodes and test different latency ranges for the Latency Range adaptive playout ingredient. Note that a higher Continuity Index and lower Latency and Duplicate% are better. We find that **always skipping or always waiting for a missing chunk is better than waiting and then skipping.** Also, the balance between Continuity Index and Duplicate chunks through number of in-club download connections as well as the trade-off between Latency and the Continuity Index through Adaptive Playout are evident. .......................... 40
3.5 **BitTorrent Live built from ingredients.** The BitTorrent Live protocol decomposed into ingredients as implemented in STREAMAID.  

3.6 **CoolStreaming source push.** Source push evaluation of CoolStreaming using the Araneola Overlay \((L = 3 \text{ and } 4)\) and the original CoolStreaming overlay \((M = 4)\) waiting for missing chunks and skipping missing chunks. All algorithms use startup buffer of 2 seconds and start buffering from the first bitmap or chunk. **Enabling the source push ingredient can only improve performance.**

3.7 **CoolStreaming/Araneola Buffer starting point.** Evaluation of CoolStreaming using the Araneola Overlay \((L = 3 \text{ and } 4)\) and the original CoolStreaming overlay \((M = 4)\) while waiting for missing chunks. All algorithms use startup buffer of 2 seconds and use source push. If the algorithm skips missing chunk (not shown), more than 80% of the users fail to play any chunks at all, and those who do have playback with shoddy continuity. When waiting for missing chunks, initializing the player module from startup reduces latency marginally but at the cost of slightly reduced Continuity Index and greatly reduced Perfect Continuity Index%.

3.8 **CoolStreaming streaming module** using the Araneola \((L = 3)\) and CoolStreaming overlays \((M = 4)\) with startup buffers of 1 and 2 sec and varying values of \(k\) (initial position). All algorithms start buffering from the first bitmap or chunk, use source push and wait if there is no chunk to play. **Initializing the player to start at a chunk earlier than the last one available strikes a trade-off between the Perfect Continuity Index\% and the Latency while keeping the Startup Delay fixed.**

3.9 **Startup delay, Latency, Continuity Index and Perfect Continuity Index\% of CoolStreaming streaming module using Araneola Overlay (L=3) and CoolStreaming overlay (M=4) waiting for missing chunks and skipping missing chunks with various startup buffer sizes.** The algorithms start buffering from the first bitmap or chunk and use source push. **CoolStreaming streaming module with the Araneola overlay module which waits for missing chunks both gets a very high Continuity Index and low Latency and Startup Delay with a startup buffer of only 1 second.**

3.10 **Startup Delay, Latency, Continuity Index and Perfect Continuity Index\% of CoolStreaming streaming module using Araneola Overlay (L=3) and CoolStreaming overlay (M=4) with startup buffer of 1 and 2 seconds and several settings of number of seconds to wait before skipping a chunk.** The algorithms start buffering from the first bitmap or chunk and use source push. **There is not much difference between \(n > 0\) and always waiting for missing chunks, waiting is even preferable.**
3.11 **CoolStreaming/Araneola Adaptive Playout.** Startup Delay, Latency and Continuity Index and Perfect Continuity Index% of CoolStreaming streaming module using Araneola Overlay ($L = 3$) with startup buffers of 1 sec, using the Adaptive Playout ingredient with various target buffer sizes and several playback initial position values. All algorithms start buffering from the first bitmap or chunk, use source push and wait if there is no chunk to play. We see that increasing the target buffer behaves in a similar fashion as increasing $k$ (the player module initialization position) and both methods can be combined to reach near optimal values of Perfect Continuity Index.

4.1 **SMARTSTREAM overview.** The framework uses the state of the environment (such as high churn rates) to choose between different instantiations of a MAB algorithm. The selected MAB algorithm in turn maintains internal weights (or bounds) on a set of potential live streaming algorithms that may be used. These weights are adjusted based on “rewards” from the environment, which are success rates of desirable P2P live streaming metrics using the chosen algorithm. An algorithm with worse streaming performance than another will reap fewer rewards, thus decreasing the probability of it being chosen by the framework in similar environmental states in the future.

4.2 **Switching between protocols.** Steps taken by SMARTSTREAM nodes to change between different live streaming protocols.

4.3 **Algorithms used for comparison.** We combined various P2P algorithm components to produce a set of different compositions for experimental evaluation.

4.4 **Stream quality in different environments.** The clear stream ratio and latency trade-offs of the algorithms in different system states. Push-pull algorithms prevail in both low churn and low upload bandwidth settings.

4.5 **Protocol switching costs.** Comparison of the continuity index for a setting where SMARTSTREAM switches between two algorithms. We change the number of rounds ($K$) the nodes retain state for both the current and previous algorithms.

4.6 **Continuity of adaptive vs. static approaches under dynamic bandwidth.** Comparison of the continuity index of SMARTSTREAM versus each of the leading algorithms of every state. The upload bandwidth of peers is 12.6Mbit until the 600 second mark and new joining peers have 2.6Mbit upload bandwidth afterwards. The churn is set to high. Each point in the graph is a weighted average of the continuity index over the preceding 100 seconds.
4.7 **Continuity of adaptive vs. static approaches under dynamic bandwidth.**
Comparison of the continuity index of each of the leading algorithms of every state with SMARTSTREAM’s performance. The upload bandwidth of new joining peers is 12.6Mbit for until the 600 second mark, then 2.6Mbit for 1200 seconds, then again 12.6Mbit between 1800 and 2400 second marks and again 2.6Mbit for the last 1200 seconds. The churn is set to high. Each point in the graph is a weighted average of the preceding 100 seconds.  

65

4.8 **Approximating the global continuity index.** Comparison of several approximation algorithms using different measurements for reward. We see that the local CI is the main contributor to the reward and that using all metrics improves the correlation coefficient.  

66

4.9 **Adaptivity.** Continuity index of SMARTSTREAM working with different multi-armed bandit algorithms while adapting to new environmental configuration. Each subfigure shows the two static algorithms in the high churn low upload bandwidth setting and one type of MAB function with the default choice being either the leading algorithm or the second-strongest.  

67

5.1 The entities in the system  

73

5.2 Payment models example  

77

5.3 An example of a path array  

80

5.4 Overall miss rate, ads sent, revenue and revenue per ad as a function of the value of the $r$ parameter of the random strategy  

83

5.5 Round miss rate and messages sent per advertisement throughout the rounds  

84

5.6 The effect of churn on round miss rate and messages sent per advertisement  

85

5.7 Nodes’ revenue as a function of nodes’ availability for all tested payment models  

86
Abstract

Internet video streaming already accounts for over 64% all global IP traffic and is projected to increase in the foreseeable future. Consequently, improving the delivery mechanisms of video streaming is an important challenge with real-world consequences that has seen a plethora of protocols and systems been published and made available by industry and academia over the years. Of these systems, peer-to-peer (P2P) delivery, either between client machines, set-top boxes, middle-boxes, or intermediate (possibly software defined) routers, is compelling as it relieves much of the congestion off the video source or networks controlled by the video provider. Further, if done correctly, distributed delivery spreads the network load in a more balanced and localized manner among participants than centralized solutions.

We first present MOLStream, a modular framework for rapid development and evaluation of P2P live streaming systems. MOLStream allows P2P streaming protocols to be decomposed into basic blocks, each associated with a standard functional specification. By exposing structural commonalities between these components, MOLStream enables specific implementations of these building blocks to be combined in order to devise, refine and evaluate new P2P live streaming protocols.

Next, we introduce the notion of ingredients: a novel abstraction denoting the smallest inter-operable units of code that each express a single design choice. We re-implement our framework, now called StreamAid, to provide tools for designing protocols in terms of ingredients, systematically testing the impact of every design decision in a simulator, and deploying them in a wide-area testbed such as PlanetLab for evaluation. We show how to decompose popular P2P live streaming systems, such as CoolStreaming, BitTorrent Live and others, into ingredients and how StreamAid can help optimize and adapt these protocols.

Finally, we explore how to dynamically join multiple systems to adapt to variable environments. We implement a multi-armed bandits (MAB) algorithm component in our framework that adjusts to changing environments and demonstrate how the performance of an autonomous adaptive approach outperforms each part in isolation.

StreamAid is written in Java and is available as an open-source project at https://github.com/alibov/StreamAid.

We then take a step back and discuss incentive mechanisms. Lately, internet usage is shifting toward mobile devices. A person can connect to the internet or web applications using different mobile devices. Tit-for-tat like incentive mechanisms in P2P networks are incapable of dealing with such usage patterns. A centralized incentive mechanism can keep track of specific users,
but, such incentive mechanisms will not scale well for large P2P networks. We propose a new approach to mobile P2P networks coupled with a new P2P advertisement mechanism. In particular, we explore several advertisement dissemination schemes combined with a few payments models and compare between them. The reported results are encouraging for this direction.
Chapter 1

Introduction

Online video services will be used by nearly two billion Internet users by 2018 according to projections, consuming over 79% of all Internet traffic [Cis14]. While video-on-demand is the major constituent of Internet video, the popularity of live video streams has been growing rapidly, with the number of global IPTV subscribers now exceeding 100 million [Poi14], and live streaming platforms such as Twitch.tv attracting more than 45 million users per month [O’N14]. Unfortunately, the technology for Internet live streaming is still far from reaching the maturity of traditional broadcast media, such as TV cable networks. Massive outages show that online broadcasts from major events such as the FIFA World Cup and the Oscars shake the foundations of current live streaming architectures [Spa14, AS14], where scalability is typically afforded through expensive provisioning of large content-distribution networks (CDNs) [NSS10, DCM10, YLZ+09].

Consequently, improving the delivery mechanisms of video streaming is an important challenge with real-world consequences that has seen a plethora of protocols and systems been published and made available by industry and academia over the years. Of these systems, peer-to-peer (P2P) delivery, either between client machines, set-top boxes, middle-boxes, or intermediate (possibly software defined) routers, is compelling as it relieves much of the congestion off the video source or networks controlled by the video provider. Further, if done correctly, distributed delivery spreads the network load in a more balanced and localized manner among participants than centralized solutions. Live streaming, where all viewers watch the stream at the same time, particularly benefits from such decentralized P2P approaches owing to the rigid timing requirements for delivery and the limited content availability.

Indeed, researchers have long advocated that leveraging peer-to-peer (P2P) networks for delivering live streams can significantly ameliorate scalability concerns by reducing burden on the broadcast source [YLZ+09, TAYZ12, ZLLY05, LGL08b, ZH12a]. The participation of peers helps providers overcome the Goldilocks paradox: provision too few resources (such as CDN nodes) and the system fails to scale – a disaster, provision too many and precious money is wasted on unneeded resources [Sch15]. Involving end-users in the multiplexing of live streams offers alluring bandwidth cost savings for businesses, yet relatively few companies have successfully incorporated the edge in content delivery – CoolStreaming [ZLLY05, LXQ+08],
PPLive [HLL+07] and Akamai NetSession [ZAC+13] are prominent examples despite the first
two having been discontinued due to legal concerns.

Developing a large-scale peer-to-peer (P2P) live-streaming system is a time-consuming,
complex and error-prone endeavor. Such systems typically require a myriad of design decisions,
such as the choice of P2P overlay structure, view maintenance algorithms and failure recovery
mechanisms, each of which requires substantial effort to evaluate [TAYZ12]. The complexity is
further compounded by the wide range of objectives and metrics used to assess a live-streaming
system, including minimizing costs, latency and bandwidth use while maximizing the quality of
experience and playback continuity. In particular, some of these concerns are at odds with one
another.

Part of the difficulty is the lack of convenient frameworks for rapid prototyping, deployment
and evaluation of new algorithms and ideas in real settings. Quick prototyping in a custom
simulator runs counter to the goal of real-world evaluation or deployment, and significant effort
is required to move from simulation to a proper evaluation of a P2P live-streaming service.
Moreover, the effort invested may be spent on parts of the system that are immaterial to proving
the efficacy of an idea, such as debugging video codecs or calibrating third-party overlays.
Evaluation typically involves a reimplementation of the simulation protocol with improved error
handling, and often requires a plethora of metrics to be implemented, measured and assessed on
a distributed testbed such as PlanetLab [CCR+03] or Emulab [WLS+02]. Effects that may have
been visible in a synchronous discrete event-based simulator may fade away when effects of
real testbeds (such as packet loss, churn or asynchrony) enter the picture, invalidating significant
investment of work [WLZ12, TAYZ12].

To facilitate progress in the field of P2P live streaming, a development and evaluation
framework satisfying the following goals is needed.

- **Generality.** The framework should be designed to support a broad family of live streaming
  constructs, such as interfaces for tree-based and mesh-based overlays, support for
  centralized and decentralized bootstrap services, implementation of diverse evaluation
  metrics, and so forth.

- **Modularity.** A P2P live streaming system should be loosely coupled with easily replace-
  able components [ZH12a]. Encapsulation enables isolated parts to be gradually improved,
  such as optimizing the dissemination overlay, without impact on other components being
  a concern.

- **Usability.** Coding and implementation should be facilitated by natural, well-defined
  interfaces. The transition from simulation to large-scale experimental evaluation should
  be seamless.

- **Measurability.** The framework should be capable of evaluating and reporting common
  performance metrics and statistics for P2P live streaming, such as end-to-end latency,
  bandwidth, playback continuity and lag.
While some frameworks, such as PeerSim [MJ09], OverSim [BHK07] and PlanetSim [GPM+05], have been developed to address some of these needs in the context of general P2P systems, they lack evaluation capabilities and functionality specific to live streaming applications. A domain-specific solution is required.

In Chapter 2, we aim to disentangle the complexity of building, improving and evaluating P2P live streaming systems by introducing a novel framework MOLSTREAM that satisfies the aforementioned goals.

By breaking down P2P live streaming algorithms into modules, MOLSTREAM is able to facilitate code reuse and better understanding of P2P live streaming systems. However, each module still hides many different design decisions. In Chapter 3 we set out to distill and single out every design decision that is made when implementing a P2P live streaming protocol. We focus on providing abstractions for understanding the impact of design decisions in live streaming systems. We introduce the concept of ingredients: atomic composable building blocks of code with minimal uniform interfaces that each specifies a partial functionality of a protocol at the level of a single design decision. As a concept, casting live streaming protocols as ingredients has several advantages.

- **Simplicity.** Each ingredient is a simple, well-defined programming task.

- **Modularity.** Developers of new protocols are motivated to make all design choices explicit and to minimize coupling within code as much as possible.

- **Extensibility.** Additional ingredients can be added into the system when they are identified without touching any parts of the code that are not directly involved in the interaction.

- **Optimizability.** Each ingredient can be tuned and subjected to scientific tests, for instance by varying an independent parameter while keeping others constant.

We re-built our MOLSTREAM framework based on the ingredient abstraction, and renamed it to STREAMAID. Aimed to help overcome the stumbling blocks identified above, STREAMAID leverages ingredients to dislodge the complexity of live streaming protocols, facilitate testing, and provides seamless deployment on PlanetLab for large-scale evaluation.

We surveyed seven prominent P2P live streaming systems [ZLLY05, GKM03, MR07, MK04, PPKB07, WXL07, Coh12], identified the fundamental pieces of functionality and design choices made by each of them, and expressed them as ingredients within our system. The unified implementation of all of these systems in a single framework enables apples-to-apples measurements and testing. Further, STREAMAID lets us replace any basic design choice of a given system and evaluate the impact on the attained performance or other metrics simply by changing the corresponding ingredients, which is done through an XML configuration file. We have performed several such experiments and this chapter highlights the insights we gained from them, underscoring the usefulness of the abstraction. We believe the framework can help identify shortcomings in less effective live streaming protocols to enhance their performance, and to characterize precisely what design choices constitute the best P2P streaming protocol for a given environment.
As mentioned before, surveying the vast literature on P2P live streaming, one is faced with a myriad of protocols [SENB09, MM02, VYF06, VF06, PPKB07, ZZSY07, WXL07, ZLLY05, PKT+05, BKK+03, Coh, ZH10, LXLQ+08, MRR+14, MR07, HLR08, ZHL2b, RPI12, ZHL2a, LGL08b, ZCL11, LGLO9, SMG+07, LGLO8a, TAYZ12, MRG07, MR06, ZLC09, DSLMM08, SFG01, SSSY11, DCM10, YLZ+09, Coh12]. Comparing these approaches reveals that many protocols rely on different assumptions about the environment, such as scale and network settings, and thus arrive at different trade-offs for their corresponding systems. These assumptions also change over time [DKH11]: a live streaming protocol that works well at small scale may operate poorly when faced with rapid growth or flash crowds, and vice versa [WRRH14]. Consequently, no one approach is appropriate for all circumstances. The operators looking to adopt P2P live streaming protocols are thus faced with the challenge of identifying what system or protocol to use for their given scenario to get best performance.

Chapter 4 reports on our search for an adaptive P2P live streaming protocol to operate in a wide range of conditions, even as the environmental conditions vary. The adaptive algorithm chooses the best performing P2P live streaming algorithm and switches to it on the fly. We have developed a framework, SMARTSTREAM, that locally learns and switches autonomously to the the best alternative among a large selection of live streaming algorithms. SMARTSTREAM is built on ideas from learning theory, specifically the multi-armed bandits (MAB) algorithms for sequential decision making with incomplete information [ACBFS95, ACBF02, Agr95].

P2P networks are often used these days for content sharing, live video streaming being just one example. Most P2P networks are self-sustained without any central authority to provide resources, thus relying only on their peers. Such resources include storage space, bandwidth, availability and possibly more. When participating in a P2P network, in order to receive a good level of quality of service, a peer has to contribute some of those resources. A P2P system should limit the possibility of freeloading [FC05, FPCS06] - receiving service while not contributing at all, although, some minimal service can be granted to convince users to contribute to get better service [Coh03]. This is where incentive mechanisms step in. Their job is to provide incentives for the peers to contribute resources. Incentives can be non economical - a user can get better service if he contributes more resources [FC05, FPCS06, Coh03], and they can also be economical - a user can get paid to provide resources [TJ08, MMK09, RCS09, SSS10].

An advertising mechanism can satisfy the requirements of the latter. Using an advertising mechanism, a user can donate resources to help spread advertisements (alongside the content) and later the advertiser can share revenues with the users that helped earning them. A P2P network with user profiles can yield even more revenue for the participating parties due to the possibility to target the advertisements for the suitable audience.

Traditionally each peer in the P2P network is a machine capable of doing some computation and is associated with a user of this P2P system. However, nowadays, mobile web gains more popularity. More and more users connect to the web and use web applications from their mobile devices. To reflect that, our model of the P2P network includes user donated machines that are connected by a P2P overlay. In order to interact with the P2P network, users use client devices to directly connect to a peer in the P2P network. In Chapter 5 we set to study such advertisement
mechanisms.
Lastly, we conclude in Chapter 6.
Chapter 2


In this chapter, we aim to disentangle the complexity of building, improving and evaluating P2P live streaming systems by introducing a novel framework MOLStream that satisfies the goals mentioned in the Introduction.

The primary contributions of this chapter are the following.

- We design and implement a modular, general framework MOLStream to facilitate rapid prototyping and evaluation of P2P live-streaming systems.

- We demonstrate how several existing protocols can be modularized with MOLStream and how this modularity accelerates improvements to these protocols.

- We evaluate MOLStream by using it to run simulations and experiments on a mash-up of components from existing protocols under several metrics.

- We demonstrate that the same code can be evaluated with MOLStream on both PeerSim [MJ09], a real deployment over a cluster of machines as well as on PlanetLab [CCR+03] deployment. Moreover, we show that the results of these runs are consistent across these environments.

This chapter is based on the publication [FLV14].

Roadmap. The remainder of the chapter is organized as follows. We discuss various related works in Section 2.1, and explain the basic terminology and assumptions in Section 2.2. We describe the design of MOLStream in Section 2.3 and provide more detailed explanation of specific components we have already implemented in Section 2.4. We then illustrate the use and benefits of MOLStream through several case studies where we reconstruct and calibrate known protocols using the MOLStream framework Section 2.5, and offer concluding thoughts in Section 2.6.
2.1 Related Work

The goals of MOLSTREAM of generality, modularity and usability are shared with a number of frameworks that have been developed for general distributed systems and algorithms. To give specific examples, Weevil [WCW08] is a programmable tool to help automate evaluation of distributed systems with focus on workload generation and experimental execution, as opposed to the development process. The SPLAY project [LRF09] enables developers to specify distributed algorithms in the Lua scripting language and experiment with them directly on PlanetLab [CCR+03], or other testbeds. SPLAY provides libraries to facilitate development including third-party C/C++ libraries such as video transcoding to experiment with adaptive streaming. ProtoPeer [GADK09] is a Java-based toolkit for prototyping and evaluating P2P systems that can transparently transition between an event-driven simulation and a live network deployment. In a similar vein, Kompics [ADH09] is another generic component model coded in Java that allows mash-ups of event-driven modules to be created, but without imposing a hierarchical component structure like ProtoPeer. To the best of our knowledge, none of these projects have been specifically customized, or used to develop or experiment with P2P live streaming protocols.

A number of event-based simulators for P2P systems have been built to ameliorate research in the field [MJ09, BHK07, GPM+05]. Nevertheless, the lack of domain-specific features for live streaming may deter their use by researchers: A review of 287 papers in the P2P literature showed that over 62% of simulation-based papers used custom-made simulators, hindering the repeatability of results [NLB+07]. MOLSTREAM and STREAMAID interact with these simulators internally, such as by allowing experiments on PeerSim [MJ09] and PlanetLab [CCR+03], but the developer is exposed only to a modular interface that pertains specifically to P2P live streaming concerns.

From the literature on ad-hoc P2P networks, MANETKit [RGCH09] is an example of a composable modular framework for developing MANET applications. While MANET applications are also a form of P2P systems, they differ from Internet based P2P in the fact that MANETs must rely on geographical proximity, forwarding, routing, and frequent network disconnections.

To achieve realism and scalability that experiments on academic testbeds, such as PlanetLab and Emulab [WLS+02] cannot achieve, ShadowStream [TAYZ12] allows experimental algorithms to be embedded in production live streaming systems without risking performance failure or playback disruptions. Whereas MOLSTREAM is focused on quick and early development of live streaming ideas, ShadowStream is tailored for last-stage evaluation of modules that are nearly ready for production.

Finally, in the area of group communication, well-known examples of modular frameworks include, e.g., Horus [vRBM96], Ensemble [Hay98], JGroups [jgr], Appia [MPR01], Quick-Silver [OBD08] and Quilt [HVBL10]. Neko [UDS02] is another example that is geared more broadly for consensus protocols and similar replication based systems. These frameworks are designed mostly for state machine replication in clusters and cloud systems and are optimized
for such environments. They lack many P2P specific features and support for live streaming, which is the focus of our work.

2.2 Background and Model

Broadly speaking, our work seeks to accelerate progress in the growing area of P2P live streaming, which we now define more concretely. A P2P live streaming system consists of a set of end user machines that act as peers in the system and who interact with the system through a client application running on each of their machines. Hereafter, we use the term peer to represent both the donated machine and its user interchangeably.

Live streaming content is offered to the network from one of the peers, known as the source for that stream. A peer invokes a join operation to begin viewing a given live stream, and subsequently starts receiving a series of chunks from the stream. The client application at the peer may decide if and when to play these chunks. Chunks can only be played in the order they were generated by the source, but a peer may opt to play only part of the chunks in a non-consecutive manner.

We assume that peers have limited bandwidth capacity, imposed either on the upload link or total link capacity. Capacity constraints limit the aggregate exchange of content that can occur at each time unit between peers.

Latency is defined as the duration of time that passes from the generation of a chunk until the chunk is played at a peer. Streaming systems have different strategies for keeping latency low, with some trying to minimize the average latency over all the peers while others trying to minimize the maximal latency.

Due to capacity constraints, many P2P live streaming systems parallelize and pipeline the delivery of chunks by forming and maintaining an overlay that enables chunks to be transmitted by the source to its overlay neighbors and then ricocheted between neighboring peers in the communication structure. The details of the overlay and forwarding protocol are the primary features that differentiate P2P live streaming systems. For instance, each chunk in an overlay traverses multiple overlay hops before reaching a given peer, so the latency of a system can also be measured in terms of hop distances (average or maximal) from the source of the stream.

Another way to define the bandwidth constraints is to define some cost function for every connection between two peers. The cost function would define the cost of using each connection as a function of the data transfer rate between the two peers. Systems minimize the overall cost of the stream dissemination over the P2P network. Typically, the cost function is a linear function that is equal for all the connections (then, the problem translates to the Steiner Tree Problem [HR92]). This method can be utilized to minimize latencies when the cost function is based on the latency between the two peers.

Another concern that P2P systems should worry about is the communication overhead they impose. The overhead can be expressed either in terms of messages or in terms of bandwidth. The former can be computed as the average of total number of messages received by each node versus the number of chunks this node has received. In the latter case, we take the total number
of bytes transmitted in the system per peer versus the total number of bytes in all the streams’ chunks.

Often, the clients are heterogeneous, i.e., every client can have different requirements for content quality. We can define a utility function that for each client would define the utility for that client for every content quality that client is receiving. Different Multiple Descriptor Coding (MDC) [Goy01] schemes can be used to achieve different qualities for different users.

Some clients may wish to consume the content without interruptions at the cost of higher latency. For example, in Coolstreaming [ZLLY05] peers buffer content for 10 seconds, then play the chunks continuously, skipping content if a chunk is not available. They define the continuity index as the number of chunks played out of the total possible chunks that could be played during the session of a peer.

Real P2P systems must handle churn, characterized by the rate at which new peers enter the system and existing peers drop out of the system. A successful P2P system should be able to continue its service despite churn and minimize the impact of churn on the performance parameters mentioned above.

Finally, for the success of a P2P network, peers need to be cooperative and execute the protocol as specified. We can distinguish between altruistic peers, who execute the protocol as prescribed even if they do not gain anything from doing so, vs. selfish peers who are willing to cooperate only if they benefit from this. In particular, peers that only consume services from the system but do not help others are known as free riders. Incentive mechanisms reward peers for their contribution to the system and greatly limit the ability of free-riders to get service from the system. This way, selfish peers are motivated to participate in the protocol rather than becoming free-riders. Incentives have been studied extensively and are not at the focus of this chapter.

2.3 MOLSTREAM Architecture

The MOLSTREAM framework consists of roughly 10,000 lines of Java code. As seen in Figure 2.1, the system architecture is modular and consists of several generic components that we describe below. Each component has a well-defined, minimal interface and may be instantiated by different implementations. The bindings to specific implementations occur at run-time based on configuration parameters. Multiple instantiations of the same component type may be executed concurrently within the same system, and one may even invoke methods of the other, as we explain later.

Cross-cutting services. As shown in Figure 2.1, MOLSTREAM includes several generic services which permeate all other components. These include Configuration, Timer, and Logging. The Configuration provides access to the systems’ configuration parameters, as we describe later in Subsection 2.4.3. These parameters are made persistent to an XML configuration file. The Timer component allows other components to register a listener method called nextCycle. The listener is invoked periodically at a frequency controlled by a configuration parameter. The Logging service records errors, debug information, and performance data in log files. In particular, it records various performance counters and statistics.
that enable viewing and exploration of many performance metrics through an accompanying REPORTING application (not shown here).

**Component layers.** Next, we give a bottom-up description of the main modules that form the bulk of the streaming system. The lowest layer is the NETWORK component which deals with all networking related issues. The component binds to the necessary interfaces for networking, which could be an external library such as when running the PeerSim simulator, or the standard UNIX socket library when running on a real IP network, as specified in the configuration file. The layer also handles NAT and firewall traversal in Internet-wide deployments using the STUNT protocol [GF07]. The interface to the NETWORK component includes a best-effort sendMessage method. It also includes a listener (upcall) handleMessage method as well as procedures for exploring and influencing various network characteristics such as upload bandwidth.

The FAILUREDETECTOR component relies on the NETWORK component for detecting failures of other peers. It exposes an isUp method that returns true if the peer seems to be up and responding and false otherwise. The method can be invoked by any other component in the system. The accuracy of the failure detector response depends on the actual operational environment [CT96].

Further up is the OVERLAY component, which manipulates a Neighbors object to form and maintain an overlay. The neighbor selection process can depend on a multitude of parameters. Incentive mechanisms that affect the neighbor selection are implemented in this component. Changes in the overlay are triggered by the nextCycle and handleMessage listener methods. OVERLAY exposes a getNeighbors method. As noted in Araneola [MK04] discussion below, an implementation of OVERLAY may invoke methods of another concurrently running implementation of OVERLAY, which removed any redundancy from our Araneola implementation [MK04]. Also, there are two kinds of components that inherit from OVERLAY:
a **TreeOverlay** and **Grouped-Overlay**. The former adds the `getParent` and `getChildren` methods; the latter splits neighbors into multiple groups that can be retrieved using the `getGroup` method.

Next is the **Streaming** component, which implements the actual chunk dissemination protocol. Typical examples include push, pull, and combined push-pull protocols over the overlay’s edges, but other mechanisms can also be used. This component also implements incentive mechanisms that affect the actual chunk exchange between neighbors. Also, **Streaming** may use multiple concurrently running **Overlay** components, for instance push chunks over a tree overlay while pulling missing chunks over a mesh for robustness, as discussed for **mTreeBone** in Subsection 2.4.1 below.

At the highest level is the **ClientApplication** component, which is responsible for generating the stream’s chunks at the source and for playing the stream’s chunks at other peers. This is also where all UI and codec issues are handled (or delegated to other modules [TAYZ12], currently outside the scope of our framework). The main interface between **ClientApplication** and **Streaming** is through obtaining generated chunks from a `ServerVideoStream` object at the source and passing chunks to a `VideoStream` object as other peers. The default implementation of **ClientApplication** includes buffering chunks for a period of time whose length is a configuration parameter as well as a policy whether to wait or skip missing chunks that is also a configuration parameter. An important role of the **Streaming** component is to decide when the playback starts. For the case when a missing chunk is skipped, this decision actually sets the latency of the whole playback. This could also lead to dangerous situations when the playback has started with too short latency and the streaming algorithm is unable to deliver the required chunks in time before the playback deadline (for example, a node was preempted in a tree overlay). To that end, the **Streaming** component can intentionally add delay to the playback, increasing the continuity at the cost of increased latency. If the **ClientApplication** waits for a chunk when it is missing, adding delays is unnecessary. However, one missing chunk could lead to big delays in extreme cases.

### 2.4 Libraries

To facilitate the development of new P2P streaming systems, **MOLStream** provides an extensible library of implementations for several popular protocols that tend to show up as building blocks for other, more complicated, protocols. We briefly discuss the main components.

#### 2.4.1 Overlay modules

The **MOLStream** library contains a number of overlay protocols that can be used as-is or extended as needed. The implementation for each overlay protocol is between 25 to 298 lines of Java code, as shown in Table 2.1.

Each overlay protocol is further subdivided into the following three constructs. The **Bootstrap** logic is responsible for finding an initial set of peers when only a single peer is known.
Table 2.1: Library modules. The lines of code (LOC) of the MOLSTREAM Java implementation for each component, including error handling.

<table>
<thead>
<tr>
<th>Module</th>
<th>Type</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coolstreaming [ZLLY05]</td>
<td>Streaming algorithm</td>
<td>278</td>
</tr>
<tr>
<td>Prime [MR07]</td>
<td>Streaming algorithm</td>
<td>305</td>
</tr>
<tr>
<td>mTreeBone [WXL07]</td>
<td>Streaming algorithm</td>
<td>35</td>
</tr>
<tr>
<td>TreePush</td>
<td>Streaming algorithm</td>
<td>46</td>
</tr>
<tr>
<td>Bootstrap: Random group</td>
<td>Overlay protocol</td>
<td>94</td>
</tr>
<tr>
<td>Bootstrap: Random node</td>
<td>Overlay protocol</td>
<td>73</td>
</tr>
<tr>
<td>General gossip-based overlay</td>
<td>Overlay protocol (gossip-based)</td>
<td>87</td>
</tr>
<tr>
<td>SCAMP [GKM03]</td>
<td>Overlay protocol (gossip-based)</td>
<td>69</td>
</tr>
<tr>
<td>BSCAMP</td>
<td>Overlay protocol (gossip-based)</td>
<td>25</td>
</tr>
<tr>
<td>Prime [MR07] overlay</td>
<td>Overlay protocol (group-based)</td>
<td>57</td>
</tr>
<tr>
<td>TreeBone [WXL07] overlay</td>
<td>Overlay protocol (tree-based)</td>
<td>298</td>
</tr>
<tr>
<td>Coolstreaming [ZLLY05] overlay</td>
<td>Overlay protocol (gossip-based)</td>
<td>182</td>
</tr>
<tr>
<td>Araneola [MK04, She04]</td>
<td>Overlay protocol (gossip-based)</td>
<td>197</td>
</tr>
</tbody>
</table>

The Neighborhood manager determines and handles connections formed between pairs of peers. Finally, the protocol should implement a Failure recovery mechanism to deal with unresponsive neighbors. These are specified in the OVERLAY template in MOLSTREAM.

SCAMP. A number of overlay algorithms have been implemented and tested in MOLSTREAM. The popular DONet/Coolstreaming system [ZLLY05] defines an overlay protocol which relies on an underlying overlay protocol called SCAMP [GKM03] to disseminate membership messages. SCAMP maintains a unidirectional overlay designed to keep the average number of neighbors for each node at $\log(n)(1 + c)$, for a constant $c \geq 0$, where $n$ is the size of the network. In the MOLSTREAM OVERLAY template, our SCAMP implementation bootstraps each new peer by sending a random seed peer from the bootstrap node. Whenever a connection request is received, the peer forwards it to its neighbors along with $c$ additional forward copies of the connection request. Upon receiving a forwarded connection request, the peer adds the new connection with some probability. To recover from failures, SCAMP peers that stop receiving messages restart the entire protocol using the bootstrap node or one of their neighbors. We have also added a bidirectional version of SCAMP called BSCAMP, where all neighboring connections are mutual.

Coolstreaming. For the overlay structure itself, Coolstreaming defines a protocol that is designed to keep a stable amount of neighbors defined by a custom system parameter $M$. As in SCAMP, a new peer begins its tenure by receiving a seed peer (called a “deputy” [ZLLY05]) from the bootstrap node. The new peer sends a request to the seed (deputy) peer for additional nodes. The peers then use SCAMP to disseminate membership messages. To manage the neighborhood, peers store a cache of known peers from the information received from the membership messages, and bias the cache towards storing those nodes with whom the peer has exchanged a large number of chunks of the stream. When the number of neighbors drops below
$M$, a peer contacts a random peer in its cache with a connection request.

**Aranoeola membership.** MOLStream also implements a scalable randomized membership protocol similar to [She04] used by Araneola [MK04]. Araneola’s membership service begins by contacting a random peer group received from the bootstrap node. Members of the service gossip once in a while with their known neighbors, one random peer per round. Upon receiving new neighbors, the membership service adds them all as neighbors and then discards random neighbors if it exceeds a maximal neighborhood size. The maximal neighbors size is a parameter of the algorithm. When a peer has no more live neighbors left or stops receiving messages, it restarts the protocol using the bootstrap node.

The two main routines behind MOLStream’s implementation of Araneola’s membership service are shown in Listing 2.1, except failure handling code which was removed for clarity. The figure demonstrates MOLStream’s interfaces in practice.

**Aranoeola overlay.** Araneola uses a separate overlay protocol for tracking its membership view [MK04]. Based on the Araneola membership service above, Araneola maintains an overlay that approximates a random regular graph. To bootstrap the Araneola overlay, nodes piggyback on the membership protocol by connecting to nodes in the current view. Nodes continually make connections and disconnect while striving to ensure that each node has exactly $L$ (a configurable parameter) or $L + 1$ neighbors. Each node is aware of the degree of each of its neighbors through a periodic exchange of information. When a peer is unable to accept a new connection because its degree is already $L + 1$, it responds to the connect request with a NACK along with a list of its least loaded neighbor nodes as a hint to the connecting node. To prevent disconnections in the overlay, Araneola nodes actively connect to other peers when their degree drops below $L$.

**TreeBone.** The TreeBone overlay is used by mTreebone [WXL07]. Initially, nodes choose a random node and become its children. A node accepts another node request only if it has enough upload bandwidth to maintain a streaming connection at the desired rate. Nodes with high uptime are viewed as stable and will gradually start joining other stable nodes near to the root of the tree. Stable nodes also perform transformations that decrease the maximal or average depth of the tree. Should the parent of a node fail, the peer will look for a new parent.

**Prime Overlay.** Finally, MOLStream also supports the overlay protocol used by the Prime system [MR07]. In Prime, all neighbor nodes are defined as either parent nodes or child nodes. The service is bootstrapped by requesting a random group of peers from the bootstrap node and sending these peers a connection request. A node will send a connection request only if it has enough download bandwidth, and nodes will accept a connection request only if they have enough upload bandwidth. When all the parents of a peer fail, the peer restarts the algorithm. Prime Overlay implements the groupedOverlay interface providing a parents group and children group.

### 2.4.2 Streaming Protocols

Each of the topologies listed so far can be spliced with a number of streaming protocols that have been implemented. We will list the main ones here.
Coolstreaming. The live streaming mechanism of Coolstreaming [ZLLY05] utilizes the Coolstreaming overlay protocol mentioned above. In this service, nodes periodically exchange data availability information with their neighbors, retrieve unavailable data from one or more neighbors, or supply available data to their neighbors. Each node continuously exchanges availability bitmaps of its segment with the neighbors, and then schedules which segment should then be fetched from what neighbor. The scheduling heuristic first calculates the number of potential suppliers for each segment. The algorithm then determines the supplier of each segment by starting with those that have only one potential supplier. When multiple potential suppliers could be chosen, Coolstreaming selects the one with the highest bandwidth and most ample time before the playback deadline. Even though the implementation of Coolstreaming in [ZLLY05] relies on the Coolstreaming overlay, notice that any other overlay protocol could have been used as well. We will explore this further, along with the modularity of modern streaming services, in a case study in Section 2.5.

mTreebone. We treat mTreebone [WXL07] as a general push-pull protocol that uses a tree overlay protocol for disseminating messages together with a pull protocol that has a limited exchange window if the tree-push protocol failed to deliver chunks nearing their deadline. To implement mTreebone as in the original paper [WXL07], we use TreePush with the TreeBone overlay for regular message dissemination and then use Coolstreaming with its Coolstreaming overlay as the pull protocol. Note that any combination of a tree and other overlay component could be used instead and may produce better results for different scenarios.

TreePush. As a baseline, we implemented a simple streaming protocol that uses a tree overlay called TreePush. In TreePush, the source node waits for chunks to become available and then sends chunks to each child. Every node that receives content from its parent node immediately forwards the content to its children.

Prime. PRIME [MR07] groups peers into levels based on their shortest distance from the source. Chunk dissemination consists of 2 phases: a “diffusion phase” in which all participating peers should receive a data unit (a single description when Multi Description Coding is used) of the chunk as fast as possible and a “swarming phase” in which peers exchange their data units with each other until receiving their desired quality of the chunk. Prime requires a grouped overlay that has a parents group and a children group (such as the Prime Overlay component).

2.4.3 System Parameters

Having described the main libraries provided by MOLSTREAM, we now elaborate on some of the system configuration parameters.

Playback settings

The handling of missed chunks is an important design decision for live streaming systems. The system can wait for the missing chunk to arrive, possibly for some time, or skip it and proceed to the next available chunk. The decision affects latency and continuity and thus the viewing experience for end users.
Another configuration option is the buffering time which is exposed as two parameters in MOLSSTREAM. The first is the time that passes after the system starts and before the playback is started, called serverStartup. The second is the duration which peer wait before starting playback, called startupBuffering. A larger serverStartup time gives additional opportunity for the overlay to organize and stabilize before the actual playback starts. startupBuffering increases the latency of stream, but can improve the continuity of the playback.

Network interface

The framework choice is orthogonal to the actual implementation of the framework components. The Network component has a NetworkModule abstract class that interfaces with the underlying network. NetworkModule defines these methods.

- **NodeAddress getAddress()** Returns a container for the address of the node. NodeAddress encapsulates the unique identity and address of a node that can later be used to send messages measure latency etc.

- **boolean send(Message msg)** Sends a message. The destination NodeAddress is already in the message.

- **long getUploadBandwidth()** Returns the upload bandwidth of the node in bits per second.

- **long getEstimatedLatency(NodeAddress key)** Returns the estimated latency to a specific node.

- **NodeAddress getSourceNode()** Returns the address of the source node. Used by the overlay for bootstrap.

- **SendInfo** The amount of data sent and received as well as the amount of useful data (content) that is sent and received from each neighbor is stored for use by different algorithms. The data can be used for peer selection.

More classes can be developed that extend the NetworkModule abstract class.

Deployment

In addition to being designed for quick prototyping, MOLSSTREAM permits protocols to be deployed easily in simulated environments and on real test-beds. Currently, MOLSSTREAM supports both running over the PeerSim simulator [MJ09] as well as running over a real IP network using UDP. The framework is extensible and support for other simulation platforms can be added. Each simulation is fully reproducible which simplifies the debugging of new components compared to non-deterministic experiments over real networks. Experiments, on the other hand, produce more realistic results and trigger corner cases that are not always reachable by simulation. To enhance the realism of the simulations, we support node failures, constraints on upload bandwidth, drop rates and latencies.
Bitrate and upload bandwidth

The bitrate of the stream, upload bandwidth of the source and the upload bandwidth distribution can be configured. The effect of these parameters is twofold. First, some algorithms use the bitrate of the stream together with the upload bandwidth to define the maximum amount of neighbors or data to be sent. Second, algorithms that do not account for the bitrate or upload bandwidth may attempt to send too much data which may hamper latency.

Churn model

Churn experiments are critical to the analysis of any P2P protocol. We have implemented several churn models in MOLS\textsuperscript{REAM} that can be employed.

- **none** No churn, the number of peers remains constant throughout the run.
- **sessionLengthInterArrival** Allows distributions for the session length of nodes and the inter-arrival time for any two new nodes to be specified. Whenever a session ends, a node fails. The addition of new nodes is independent of the failures.
- **sessionLengthAddOnFailure** A distribution is specified for the session length of a node. When a session ends, the node fails and a new one joins at the same instant.
- **sessionLengthOffLength** In this model, a distribution for the session length of a node and a distribution of cool-off times. After a session ends, the node fails and a new one joins after the cool-off period.
- **availabilityFile** To support trace-driven simulation, this model parses a file describing the arrival and departure times of nodes. The nodes in the system behave as described in the file.
- **eventBased** Under this model, join and leave events can be specified. The number of nodes for each event and delays between events are parameters. This model can simulate flash crowds, i.e., large amount of nodes joining or leaving the system.

2.4.4 Supported Performance Metrics

MOLS\textsuperscript{REAM} logs a vast array of performance counters and statistics during each run. Some of the counters are time based, e.g., \textit{startup delay} - the average time it takes for a new peer to start receiving content, \textit{average latency} - the average time from the chunk generation until the chunk playback and \textit{lag} – the difference in latency between the chunks played last and first. Other counters measure the data consumption such as total amount of data sent by a protocol, or amount of data sent by the nodes during each second of the playback. Last, there are counters that are chunk related such as the number of chunks played each second and the number of peers that have played each chunk. Some statistics are available based on the distance of the node from the source, for example the average latency per hop distance.
2.5 Case Studies

To demonstrate the effectiveness of MOLSTREAM, we investigate the effort taken to implement, test and deploy a popular P2P live streaming service in the literature: Coolstreaming [ZLLY05]. The first goal is to replicate the implementation and results from the paper by Zhang et al. and then show how MOLSTREAM facilitates modular improvements.

Assumptions

In the following, we show usage examples of MOLSTREAM for different cases. If not stated otherwise, we have used the PeerSim simulator. If a chunk is unavailable when the playback time is reached, the chunk is skipped. The server startup time is set to 10 seconds. When using PeerSim, we run each experiment five times with a different random seed, and average the results.

Coolstreaming

We have implemented Coolstreaming inside MOLSTREAM in only 278 lines of Java code plus 184 LOC for the modified SCAMP overlay used by Coolstreaming. We experimented the implementation with a similar settings to the ones reported in [ZLLY05] and obtained similar results as in their work. For our deployment, we used the PeerSim simulator with 10 to 200 nodes, with playback of 10 minutes of 500Kbps, 10 second startup buffer and no churn. In Figure 2.3(a), we can see the continuity index for different network sizes and different values of \( M \) — the overlay parameter of Coolstreaming. Figure 2.3(b) shows the overhead incurred by the different \( M \) settings. In the graph, the playback continuity index improves with higher \( M \) values. However, after \( M = 4 \), the improvement becomes marginal. The graphs strongly resemble the graphs in the Coolstreaming paper [ZLLY05] as can be seen in Figure 2.3, suggesting that MOLSTREAM enabled the prototyping of a relatively complicated protocol in less than 500 LOC of Java in addition to facilitating experimentation.

Coolstreaming with different overlays

We have tested the Coolstreaming streaming algorithm with different overlays other than the original Coolstreaming overlay. We have used a startup buffer of 5 seconds and a network of 200 nodes with playback time of 12 minutes and no churn. Each node’s upload bandwidth is set to 5.56 Mbps\(^1\), while, the server’s upload bandwidth is set to 16.68 Mbps (3 times the average). We have chosen the parameters of the different overlays so that the average node degree would be as similar as possible. Figure 2.2 shows the average degree as a function of the uptime of a node. Recall that Coolstreaming overlay periodically drops the lowest scoring partner. We have set the parameter of Coolstreaming overlay to drop the lowest scoring neighbor every 30 seconds. As can be seen in Figure 2.2, indeed there is a drop in the Coolstreaming Overlay degree every 30 seconds. Table 2.2 summarizes the results. In this setting, Araneola and BSCAMP (Bidirectional

\(^1\)Average upload bandwidth taken from http://www.netindex.com/upload/.

20
Table 2.2: Continuity Index, Playback Latency and Control overhead measured for different overlays with the Coolstreaming streaming component

<table>
<thead>
<tr>
<th>Overlay</th>
<th>Continuity Index</th>
<th>Playback Latency (ms)</th>
<th>Control Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Araneola ((L = 3))</td>
<td>0.997</td>
<td>9893</td>
<td>0.008</td>
</tr>
<tr>
<td>BSCAMP</td>
<td>0.996</td>
<td>9771</td>
<td>0.008</td>
</tr>
<tr>
<td>Coolstreaming ((M = 4))</td>
<td>0.97</td>
<td>9239</td>
<td>0.01</td>
</tr>
<tr>
<td>Prime</td>
<td>0.925</td>
<td>8624</td>
<td>0.006</td>
</tr>
</tbody>
</table>

SCAMP) obtain a high continuity index. Nevertheless, their playback latency is also high. On the other end of the spectrum, the Prime overlay has the lowest continuity index, but obtains the lowest latency and control overhead. Prime does not make any action after the initial bootstrap. Prime manages to get such low latency and overhead only because there are no failures in the setting. It is important to note that Prime is a hierarchical group-based overlay, whereas Coolstreaming treats all neighbors of Prime equally and makes no distinction based on group membership.

Figure 2.2: Average degree of Coolstreaming with different overlays

Araneola parameter tuning

To demonstrate how MOLSTREAM can be used to tune the parameters of the different algorithms, we tested Coolstreaming with the Araneola overlay in a network of 512 nodes simulated by PeerSim. The simulation length is 10 minutes. We use the sessionLengthAddOnFailure churn model: the session length of the nodes is log-normally distributed with mean \(= 4.29\) and variance \(= 1.28\) (following Magherei et al. [MRG07]). When a node fails, a new one is added. Each node had an upload bandwidth of 5.56 Mbps, while the simulated stream bitrate was 450 Kbps. In these settings, each peer can maintain roughly 12 neighbors to which it could send one chunk of the stream each second \((5560/450 \simeq 12)\). In Figure 2.5(c), we see that the minimal average latency for the Araneola algorithm is reached when the \(L\) parameter is exactly 12. This is because Araneola tries to achieve exactly \(L\) or \(L + 1\) neighbors for every node. The same
Figure 2.3: Case study of Coolstreaming. The continuity index and the overhead of a MOLSTREAM Coolstreaming implementation protocol in the MOLSTREAM compared to the original results of the Coolstreaming protocol.

behavior is evident in Figure 2.5 which portrays the average degree over uptime. As can be seen in Figure 2.5(b), Araneola takes roughly 5 seconds to amass neighbors and reaches $L$ in roughly 25 seconds. The degree is maintained throughout the whole run as evidenced in Figure 2.5(a).

MOLSTREAM network framework

We have used a cluster of 31 machines to test the network framework of MOLSTREAM. To that end, we have ran the original Coolstreaming streaming algorithm with the Coolstreaming overlay with the parameter $M = 4$. We ran a stream of 500Kbps for 10 minutes with no churn. We compare these results to an identical settings using the PeerSim simulator and PlanetLab.

As shown in Figure 2.6, the results are comparable. The PeerSim simulation has a slightly higher latency since the simulated latencies are probably higher than the actual latencies in the cluster. In PlanetLab the latencies are naturally higher.

We have also tested on PlanetLab with up to 180 nodes. Figure 2.4 shows that the results scale up with a slight improvement of the continuity index as the system grows.
Figure 2.4: Continuity index and one SD for various sizes of networks of PlanetLab nodes

Figure 2.5: Araneola overlay. (a) and (b) Average node degree as a function of uptime of a node (c) Average latency of different L settings.

mTreeBone transformations

We have implemented the mTreebone algorithm [WXL07]. In this experiment, we use the standard Coolstreaming streaming protocol (with $M = 4$) as the fallback streaming protocol of mTreebone. Also, we simulate churn using the $sessionLengthOffLength$ churn model. In which, the session length is exponentially distributed with mean of 50 seconds and then the node waits
We have tested the effect of the stable coefficient on the performance of the protocol. The stable coefficient defines when a node becomes stable, and thus able to perform transformations, with a coefficient of zero implying that all nodes are always stable and a coefficient of one implying that no node is ever stable (except the source node). We found that the continuity index is roughly the same for all the settings. In contrast, as we can see from Figure 2.7, higher coefficient means less overhead, and lower latency. However, starting from 0.2, the differences are marginal.

2.6 Discussion and Conclusion

We have described MOLSTREAM, an open source modular framework for rapid prototyping, testing and performance tuning of P2P live streaming protocols. As demonstrated, MOL-
STREAM facilitates comparing between different protocols, and in particular, between the various aspects of the protocols such as overlay maintenance and streaming. In particular, MOLSTREAM includes a built-in logging and reporting mechanism that records and generates statistics on common performance metrics we have found in various research papers on the subject, including, for instance, latency, throughput, communication overhead, continuity index and lag. Engineers can use our framework to establish which solution works best for their settings, while developers of new protocols and researchers benefit from MOLSTREAM as it enables them to focus only on the parts they wish to study and improve.

The ability to run the same code on both a simulator (PeerSim) and in a real deployment facilitates the transition from a proof-of-concept simulation to a real-world experiment, and from communication over emulated environments to physical networks. PeerSim is a mature and realistic simulator that we use by default; changing to a different or home-brewed simulator within MOLSTREAM is simple to do as simulation code is encapsulated behind a simple interface in the NETWORK component.

We have shown that various known protocols have been implemented and tested in MOLSTREAM. Relatively little effort is required: the implementations for each of the various components of MOLSTREAM, for example, comprise only between 35-305 lines of Java code. As mentioned earlier, MOLSTREAM is available at https://github.com/alibov/StreamAid.
Listing 2.1 Sample code. General gossiping protocol implemented in MOLSTREAM. The extract shows the main source code to implement the Araneola membership overlay [MK04], except for failure handling cases.

```java
@Override
public void handleMessage(final Message message) {
    if (message instanceof PartialMembershipViewMessage) {
        final List<NodeSpecificImpl> neighborsList =
            ((PartialMembershipViewMessage) message).neighborList;
        // add the neighbors received in the message
        for (final NodeSpecificImpl newNeighbor : neighborsList) {
            addNeighbor(newNeighbor);
            addNeighbor(message.sourceId);
            // prune random neighbors if needed
            while (getNeighbors().size() > groupSize) {
                final List<NodeSpecificImpl> nList =
                    new ArrayList<NodeSpecificImpl>(getNeighbors());
                removeNeighbor(nList.get(random.nextInt(nList.size())));
            }
        }
    }
    @Override
    public void nextCycle() {
        super.nextCycle();
        if (currentDelay-- == 0) {
            currentDelay = gossipDelay;
            final Set<NodeSpecificImpl> neighbors = getNeighbors();
            final List<NodeSpecificImpl> neighborsList =
                new ArrayList<NodeSpecificImpl>(neighbors);
            for (final NodeSpecificImpl neighbor : neighbors) {
                neighborsList.remove(neighbor);
                Collections.shuffle(neighborsList, random);
                final List<NodeSpecificImpl> sublist =
                    neighborsList.subList(0,
                        Math.min(amountToSend, neighborsList.size()));
                node.send(new PartialMembershipViewMessage(getMessageTag(),
                    node.getImpl(), neighbor, sublist));
                neighborsList.add(neighbor);
            }
        }
    }
}
```
Chapter 3

Distilling the Ingredients of P2P Live Streaming Systems

The primary contributions of this chapter are the following. First, we present the ingredient abstraction to disentangle the often complex components of live streaming systems into fundamental, atomic building blocks. This granular abstraction gives designers and implementers a rigorous methodology for creating, presenting, optimizing and evaluating individual design decisions of their protocols.

Second, we implement the STREAMAid system which supports the ingredient abstraction in two ways. First, STREAMAid provides a Java interface by which developers can program new ingredients for live streaming protocols, and an XML configuration file that allows ingredients to be blended. Further, we provide an extensive framework to rigorously test and calibrate the resulting protocols, to evaluate them through local simulation and to deploy them on real distributed testbeds such as PlanetLab [CCR+03]. In contrast to prior work, the framework allows experimentation on individual ingredients while keeping all others constant, providing a scientific way for testing and optimizing a live streaming system.

Finally, we show how several popular live streaming protocols can be ported to use the ingredient abstraction, including BitTorrent Live [RKH14], CoolStreaming [ZLLY05] and others. We found that many existing protocols share ingredients, with several published protocols having identical functionality except for a single ingredient. We report on experiments where we evaluate individual ingredients of these protocols, and systems that have been composed of several existing ingredients. Our results show how STREAMAid helps expose and balance the impact of different trade-offs in live streaming protocols.

The code for STREAMAid and the protocol implementations are freely available online from https://github.com/alibov/StreamAid under a permissive BSD license.

This chapter is based on the publication [FLV15].

Roadmap. The rest of this chapter is organized as follows: We describe our concept of ingredients in Section 3.1. We continue with the breakdown of P2P live streaming protocols to their basic design decisions in Section 3.2. We evaluate several design decisions in Section 3.3, and propose optional debug ingredients in Section 3.4. We survey related work in Section 3.5.
Finally, we discuss our conclusions and future work in Section 3.6.

3.1 Modules and Ingredients

Distributed systems are commonly engineered as a stack of micro-protocol layers that each serve a well-defined function [jgr, MPR01, UDS02, vRBM96]. In such systems, a message sent by a specific layer \( L \) goes through every underlying layer at the sender. Upon reception on the receiving node, the message passes through the layers in reverse before reaching the corresponding layer \( L \). For example, as depicted in Figure 3.1, an encryption layer may encrypt all outgoing messages of a node after the message is processed by other micro-layers, and decrypt all received messages of a receiving node before the message is processed by other micro-layers.

The micro-layer design is ideal when messages are relevant for all underlying layers, since messages will always flow through them. P2P live streaming protocols fit this pattern. The design is also effective when building general-purpose communication middleware for accommodating a large variety of communication patterns and systems. When these conditions fail to hold, however, the micro-layer approach imposes overhead and can make implementations awkward.

3.1.1 Ingredients of P2P Live Streaming

Our work is focused on finding thin yet informative micro-layers for P2P live streaming.

Listing 3.1 CoolStreaming [ZLLY05] built using ingredients.

```
<streamingAlgorithm algorithm="PullAlgorithm" size="200">
  <overlayAlgorithm H="200" M="4" c="1" amountToSend="6"
  exploreRound="30" gossipTimeout="6"
  algorithm="CoolStreamingOverlay">
    <ingredient name="NeighborChunkAvailabilityIngredient"
      operationMode="updateEveryRound"/>
    </overlayAlgorithm>
  </streamingAlgorithm>
```

Our work is focused on finding thin yet informative micro-layers for P2P live streaming.

Figure 3.1: Micro-Layers: a model where messages pass through a stack of thin layers for transmission and reception on the destination.
systems, which we call ingredients. The idea is for the aggregate functionality of a P2P live streaming system to be divided into both coarse-grained modules that are then further subdivided into these micro-layers ingredients. A module can be viewed as a composite set of ingredients along with a so-called core, as illustrated in Figure 3.2 and explained below. Within a single network node, modules may interact with one another through method invocation through a fixed interface. Different implementations of modules can also be mixed and matched, as described in Chapter 2.

This chapter investigates how modules can be broken down into a single core and multiple ingredients. The core contains the “essential” uniquely identifying functionality of a given protocol for implementing that module, whereas ingredients include reusable pieces of functionality that could be used by several protocols, even to enhance the protocol. For maximal modularity, the core should intuitively be as small as possible, ideally even empty. Yet, sometimes certain pieces of functionality uniquely characterized a protocol and we found no reasonable way to augment them or include in other protocols. In those cases, we kept the functionality as part of the core, as this helped produce more simple and readable code. As discussed below, in modules for some P2P live streaming protocols we managed to obtain an almost empty core and organize almost all functionality into ingredients, while in others we were less successful.

Our approach resembles aspects, as per aspect-oriented programming [KLM+97], which handle cross-cutting concerns that affect many locations in a system’s code, for instance logging. We remark that our approach differs in that an ingredient is encapsulated logic which is only instantiated for its specific purpose and which is also tethered to a specific module.

Architecture. STREAMAID comprises four modules: a player module, an overlay module, a streaming module and a network module (Figure 3.3). The network module handles all communication aspects, providing a simple-to-use API with a SENDMESSAGE method and a RECEIVEMESSAGE upcall. The overlay module uses the network module to construct the actual P2P overlay. There may be multiple initiations of this module running in parallel. The streaming module queries an overlay module for neighboring connections and initiates video chunk exchange with the neighbors according to its protocol. Finally, the player module handles all buffering, encoding, decoding and playback of the video content. This approach allows the overlay modules for one known system to be mixed and matched with streaming modules corresponding to other systems.

For clarity of presentation, Figure 3.3 and Listing 3.1 only exhibit a partial view of all possi-

![Figure 3.2: STREAMAID Module: A module is composed of a set of ingredients in STREAMAID.](image-url)
ble ingredients. Each of the ingredients encapsulates a specific design decision in implementing the module’s functionality. We explore the details in Section 3.2 below. Here, we focus on discussing conceptual aspects of ingredients and their communication and interaction model.

**Communication model.** Messages generated by a certain module on a given peer can only be received by the same module on the other peer\(^1\). Such a message can either be addressed to a specific ingredient on the receiving peer, or to all ingredients of the corresponding module on that peer as well as the core of that module. Further, a message can only be addressed to the ingredients of the same module in which they were generated. When such a message is addressed to all ingredients of a given module, the message is passed to these ingredients one after the other in the order these ingredients appear in the configuration file. All sent and received messages pass through the network module, which only contains ingredients relevant to all messages in the system.

At any event, passing messages across modules is not allowed. The only way to pass information between modules, and only between modules of the same peer, is through the defined interface for such invocations, as depicted in Figure 3.3.

In terms of usability, the actual set of ingredients to be invoked is specified in a configuration file that is parsed at runtime. The appropriate ingredients listed in the configuration file are instantiated when the framework is started.

**Example.** Consider the example ingredients as shown in Figure 3.3 and its corresponding configuration shown in Listing 3.1. Here, the overlay module sends and receives messages from the corresponding overlay module on the remote peer. The “chunk availability” ingredient sends and receives messages from its remote counterpart “chunk availability” ingredient. The “request sending” ingredient sends a message to the corresponding streaming module, so that all associated ingredients of that module can receive the chunk request message. All messages in

\(^1\)Except for the network module, which is involved in passing every message sent between peers.
Figure 3.3 pass through the encryption ingredient embedded inside the network module.

The STREAMAID architecture and communication model maintains a clear separation between modules, thus ensuring that modules do not interfere with the tasks of one another. On the other hand, ingredients contained in the same module require a more flexible communication model – in particular an ingredient may not know at development time which ingredients will be instantiated alongside it during runtime to handle messaging.

### 3.1.2 Discussion

Our approach has several benefits: independent modules allow for encapsulation and a loosely coupled protocol structure, while the intra-module ingredients allow messages to be handled according to a chain-of-responsibility and enable seamless addition of fine-grained functionality. Further, ingredients in the network module mimic the micro-layer architecture, allowing us to define system-wide ingredients that affect all sent and received messages. For example, the operation of a pull-based streaming module can be partitioned into several ingredients, as elaborated further below. One such ingredient can handle chunk requests. Different nodes may implement different ingredients for handling chunk requests. Thus, the chunk request message would be sent to the streaming module to be disseminated to all ingredients. Further, an optional reputation ingredient could be added to the streaming module that would track chunk requests to track some statistics, all without modifying any other part of the protocol. This example is illustrated in Figure 3.3.

Should the protocol designer decide to experiment with a different implementation for a specific design decision, they would only need to implement that specific ingredient and replace the previous implementation with the new one. An ingredient can easily be changed via a configuration file setting, making testing different design decisions a simple and fast process. For example, in Chapter 2, the CoolStreaming [ZLLY05] algorithm was decomposed to a streaming module and an overlay module. Here, we have further decomposed the streaming module to separate ingredients so that the CoolStreaming protocol could be defined solely using ingredients. The results are shown in Listing 3.1, which depicts the portion of the STREAMAID configuration file that defines the P2P live streaming algorithm to be used.

### 3.1.3 Ingredients in STREAMAID

STREAMAID is an open-source modular framework, developed for rapid implementation and evaluation of P2P live streaming protocols. STREAMAID supports running experiments using the PeerSim [MJ09] P2P simulator as well as deployment on a real IP network, for instance through PlanetLab, such that the exact same code can be executed in both environments.

Under the STREAMAID model, the support for ingredients enables fine grained control of the protocols produced. An ingredient in STREAMAID is a class that implements two basic methods:

- The `NEXTCYCLE` method is called periodically and can be used to do routine checks and to invoke actions such as sending messages.
The `HandleMessage` method handles messages received by this ingredient, or the underlying module of the ingredient. When sending messages, an ingredient may choose to send the message to the ingredient layer on another node or to the entire corresponding module including all the ingredients associated with the module.

This simple interface proved enough to accommodate all the functionality needed to express a plethora of protocols; in particular every ingredient mentioned in Section 3.2.

### 3.2 Expressing Design Decisions

We surveyed a large number of P2P live streaming systems and identified how their behavior could be encapsulated with the ingredient abstraction. By examining the range of systems in detail, we were able to identify several common design decisions involved in building these systems. We discuss several ingredients corresponding to these choices, and omit a number of others due to space constraints.

Recall that we view P2P live streaming system as comprising four major modules: network interface, overlay maintenance, stream dissemination, and player issues.

#### 3.2.1 Player Module

The player module is a consumer that displays the chunks received by the streaming module and as such does not send out any messages. However, as shown in Section 3.3, there are several important decisions to be made which affect the overall performance of the streaming protocol.

**Player Initialization Time Ingredient**  The streaming module chooses when to initialize the player module. Upon initialization, the player module waits (buffers incoming chunks) for a predefined amount of time (a parameter) and then begins playback. This buffer time is an important design decision by itself, but the point in time when the player module is initialized also impacts overall performance. One option is to initialize the player module on startup; another is to wait for a first bitmap or chunk to be received.

**Player Initialization Position Ingredient**  When the player module is initialized, the streaming module also sets the chunk from which the playback starts. If the player is initialized when a chunk is received, the playback can start from that chunk. However, if the player module is initialized when a bitmap is received, there are several possibilities for the playback starting point. This is not a trivial problem [LXQ+08] since playback should start sometime between the first available chunk and the last one.

**Skip Chunks Ingredient**  After the player was initialized, and the playback has started, the player module can reach a state where the next chunk to be played is missing. The decision of what to do when it happens is usually treated as a binary one - either skip the missing chunk or wait for it. In fact, BitTorrent Live proposed to combine the two by waiting for some time and
then skipping. However, throughout the run of a P2P streaming protocol, the decision whether to wait or skip a missing chunk is circumstantial. The algorithm ought to wait while the buffer window is empty, but if only the next or a few chunks are missing they may be skipped.

**Adaptive Playout Ingredient** Several articles mention the possibility to use *Adaptive Playout*: increase the playback continuity by slightly changing the playback speed while fixing the sound pitch so that the change would be unnoticed by users [SFG01]. Streaming algorithms can slow down the playback to increase the buffer window, increasing resilience and continuity at the cost of increased latency. If the system is stable enough and the buffer window is too large, a faster playback can be used to decrease latency and window size. Adaptive Playout can also be used to decrease startup delay: a streaming protocol can start with a small buffer window, minimizing the startup buffer time, and increase the window by slower playback to get to the desired window size. Listing 3.2 presents a simple ingredient that uses Adaptive Playout for reaching certain latency characteristics.

### 3.2.2 Streaming Module

In most streaming algorithms, chunks are only exchanged with neighbors, which by definition relies on an underlying overlay module. Pull-based streaming modules can normally work with any overlay, whereas push-based streaming modules usually work with tree-based overlays.

**Push-based Streaming**

Push-based streaming is simple: whenever a chunk is received, forward it to all child nodes. The overlay module is responsible for distinguishing child nodes.
Table 3.1: Overlay algorithms classification

<table>
<thead>
<tr>
<th>Overlay</th>
<th>Type</th>
<th>Known List Sources</th>
<th>Neighbor List Dependencies</th>
<th>Request reply</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCAMP [GKM03]</td>
<td>Non-symmetric</td>
<td>Tracker, forward messages</td>
<td>number of neighbors</td>
<td>N/A</td>
</tr>
<tr>
<td>Coolstreaming [ZLLY05]</td>
<td>Symmetric</td>
<td>SCAMP, gossip</td>
<td>number of neighbors, exchange history</td>
<td>number of neighbors</td>
</tr>
<tr>
<td>Araneola [MK04]</td>
<td>Symmetric</td>
<td>Tracker, gossip, forward messages</td>
<td>number of neighbors</td>
<td>number of neighbors, number of neighbors of requesting node</td>
</tr>
<tr>
<td>Pulse [PPKB07]</td>
<td>Non-symmetric</td>
<td>SCAMP</td>
<td>number of neighbors, recent exchange history, average latency, average latency of potential neighbor</td>
<td>N/A</td>
</tr>
<tr>
<td>Prime [MR07]</td>
<td>Symmetric</td>
<td>Tracker</td>
<td>number of parent neighbors, download bandwidth, stream bitrate</td>
<td>number of child neighbors, upload bandwidth, stream bitrate</td>
</tr>
<tr>
<td>mTreeBone [WXL07]</td>
<td>Symmetric tree based</td>
<td>Tracker, tree ancestors</td>
<td>Existence of parent neighbor, upload bandwidth of potential neighbor, distance from source of potential neighbor, stream bitrate</td>
<td>number of child neighbors, upload bandwidth, stream bitrate</td>
</tr>
<tr>
<td>BitTorrent Live [Coh12]</td>
<td>Symmetric multi-tree based</td>
<td>Tracker, gossip</td>
<td>number of parent neighbors, number of children, number of children of potential neighbor</td>
<td>Always positive</td>
</tr>
</tbody>
</table>

The underlying overlay masks neighbor failures, so the only remaining concern for a streaming protocol is to recover the chunks that were missed while the parent node was being replaced. One option is to recover nothing, which is legitimate if another protocol running in parallel will handle chunk recovery, or if missing chunks are skipped during playback. Another option is to request the new parent node to continue sending chunks from where the old parent left off while also catching up if bandwidth allows.

Multi-tree streaming operates in a similar fashion, except that the overlay maintains $t$ spanning trees, each of which spans all peers, where $t$ is a parameter. For each chunk received by the streaming module, it queries the overlay module for its children in one of the trees and disseminates the chunk among these child nodes. The tree index is chosen by the overlay module. For example, a simple method is to use substreams, effectively the remainder modulo $t$ of the chunk index. Another method is to split the stream to descriptors using Multi Descriptor Coding (MDC) [Goy01]. Then, each tree can be used to disseminate a single descriptor of the encoded stream, the overlay module chooses the spanning tree based on the descriptor.

BitTorrent Live [Coh12, RKH14] takes a Multi-Tree approach, where each tree is called a club, allowing multiple parents in the same club. The choice increases the continuity of the stream at the cost of upload bandwidth consumed by redundant chunks being sent by multiple parents in the same club. Our decomposition of the BitTorrent Live protocol is illustrated in Figure 3.5.
Listing 3.3 The extract shows the source code of the CoolStreaming streaming module.

```java
public class CoolStreaming extends PullAlgorithm {
    public CoolStreaming(P2PClient client, OverlayAlgorithm<?> overlay,
        boolean serverPush, int maxOffset, int maxInitLatency) {
        super(client, overlay);
        NeighborChunkAvailabilityIngredient NCAB =
            (NeighborChunkAvailabilityIngredient) overlay
            .getIngredient(NeighborChunkAvailabilityIngredient.class);
        if (NCAB == null || NCAB.opMode != OperationMode.updateEveryRound) {
            NCAB = new NeighborChunkAvailabilityIngredient(  
                OperationMode.updateEveryRound, !serverPush);
            overlay.addIngredient(NCAB, client);
        }
        if (serverPush) {
            addIngredient(new SourcePushIngredient(), client);
        }
        addIngredient(new EarliestContinuousChunkVSInitIngredient(  
            maxOffset, maxInitLatency), client);
        addIngredient(new HandleChunkRequestsOnArrivalIngredient(), client);
        addIngredient(new CoolstreamingChunkRequestIngredient(), client);
    }
}
```

Pull-based Streaming

Pull-based streaming requires more communication and decision points. All pull-based streaming algorithms that we surveyed, however, made the same types of decisions, each of which we were able to successfully encapsulate as an ingredient. Accordingly, the core of the pull-based streaming module was virtually empty, as exemplified in Listing 3.3 that shows the CoolStreaming streaming module class as implemented in STREAMAID.

Chunk Availability Ingredient When pull based streaming is involved, neighboring peers exchange *bitmaps* containing information on which chunks each peer has. The bitmap exchange can be done periodically (as in CoolStreaming [ZLLY05] and PULSE [PPKB07]) or a bitmap can be sent for every new chunk received (as in Chainsaw [PKT05]). As chunk availability exchange only occurs between neighboring peers, this ingredient is always tied to a specific overlay.

Request Sending Ingredient Another decision in the pull-based streaming module is a choice between what chunks to send. That is, using the neighbor chunk availability and potentially additional parameters (such as latency, chunk time to deadline, requests already sent to neighbor, history with neighbor, etc.), the algorithm decides which chunks to request and from which neighbors. CoolStreaming sends requests based on rarest first but only to peers that can send the chunk before the deadline. Chainsaw sends out requests randomly, but limits the amount of
requests sent to each neighbor, whereas PULSE prioritizes peers using the exchange history and buffer window overlap as parameters.

**Request Handling Ingredient**  While CoolStreaming and Chainsaw handle requests right on arrival (except for some special handling by the source in Chainsaw), PULSE prioritizes request handling according to similar parameters as the request sending ingredient. Apart from disseminating chunks, the streaming module also initializes the player module. The streaming module sets the initial chunk to be played and the buffer time to wait before starting playback. We explore several different options for these ingredients in Section 3.3.

**Source Push Ingredient**  Finally, as in the overlay module, the stream’s source node may employ an algorithm different from other nodes. For instance, in an effort to reduce latencies, the source node can push the newly generated chunks to its neighbors regardless of the protocol run by other nodes. While other ingredients shown in this section are mandatory and are required for the operation of the module, Source Push Ingredient is optional and can be turned off at will. This ingredient is also tested in Section 3.3.

The mandatory ingredients required for a push-based streaming module are summarized in Table 3.2. Each column specifies an ingredient and lists the parameters that can be used for the operation of that ingredient.

### Push-Pull Hybrid Streaming

Push and pull algorithms can be combined by running them in parallel, normally by dividing future chunks between them. Here, chunks closer to the playback deadline are actively being requested by the pull algorithm while chunks further ahead are expected to be received using the push algorithm. The pull and push algorithms may use the same overlay or different overlays. For example, in mTreeBone [WXL07], the pull algorithm is the CoolStreaming streaming module using the CoolStreaming overlay, while the push algorithm uses the simple push-based streaming module described in Section 3.2.2, leveraging a tree overlay described in their paper. As for the division of chunks between the push and the pull algorithm, in mTreeBone [WXL07] the pull window is always kept one chunk behind the last chunk received by the push algorithm.
### Overlay Module

A fundamental building block of a P2P live streaming system is the underlying overlay. The overlay maintains two basic lists of nodes which are constantly updated: the set of *known* nodes and the set of *neighbor* nodes. Known nodes are populated either by querying an agreed upon *tracker* node [GKM03, MR07], through *gossiping* [MK04, ZLLY05], or using neighbors of another underlying overlay [PPKB07, ZLLY05]. Nodes are then chosen from the known list to be included in the neighbors list. Several factors influence the choice of electing node $p$ to be a neighbor:

- **Local state**: the local state of the node may include the current number of neighbors the node has, the average latency of the node, the chunks the node has, the total uptime of the node, and possibly the total upload bandwidth of the node, as summarized in Table 3.3.

- **Exchange history**: the chunks sent to $p$ and received from $p$ in total or in a recent time window.

- **Neighbor state**: the same properties as in the local state but of the potential neighbor $p$. Any such property that is used in the decision must be specifically sent by $p$ and kept up to date. Updates can be either scheduled or as soon as the change in the state occurs.

- **Link to potential neighbor**: latency or throughput between the node and the potential neighbor $p$.

If the protocol dictates symmetry between neighbors (*i.e.*, if $p$ is a neighbor of $q$ then $q$ must be a neighbor of $p$), then neighbor requests must be approved by the potential neighbor. The choice of accepting a neighbor can now involve the aforementioned factors.

Sometimes properties of the node requesting the connection differ from those of the node approving the request, for instance in tree-based overlays. Here, the requesting node is usually referred to as a *child* and the node receiving and approving the request is referred to as a *parent*. Hence, the list of neighbors is divided into two separate lists: a list of child nodes and a list of parent nodes with possible different treatment for different states of these lists.

The source node of a given live stream can employ a different algorithm than the other nodes. For instance, to battle free-riding and spread chunks more evenly, the source node may periodically switch neighbors to a random subset of the known nodes [PPKB07].

<table>
<thead>
<tr>
<th>Node state</th>
<th>Number of neighbors</th>
<th>Average lag</th>
<th>Chunk availability</th>
<th>Uptime</th>
<th>Upload bandwidth</th>
</tr>
</thead>
</table>

*Table 3.3: Node state components*
Table 3.4: Overlay design decisions

<table>
<thead>
<tr>
<th>Known nodes</th>
<th>Neighbor nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Underlying Overlay</td>
<td>Local state</td>
</tr>
<tr>
<td>Gossip</td>
<td>Neighbor state</td>
</tr>
<tr>
<td>Forward messages</td>
<td>Exchange history</td>
</tr>
<tr>
<td></td>
<td>Link state</td>
</tr>
</tbody>
</table>

Table 3.1 shows our classification of several popular overlay algorithms. The column “Known List Sources” summarizes the sources from which a node learns about other nodes in the system and stores them in the known list. For example, SCAMP initializes the known list using a tracker node and adds to the known list nodes that were included in messages forwarded from other peers (so-called forward messages). Further, Pulse runs SCAMP and uses its neighbors to fill up its known list. Next, the column “Neighbor List Dependencies” states the parameters that affect the decision to include a node from the known list in the neighbor list. SCAMP, for instance, only uses the current number of neighbors to decide whether to include a node in the neighbor list. In contrast, PRIME uses the number of parent neighbor nodes, its download bandwidth and the bitrate of the stream. Finally, symmetric overlays have parameters for accepting neighboring requests. We state them in column “Request Reply”. CoolStreaming, for example, only uses the current number of neighbors to decide whether to accept a request while Araneola also requires the amount of neighbors the requesting node has.

As overlay construction algorithms greatly differ, most of the logic resides in the core of the overlay module. However, we were able to extract some general ingredients applicable to many overlay construction algorithms.

Information Exchange Ingredient

Table 3.1 shows that overlay modules can depend on the state of other nodes. Hence, some information exchange via messages has to take place to provide this information. We identify several approaches of information exchange.

Whenever a node sends a connection request message, some symmetric overlay protocols may need to be informed about the requesting node to decide whether to accept the connection. Also, other components using the overlay may need specific information about neighbors as soon as a neighboring connection is established. For example, the Araneola overlay requires the current number of neighbors of the requesting node to decide whether to accept the connection.

For symmetric overlay protocols that have connection acceptance messages, a component using the overlay may need to piggyback additional information as soon as a connection is established. This information exchange happens only once during the connection process. Thus, the information exchanged is usually either needed for the connection process itself, or is constant and needs to be sent only once. Information that in flux and is required continuously by components to operate can be sent either periodically to all neighbors, or when the information
is updated. To prevent redundancy in different components that require information exchange and facilitate piggybacking of messages, all components that need information about other nodes can use the information exchange ingredient and specify the exact information they need to exchange, the type of exchange, and the overlay the exchange is tied to since the exchange is always carried out among neighbors in some overlay.

**Bootstrap Ingredients**

Every P2P live streaming protocol requires each peer to execute some procedure when the peer joins the system. Those bootstrapping actions usually include connecting to a centralized location – a tracker – and asking for a peer or a group of peers already running the protocol. Alternatively, a peer may connect to the P2P overlay using an existing peer or a group of peers already in the overlay. This behavior is very similar across P2P streaming protocols. We have identified several possible bootstrap ingredients: requesting a random peer, requesting a random group of peers (of configurable size), and requesting a group of peers that joined recently (of configurable size).

An overlay module can function without a bootstrap ingredient if the module uses another overlay model as a membership service. This is the case in the Coolstreaming, Chainsaw [PKT+05] or PULSE [PPKB07] overlays that use SCAMP as a basic membership overlay. This is also the case in Araneola [MK04] that uses a basic gossiping overlay as a membership service.

**3.2.4 Network Module**

By definition, the network module should be protocol independent. Thus, the only ingredients that are applicable are system-wide ingredients that intercept all messages and are also protocol independent. An example of such an ingredient is an encryption/decryption ingredient that encrypts outgoing messages and decrypts incoming messages. Other examples include logging, auditing, and checksum enhancements.

**3.3 Evaluation**

We evaluate how the STREAMAID framework works in practice and the extent to which ingredients provide modularity, extensibility and optimizability by experimenting with protocols expressed within the framework. We use the CoolStreaming and BitTorrent Live protocols as case studies.

**3.3.1 Setup and Measurements**

The metrics we adopt concern the viewing experience of end-users: (i) the average time it takes from login to the playback of the first chunk (*Startup Delay*); (ii) the average time from chunk
Figure 3.4: We test our implementation of the BitTorrent Live protocol with startup buffer of 2 sec. Figure 3.4(b) shows different missing chunk handling methods. In Figure 3.4(a) we use the skip ingredient for handling missing chunks and test various download connection ranges. In Figure 3.4(c) we use the skip ingredient for handling missing chunks, 2-3 parent nodes and test different latency ranges for the Latency Range adaptive playout ingredient. Note that a higher Continuity Index and lower Latency and Duplicate% are better. We find that always skipping or always waiting for a missing chunk is better than waiting and then skipping. Also, the balance between Continuity Index and Duplicate chunks through number of in-club download connections as well as the trade-off between Latency and the Continuity Index through Adaptive Playout are evident.

generation to chunk playback (Latency), (iii) the average fraction of chunks played of those that are playable (Continuity Index) (CI) and (iv) the percentage of users for which the Continuity Index is perfect (Perfect Continuity Index%, PCI%). We measure the CI and PCI% only for users who actually played chunks. We also define the Zero Playback% metric as the percentage of users who did not play any chunks at all. We ran tests in STREAMAID using both the PeerSim simulator [MJ09] and deployment on Planetlab.

In PeerSim, message latencies are distributed uniformly between 200 and 400 ms. All tests on PeerSim use LogNormal(\(\mu = 4.29, \sigma^2 = 1.28\)) distribution of session lengths [MRG07] and failed peers are immediately replenished. Each peer has upload bandwidth limit of 5.6 Mbps and the source’s upload limit is 16.8 Mbps. Each test simulates 300 seconds of a 300 Kbps stream on 300 peers, while the stream generation starts at second 10. Each result shown is an average of 10 runs with different initial random seeds. We test the popular CoolStreaming streaming algorithm with two different overlays: the original overlay used by CoolStreaming
(with $M = 4$) and the Araneola Overlay (with $L = 3$ or 4). The $L$ and $M$ parameters of the overlays were chosen so that the overlays would have a similar node degree.

On PlanetLab, we tested on 200 nodes and ran each test 5 times. We did not induce extra churn. However, PlanetLab nodes exhibit highly heterogeneous latencies and responsiveness, producing behavior reminiscent of actual churn. On PlanetLab, we tested our ingredient-based implementation of the BitTorrent Live protocol as described by Cohen et al. [Coh12].

For each of the following sections, we pick a single design decision and assess its effect on the aforementioned metrics. Each such design decision is implemented as an ingredient in STREAMAID.

### 3.3.2 Case Study: BitTorrent Live

BitTorrent Live was a highly anticipated decentralized live streaming protocol from Cohen et al., the author of BitTorrent [Coh12], and which to the best of our knowledge has not been implemented independently before. Figure 3.5 shows our decomposition of the BitTorrent Live protocol into ingredients. In the overlay module, a newly joined peer contacts the tracker and receives a partial list of online peers in the bootstrap ingredient. The peer also receives number of clubs to join and decides to join them or choose different clubs to join in the ClubJoin ingredient. For every club, the peer gathers information on other peers in that club in an instance of the ClubInfo ingredient, and forms connections with peers in the club in the NeighborAdd ingredient, which also preserves upper and lower bounds on in-club and outer-club connections. Parent nodes that fail to send chunks on time are removed in the ParentRemove ingredient. In the streaming module, in order to reduce the amount of redundant chunks being sent by multiple parents, children update their in-club parents upon receiving a chunk in each club in the Chunk Availability ingredient. In the player module, the speed of the playback to adjusted to keep the latency bounded in the Latency Adaptive Playout ingredient and different options of handling missing chunks (skip, wait, wait and then skip) are also implemented as ingredients. Chunk authenticity is validated by the network module upon reception by the chunk hash validation ingredient.

**Handling of Missing Chunks**

When the player cannot play the next chunk because it is missing it will halt. The BitTorrent Live patent proposes several options of handling this case: Wait for the missing chunk, skip the missing chunk or wait for some time and then skip. Each of these options was implemented in STREAMAID as an ingredient. As can be seen from Figure 3.4(b) (where waitX means waiting for X seconds and then skipping), skip and wait have the best continuity index and waiting and then skipping only hurts the continuity index while also increasing latency. It should be noted that the latency is marginally increased when waiting for missing chunks, but not as much as waiting for more than 1 second and then skipping.
Number of In-Club Parent Connections

Another interesting parameter of BitTorrent Live is the number of in-club download connections. As shown in Figure 3.4(a), as the number of download connections grow, the Continuity grows, but so does the percentage of duplicated chunks that are sent. We can see that in these settings, having two parents is enough for good continuity index and only yields about 14% duplicates. Note that even when there is only one parent there is a small amount of duplicate chunks due to parent switching.

Latency Adaptive Playout Ingredient

As suggested in the BitTorrent Live patent, we have added an ingredient that slows down or speeds up the playback in order to reach some set latency range. As was expected, and evidenced on Figure 3.4(c), higher ranges improve the CI, but, increase the latency.

3.3.3 Case Study: Pull-based Streaming

As discussed earlier, a decomposition of a pull-based algorithm into ingredients is shown in Figure 3.3. Our focus is on variants of CoolStreaming [ZLLY05] and Araneola [MK04].

Source Push Ingredient

The source node can push newly generated chunks to its neighbors by running a push-based protocol regardless of the streaming protocol run by other nodes. We test this ingredient for several different settings. Figure 3.6 shows that enabling source push always improves the results. In all settings tested, enabling source push lowered the Startup Delay by 0.2-0.3 seconds.
Figure 3.6: CoolStreaming source push. Source push evaluation of CoolStreaming using the Araneola Overlay ($L=3$ and $4$) and the original CoolStreaming overlay ($M=4$) waiting for missing chunks and skipping missing chunks. All algorithms use startup buffer of 2 seconds and start buffering from the first bitmap or chunk. Enabling the source push ingredient can only improve performance.

and Latency by 1.2 seconds on average and always improved both the CI and the PCI%. We enabled source push in all remaining experiments.

Player Initialization Time Ingredient

We now test how the point in time when the player module is initialized affects the system performance. One option is to initialize the player module on startup. Another is to wait for a bitmap or chunk to be received.

As can be seen in Figure 3.7, the only advantage of initializing the player module from the startup is reduced latency. However, when skipping missed chunks, more than 80% of the users do not manage to play any chunks at all, and those who do play it with a very low
Continuity Index. This happens because when skipping missing chunks, the latency is kept constant and many users cannot get new chunks in 2 seconds (the buffering time for this experiment). Conversely, when waiting for missing chunks, initializing the player module from startup reduces latency a bit but at the cost of slightly reduced Continuity Index and greatly reduced Perfect Continuity Index%.

When initializing the player module from the startup, the Startup Delay is also higher.
The extract shows the source code that initializes the player module’s video stream, according to the EarliestContinuousChunk policy.

```java
@override public void nextCycle() {
    if (neighborAvailability.isEmpty() || client.isServerMode()
        || client.getVideoStream() != null) {
        return;
    }

    long maxChunk = Long.MIN_VALUE;
    for (final Availability avail : neighborAvailability.values()) {
        final long currMaxChunk = avail.getEarliestContinuousChunk(maxOffset);
        if (currMaxChunk > maxChunk) {
            maxChunk = currMaxChunk;
        }
    }

    client.initVideoStream(Math.max(maxChunk,
        Math.max(0, Utility.getMovieSecond() - maxLatency)));}
```

because the player module is initialized with the current movie time without taking into the account the state and latency of neighboring peers. Hence, in all following experiments, we initialize the player module from the first chunk or bitmap.

### Player Initialization Position Ingredient

When the player module is initialized by the streaming module, the streaming module must first choose a chunk from which the playback would start, while also striving to keep latency low. We achieve better performance when initializing the player module only upon receiving a bitmap or chunk. When the player is initialized after receiving a bitmap, there are several possible playback starting positions, which we explore in an experiment. We propose a parameterized ingredient that begins the playback at most $k$ chunks before the most recent available chunk reported in the bitmap. In order to also increase the likelihood of continuous playback, the ingredient initializes the play module to the beginning of the longest consecutive sequence of recent chunks, bounded by $k$. However, since the first bitmap may be received from a peer who lags behind, we limit the maximum allowed latency of the initialization position, as shown in Listing 3.4.

As can be seen in Figure 3.8(a), the Startup Delay is unaffected by the setting of $k$ since the initialization of the player happens at the same time regardless of $k$. However, other metrics are greatly affected. In Figure 3.8(b), we see how the latency grows logarithmically with each added chunk. Also, Figure 3.8(c) and Figure 3.8(d) show logarithmic improvement of the CI and the PCI%. Although the improvement of the CI is marginal (even with $k = 0$ the index is already at 0.978), the PCI% grows dramatically (Figure 3.8(d)). We observe that for both the CI and the PCI% the improvement stops for $k$ larger than 3-5 (depending on the overlay), while
the latency keeps growing, albeit at a slower pace. We conclude that initializing the player to start at a chunk earlier than the last one available strikes a trade-off between the PCI% and the Latency while keeping the Startup Delay constant. In other words, assuming a hit in latency is tolerable, increasing $k$ boosts Continuity, and greatly increases the percentage of users with Perfect Continuity.

**Buffer Size in CoolStreaming**

After the player module is initialized, the player module buffers for some time before starting playback. We ran tests for four different algorithms with source push enabled and with the player module initializing upon receiving the first chunk or bitmap. As can be seen in Figure 3.9, all metrics increase when the buffer grows. Algorithms that wait for missing chunks have a high Continuity Index even for a low startup buffer of 1 second, while Latency and Startup Delay (graph omitted) are mostly affected by the overlay choice (Araneola vs. CoolStreaming Overlay). Thus, the CoolStreaming streaming module with the Araneola overlay module that waits for missing chunks both gets a very high Continuity Index and low Latency and Startup Delay with a startup buffer of only 1 second.
Figure 3.9: Startup delay, Latency, Continuity Index and Perfect Continuity Index% of CoolStreaming streaming module using Araneola Overlay (L=3) and CoolStreaming overlay (M=4) waiting for missing chunks and skipping missing chunks with various startup buffer sizes. The algorithms start buffering from the first bitmap or chunk and use source push. CoolStreaming streaming module with the Araneola overlay module which waits for missing chunks both gets a very high Continuity Index and low Latency and Startup Delay with a startup buffer of only 1 second.

Skip Chunks Ingredient in CoolStreaming

Subsection 3.2.1 explains that the player module can reach a state where the next chunk to be played is missing. We evaluate an ingredient that skips playback of a missing chunk only if the following $n$ chunks are present, with $n$ being a parameter of the ingredient. Note that when $n = 0$, the ingredient always skips when a chunk is missing, while $n$ larger than the latency will always result in waiting for the missing chunk. We conclude that while always skipping chunks is inherently a poor user experience, as can be seen in Figure 3.10, there is not much difference between $n > 0$ and always waiting for missing chunks. Moreover, always waiting for missing chunks yields some latency benefits and allows several optimizations to be made since peers can always know that if a peer reached some chunk in playback, it has all previous chunks since joining the stream.

Adaptive Playout Ingredient

We discussed in Subsection 3.2.1 how Adaptive Playout can be used to decrease Latencies and Startup Delays. The Adaptive Playout ingredient may only depend on the window size. In
Figure 3.10: Startup Delay, Latency, Continuity Index and Perfect Continuity Index% of CoolStreaming streaming module using Araneola Overlay (L=3) and CoolStreaming overlay (M=4) with startup buffer of 1 and 2 seconds and several settings of number of seconds to wait before skipping a chunk. The algorithms start buffering from the first bitmap or chunk and use source push. There is not much difference between \( n > 0 \) and always waiting for missing chunks, waiting is even preferable.

Listing 3.5, we show an example code of a simple Adaptive Playout ingredient implementation that tries to reach a specific pull window size defined by a parameter. More sophisticated ones may leverage other available information. We propose an Adaptive Playout ingredient which tries to measure the minimal window size required for maximal resilience.

Our algorithm works as follows. We measures how long before the playback deadline missing chunks get before they are received. These measurements are being averaged with exponential weights using a sliding window. If chunks are not received before the deadline, then the buffer is too small. Otherwise, if chunks are received long before the deadline, the buffer can be shrunk without losing continuity. We set a target buffer and change the playback speed to reach the target buffer size. The speed change is limited to 10% to minimize annoyance for the user.

In Figure 3.11, we test the CoolStreaming streaming module with Araneola overlay module \((L = 3)\) using startup buffer of 1 sec, and for several player module initialization position values \(k\). We found that increasing the target buffer behaves in a similar fashion as increasing \(k\): the latencies grow linearly while the Continuity Index and the Perfect Continuity Index% grow logarithmically reaching a plateau at roughly 7 chunks. The methods can be combined
Listing 3.5 Adaptive playout ingredient. The extract shows the source code to implement a very simple Adaptive Playout ingredient that tries to reach a set pull window size.

```java
@override public void nextCycle() {
    final VideoStream videoStream = client.getVs();
    if (videoStream == null || client.isServerMode() || videoStream.isBuffering()) {
        return;
    }
    final int pullWindowSize = (int) (Math.min(pullAlg.maxChunkToRequest,
        Utils.getMovieSecond()) - (videoStream.windowOffset - 1));
    if (pullWindowSize > targetBufferSize + 1) {  
        videoStream.playSpeed = 1 + maxSpeedChange;
    } else if (pullWindowSize < targetBufferSize) {
        videoStream.playSpeed = 1 - maxSpeedChange;
    } else {
        videoStream.playSpeed = 1;
    }
}
```

to reach very high Perfect Continuity Index%. For instance, with \( k = 4 \) and the target buffer of 7 chunks, the PCI% hovers above 85% with an average latency of 12 seconds.

3.4 Debug Ingredients

In Section 3.3, we saw several design decisions and how they affect several important metrics. Here, we propose several protocol independent ingredients that can aid in debugging any P2P live streaming protocol:

- **CHUNKDROPINGREDIENT** drops a configurable portion of the chunks messages sent by a peer. This network module ingredient can be used to test how the system copes with freeriders that omit chunk messages.

- **BITMAPDROPINGREDIENT** drops a configurable portion of the bitmap messages sent by a peer. This network module ingredient can be used in pull-based streaming systems to test the system in the presence of more sophisticated freeriders that do not fully disclose their chunk availability. The ingredient works with any streaming protocol and uses aspect oriented programming (AspectJ) to prevent the streaming algorithm from sending stream chunks.

- **DOUBLEREMOVINGREDIENT** This overlay module ingredient provides a disconnection service such that the connection between two peers is removed by both peers. This prevents one sided connections from occurring when a peer is forcibly disconnected using one of the ingredients.

- **INVALENTHINGREDIENT** is a debug ingredient residing in the streaming module that verifies that a specific logical invariant holds. A logical invariant is a condition that
Figure 3.11: CoolStreaming/Araneola Adaptive Playout. Startup Delay, Latency and Continuity Index and Perfect Continuity Index% of CoolStreaming streaming module using Araneola Overlay ($L = 3$) with startup buffers of 1 sec, using the Adaptive Playout ingredient with various target buffer sizes and several playback initial position values. All algorithms start buffering from the first bitmap or chunk, use source push and wait if there is no chunk to play. We see that increasing the target buffer behaves in a similar fashion as increasing $k$ (the player module initialization position) and both methods can be combined to reach near optimal values of Perfect Continuity Index.

must always be true for the duration of the stream while all peers follow the protocol. We propose the following invariant: If Peer $A$ received a chunk at time $X$, and peer $B$ was a neighbor of peer $A$ from $X$ through $X + 3$ (seconds), then at time $X + 3$ peer $B$ must also have that chunk. Note that the check can be completed both by peers $A$ and $B$ since they exchange chunk availability information. This invariant can be effectively used to debug protocols and protocol implementations: the invariant must hold true for all nodes throughout the entire streaming process.

- **BLACKLISTINGREDIENT** is used as a service by other ingredients. The ingredient maintains a black list of peers and initiates immediate disconnection from neighbors that are also in the black list. The ingredient uses the **DOUBLEREMOVEINGREDIENT** for the forced disconnection.
3.5 Related work

**Ingredients.** Coarse grained decomposition of complex functionality to provide modularity and extensibility has been considered before in different settings, such as micro-protocol layers in replication systems [MPR01, vRBM96], Portable Interceptors in OMG’s CORBA communication broker [COR], and RPC style remote method invocations augmented with channels in Microsoft’s Windows Communication Foundation (WCF) [Kle07]. Our ingredient abstraction is inspired by these works, while cast in the P2P live streaming environment.

**P2P Live Streaming.** There are shrewd surveys of P2P live streaming systems and principles [LGL08b, ZH12a, HLR08, RPI12, SMG+07]. Most of these works classify systems into tree-based or mesh-based [HLR08, MRG07], while Zhang et al. [ZH12a] provide a taxonomy for classifying P2P live streaming protocols. ShadowStream [TAYZ12] introduces methods for transparently embedding a live streaming protocol to be evaluated into large-scale live streams without affecting the quality for viewers, but relies on access to a production Internet live streaming network.

**Chunk scheduling.** Many works focus on the chunk scheduling aspect of P2P live streaming. Zhou et al. [ZCL11] give an analytic evaluation of the RAREST-FIRST and GREEDY strategies for chunk request scheduling using their own stochastic model and propose a new mixed strategy that achieves the best of both worlds. Shakkottai et al. [SSY11] also evaluate these strategies for minimizing the buffer size and propose a hybrid policy that reduces the required buffer size to ensure high probability of chunk playout. Zhao et al. [ZLC09] propose a general and unified mathematical framework to analyze a large class of chunk selection policies. Other works propose and evaluate various chunk scheduling algorithms in different settings. Liang et al. [LGL09] test five chunk scheduling algorithms in a variety of settings such as different source upload bandwidth, buffer delays, source chunk scheduling algorithms and node degrees. Our work corroborates many of their findings, and expands to other design decisions made by P2P streaming algorithms.

**Overlays.** Other works cover the overlay building aspect of P2P live streaming. Liu et al. [Liu07] analytically derive a new overlay for push-based dissemination; Zhang et al. [ZH12b] evaluate two different overlay construction strategies: a RANDOM overlay choice where a peer selects neighbors without considering their network locations, and a NEARBY-OVERLAY where a peer only neighbors with nearby peers. In some systems, upload bandwidth affects the number of neighboring peers [dSLMM08].

Our work encapsulates the concerns discussed in the literature about chunk scheduling, buffer delay, source chunk scheduling and overlay construction into ingredients, allowing them to be evaluated, improved and reasoned about while keeping other aspects of the P2P live streaming system fixed. By systematically applying our abstraction, we also identify several other concerns that we feel have been generally overlooked, such as the player module initialization time and position, and show how they affect the overall performance of the P2P live streaming protocol.
3.6 Conclusions

We presented the ingredients concept and an implementation of the STREAMAID framework for constructing extensible middleware for P2P live streaming. We disentangled several such protocols into their most basic design decisions, uncovering in the process commonality between several systems.

Our large-scale experiments illustrate the power of the abstraction and the flexibility of STREAMAID. We show how continuity can be traded for latency or duplicate chunk percentage in the BitTorrent Live protocol; how pull-based streaming protocols always benefit from the source constantly pushing fresh chunks to its neighbors; how choosing the first chunk in playback is an implicit trade-off between latency and playback continuity; how adaptive playout can be used for the same trade-off, and how seemingly minor design decisions significantly impact the overall performance of the live stream. We believe the ingredients abstraction and our open-source framework can help accelerate development and discovery for future P2P live streaming systems.
Chapter 4

A Dynamically Adaptive Framework for P2P Live Streaming

This chapter reports on our search for an adaptive P2P live streaming protocol to operate in a wide range of conditions, even as the environmental conditions vary.

This is a challenging property to orchestrate. First, live streaming protocols may work differently in different settings, compounding the complexity of an adaptive approach. Second, the impact of the environment on a protocol may not be fully understood which creates the need for profiling and experimentation within the protocol. Finally, the protocol should allow for adaptation to new quantifiable metrics without requiring major changes.

Rather than devising a new and complex adaptive protocol, we developed a framework, SMARTSTREAM, which learns from local information from peers and switches the system autonomously to the best alternative among a selection of established live streaming algorithms from the literature. SMARTSTREAM is built on ideas from learning theory, specifically the multi-armed bandits (MAB) algorithms for sequential decision making with incomplete information [ACBFS95, ACBF02, Agr95], allowing us to significantly reduce complexity. We have surveyed a large number of systems and protocols and have identified the essential functional blocks of the adaptive live streaming problem that each address a specific concern. By combining specific instances of these functional blocks, we are able to implement every system we surveyed. Furthermore, we are even able to compose new combinations of these functional blocks to create novel and original live streaming systems.

In SMARTSTREAM, we run a MAB training phase under various extreme networking situations to allow the framework to learn which combinations respond best to each condition. After the learning phase, we use an adaptive protocol that dynamically changes its settings to match the best learned combination for the situation at hand. We stress that in our framework, the system has no explicit knowledge of the network settings. Rather, SMARTSTREAM’s behavior is based on local measurements and observations, such as peers’ upload and download bandwidth, reliability and availability levels, and their churn rate.

We conduct systematic exploration of SMARTSTREAM using simulations with PeerSim [MJ09], which resulted in new insights about P2P live streaming protocols. More specifically,
we begin by evaluating the trade-off between latency and stream continuity under different states of the environment. We find that push-pull algorithms fare better in environments with low upload bandwidth than competing approaches, with different variants working better in low and high churn settings. We also observe that the selection strategies for when multiple neighbors have a desired data chunk play minimal role in the performance of the algorithms in terms of latency and stream continuity. Finally, we discovered that the overlay topology, specifically the average node degree, has significantly more effect on stream continuity than specifics of the streaming algorithm itself.

The main approach behind SMARTSTREAM is general and may be applicable to other distributed computing problems besides P2P live streaming. In particular, we hypothesize that problems that have many different solutions, no single one of which is optimal across all states, may be amenable to our approach to adaptivity, especially if it is possible to define a consistent reward function across solutions and switch between solutions without significant or lasting performance impact.

**Contributions.** This chapter makes the following contributions.

- We describe SMARTSTREAM: a framework for harnessing P2P live streaming protocols to make them adaptive to dynamic environments.

- We demonstrate experimentally how multi-armed bandits (MABs) from learning theory can be used to make adaptive decisions in a distributed system.

- We evaluate SMARTSTREAM on diffuse and changing environments and peer configurations on PeerSim [MJ09] against a multitude of existing and explicitly composed P2P live streaming protocols.

**Roadmap.** The remainder of the chapter is organized as follows. We give a self-contained introduction of multi-armed bandit algorithms in Section 4.1, and then show a methodology for how our SMARTSTREAM framework uses these algorithms to optimize P2P live streaming settings in Section 4.2. Using simulations, we report on a series of experiments we conducted to calibrate parameters and confirm the validity of our approach in Section 4.3. We survey related work in the P2P live streaming literature in Section 4.4 before concluding in Section 4.5.

### 4.1 A MAB Primer

The core challenge in our approach is to periodically choose the most appropriate live streaming protocol to use from a set, even with no or limited knowledge about how the protocol will perform in the circumstances at hand. To make progress, we leverage so-called multi-armed bandit (MAB) techniques from learning theory that have recently been developed to address sequential allocation problems defined by a set of actions [BCB12]. During every time step $t$ of the sequence, an action $I_t \in \{1, 2, \ldots, N\}$ is chosen and an observable reward $X_{I_t, t} \in [0, 1]$ is subsequently obtained for the chosen action – no payoffs are revealed for other possible actions.
The goal of multi-armed bandit problems is to maximize the total rewards over time even though decisions are taken with limited information. A common metric is to compare the total rewards accrued by the algorithm to what the best action would have accumulated in hindsight. This measure is called the expected (pseudo-) \textit{regret} of the algorithm, more precisely defined as

\[
R_T = \max_{i=1, \ldots, N} E \left[ \sum_{t=1}^{T} X_{i,t} - X_{I,t} \right],
\]

where the expectation is taken over both the randomness in choosing an action and the environment’s randomness in assigning rewards.

To succeed, MAB algorithms naturally balance \textit{exploitation} of actions that did well in the past against \textit{exploration} for actions that may yield higher future rewards. The adaptation that occurs over time makes MAB particularly appropriate for the P2P protocol selection as it allows the overall system to respond to dynamic changes in scale and environment.

In the context of P2P live streaming, the P2P protocol choices can be viewed as choices, and the sequence is defined by the times when these choices are made. The rewards are defined by functions over the quantifiable metrics that the operator desires to optimize, such as playback continuity. These configurations are described further in the following section.

### 4.1.1 $\varepsilon$-greedy algorithm

Rewards act as a one-dimensional signal about the quality of a strategy choice. First, assume that the rewards are chosen independently and with identical distribution (iid), the so-called \textit{stochastic} setting for MAB. Let $\mu_i$ for $i \in \{1, \ldots, N\}$ denote the expected value of choosing action $i$, and set $\mu^*$ as the smallest (optimal) expected value.

A simple and widely used heuristic for stochastic MAB problems is the $\varepsilon$-greedy algorithm [LR85, Agr95] and variants thereof. At each time step $t$, the algorithm greedily chooses the action with highest empirical mean with probability $1 - \varepsilon_t$ (exploitation), and chooses a random algorithm with probability $\varepsilon_t$ (exploration), where the $\varepsilon_t \in [0, 1]$ family forms a parameter \textit{schedule}. The schedule is an important parameter setting for the $\varepsilon$-greedy algorithm which we will discuss in more details for the live streaming context below.

The $\varepsilon_t$ does not have to be a fixed constant. When $\varepsilon_t$ is a function which decays as the reciprocal of $t$, the per-step regret $R_T/T$ of $\varepsilon$-greedy converges to 0 with probability 1 [Agr95]. More precisely, the expected pseudo-regret in step $T$ in this case is at most

\[
\overline{R}_T \leq 1 + \frac{2N \log (2NT)}{(\max_{1 \leq i \leq N} (\mu_i - \mu^*))^2},
\]

assuming the denominator is positive [Agr95, ACBFS95].
4.1.2 The UCB1 algorithm

A different class of algorithms for stochastic MAB aims to improve on the unguided exploration used by the $\varepsilon$-greedy strategy by paying closer attention to what information it can glean from each action rather than only focusing on accrued rewards. The upper confidence bound algorithm (UCB1) [ACBF02] maintains upper bounds on the likely rewards for each action and then chooses the action with the highest such estimate, thus meticulously guiding exploration and exploitation. After playing each action once, UCB1 proceeds as follows in every time step $t$.

First, the algorithm updates its internal upper confidence bound estimate $\hat{\mu}_{i,t}$ of each action $i$ with the latest empirical information:

$$\hat{\mu}_{i,t} = \frac{1}{|\xi(i,t)|} \sum_{s \in \xi(i,t)} X_{i,s}$$

where $\xi(i,t) = \{1 \leq s \leq t \mid I_s = i\}$ is the set of time steps when the algorithm chose action $i$ before time $t$. The algorithm then selects an action for time $t$ according to a Chernoff-Hoeffding style bound:

$$I_t = \arg \min_{1 \leq i \leq N} \left( \hat{\mu}_{i,t} + \sqrt{\frac{2 \log t}{|\xi(i,t)|}} \right).$$

The UCB1 algorithm is known to be both efficient and give results close to optimal in stochastic MAB settings, providing high-probability guarantees on the expected regret. Specifically, the expected pseudo-regret of UCB1 is

$$\mathcal{R}_T = O\left( \log \frac{T}{\max_{1 \leq i \leq N} (\mu_i - \mu^*)} \right),$$

again assuming the denominator is positive [ACBF02, ACBFS95].

4.2 Applying MAB to P2P Live Streaming

Here, we describe the approach we took in developing an adaptive protocol for P2P live streaming. We specifically refrain from proposing a new protocol, and instead focus on being able to identify the “best” pre-existing protocol to use by a collection of peers with respect to known metrics. To quantify the quality of our decisions during adaptation, we use the following well-known metrics for measuring live streaming performance.

- The **Continuity Index** measures the normalized rate at which a peer obtains new chunks once playback has started, capturing interruptions in the stream. For example, a continuity index of 1 means that the stream playback on the peer’s machine was uninterrupted throughout the whole session.

- The **Clear Stream Ratio** is the fraction of peers whose streams have been fully uninterrupted, *i.e.*, the percentage of peers with continuity index of 1.
Figure 4.1: SMARTSTREAM overview. The framework uses the state of the environment (such as high churn rates) to choose between different instantiations of a MAB algorithm. The selected MAB algorithm in turn maintains internal weights (or bounds) on a set of potential live streaming algorithms that may be used. These weights are adjusted based on “rewards” from the environment, which are success rates of desirable P2P live streaming metrics using the chosen algorithm. An algorithm with worse streaming performance than another will reap fewer rewards, thus decreasing the probability of it being chosen by the framework in similar environmental states in the future.

- **Latency** measures the average length of time between a data chunk being generated and being played by a peer.

The key question we need to answer regards how we choose a streaming protocol, under the assumption that different P2P live streaming protocols have disparate operating behavior in different settings. (We will verify this assumption in Section 4.3.) Moreover, the protocol switch must be adaptive, *i.e.*, happen automatically in response to changes in the environment.

The MAB framework introduced above fits the scenario well, allowing well-reasoned dynamic changes even when feedback is only obtained for the algorithms that have been tried.

An overview of our approach is provided on the diagram in Figure 4.1, showing how the environment interacts with the MAB corresponding to the current perception of the environment state (such as “high churn, low upload bandwidth”). The chosen MAB periodically selects and adopts a promising P2P live streaming protocol to use, and then in turn reaps “rewards” from the environment that provide feedback on how well this protocol performed given the current settings.

The use of MAB algorithms for P2P live streaming protocol selection prompts several questions:
1. What are appropriate candidate streaming protocols?

2. Which MAB model and algorithm should be used?

3. How do we define the reward used by the MAB algorithm to rank the quality of different protocols?

4. How to dynamically switch between candidate protocols on the fly with minimal performance degradation?

We will address these questions in order.

### 4.2.1 Selecting Candidates

A large number of live streaming protocols have been suggested in the literature [MRG07, ZH12a]. Several works have demonstrated that a large family of P2P live streaming protocols can be decomposed into several well-defined and inter-operable atomic entities [LGL09, MSR+14, WRRH14].

In this work, we use our STREAMAID framework. Recall that STREAMAID defines modules that compose every P2P live streaming algorithm:

- **Overlay**: An overlay maintenance module which creates and manages neighboring connections that in turn form the overlay network. Sometimes the overlay component is broken down further into a peer sampling sub-component and a neighbor-selection sub-component which chooses neighbors from the peer sampling sub-component.

- **Streaming**: A chunk dissemination component that uses the overlay network constructed by the overlay maintenance component to disseminate chunks. The streaming component can operate in both pull and push modes, or a combination of both. When operating in push mode, the streaming component can guarantee that it is instantiated only with tree based overlays.

- **Player**: An application component which handles chunk playback.

Each of the above components can have a myriad of implementations, and multiplexing those implementations yields an exponentially larger set of possible candidate protocols. In our study, we choose several popular implementations for each of these components. As a first step, we try all possible combinations in several network settings to single out combinations of building blocks that perform well.

Liang *et al.* showed that churn and upload bandwidth are the main impact factors on the performance of P2P live streaming protocols [LGL09]. We have thus chosen to test the various candidate protocols under different churn and peer upload bandwidth settings.

We remark that MAB is an ineffective approach when presented with too many candidates (actions), as both complexity and regret grows. To make the best use of the tool, we initially identify a small number of good candidates and continue with them. In Section 4.3 below, we
show how continuity index balances with Latency when changing different algorithm settings, and select the algorithms that provide the best continuity index for the next experiments.

### 4.2.2 General MAB Approach

The P2P tracker registers the join rates and upload bandwidths of new peers. Peers periodically send heartbeat to the tracker, allowing the tracker to approximate the churn rate and upload bandwidth distribution. Whenever a change in the network settings is detected, SMARTSTREAM takes the following steps.

- If the best protocol for the new state is known, we switch to that protocol.
- Otherwise, we invoke MAB in order to learn the new best protocol among the set of candidates.

Specifically, we employ a stochastic MAB-based method to choose the next algorithm.

In SMARTSTREAM, each candidate streaming algorithm serves as an action in the MAB algorithm. We partition time into discrete time steps (or rounds) of fixed intervals, and use the performance data collected by the tracker for the previous round to calculate the reward for that round. Every round, the tracker calculates the reward using some customizable reward function and then employs a stochastic MAB algorithm to decide on the next algorithm to use.

The next question is how the reward function should be defined. Our reward function should reflect the performance of the current algorithm on the overall live streaming system. We identified several local metrics that can faithfully reflect the global continuity index.

- **Chunks Received (CR)** is the number of chunks a single peer has received during the measured time period.
- **Chunks Received Available (CRA)** is the number of chunks a peer has received out of the missing chunks that were available at his neighbors. This metric can only be evaluated if the availability of chunks at neighbors is known.
- **Continuity Index (CI)** the number of chunks played by the peer during the measured time.

We then run several combinations of different algorithms from Subsection 4.2.1 while measuring the three metrics and the true continuity index to form a set of training data. Using the data collected, we train a linear regression model combining the three metrics to best approximate the global continuity index.

Finally, we compare UCB1 [ACBF02] and \( \varepsilon \)-greedy [ACBFS95, Agr95] stochastic MAB algorithms with several variations of the \( \varepsilon \)-schedule using our computed reward function to see how quickly SMARTSTREAM converges to the ideal streaming algorithm in every state. For this work, we specifically chose simple stochastic MAB algorithms, as we believe that when the environment is stable, the necessary assumptions hold. We remark that it is possible that other variations of MAB algorithms may be more suitable for some settings (*e.g.*, bandits with switching costs [AT96]). We defer the study of more advanced models to future work.
4.2.3 Switching Between Protocols

Shifting between protocols must be conducted carefully to avoid the very performance or availability problems our work set out to avoid. Whenever SMARTSTREAM decides to switch protocols based on the mechanism described above, the following steps are taken:

- Each client periodically (by default, every 10 seconds) sends performance data to the tracker – possibly ricocheted through other nodes.

- The tracker periodically (by default, every 100 seconds) decides on a new live streaming algorithm to use for the entire overlay network. If the algorithm is different than the currently running algorithm, then the tracker propagates a ChangeProtocol message specifying the new algorithm and which chunk is the first to be distributed using the new algorithm.

- Upon receiving the ChangeProtocol message, each peer starts running both the current protocol and the new protocol. To account for peers that might lag behind, each peer runs the old protocol for $K = 20$ more chunks after switching to the new one. Then, the peer disconnects from all neighbors in the old protocol, and stops sending and processing messages related to the old protocol.

Our implementation of SMARTSTREAM leverages STREAMAid where each module is a class, and every module implementation can effectively contain any number of other modules. Using this feature, to facilitate algorithm switching we have implemented a new streaming module that can hold and run simultaneously several different streaming modules – effectively running several P2P live streaming algorithms in parallel. In Section 4.3 below, we experiment...
### Figure 4.3: Algorithms used for comparison. We combined various P2P algorithm components to produce a set of different compositions for experimental evaluation.

<table>
<thead>
<tr>
<th>Overlay</th>
<th>Streaming</th>
<th>Missing Chunk Handling</th>
<th>Scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Coolstreaming(M=4)</td>
<td>• Coolstreaming(M=7)</td>
<td>• Araneola(L=4)</td>
<td>• RandomMinSize4</td>
</tr>
<tr>
<td>• Coolstreaming(M=7)</td>
<td>• Araneola(L=7)</td>
<td>• Araneola(L=7)</td>
<td>• BSCAMP</td>
</tr>
<tr>
<td>• Araneola(L=7)</td>
<td>• RandomMinSize4</td>
<td>• Araneola(L=7)</td>
<td>• BSCAMP</td>
</tr>
<tr>
<td>• Coolstreaming</td>
<td>• Chainsaw</td>
<td>• Skip</td>
<td>• Pull-based</td>
</tr>
<tr>
<td>• Chainsaw</td>
<td>• Skip</td>
<td>• Wait</td>
<td>• Pull-based</td>
</tr>
<tr>
<td>• Pull-based</td>
<td>• Pull-based</td>
<td>• Push-pull based</td>
<td>• Push-pull based</td>
</tr>
</tbody>
</table>

with switching to the best algorithm to see what continuity index is achieved. We found that our approach determines the best algorithm quickly with only minor degradation in performance, and the result is robust for a wide range of $K$ values.

The methodology for switching between protocols is summarized in Figure 4.2.

### 4.3 Evaluation

Our experimental evaluation seeks to answer the following questions about SMARTSTREAM.

1. **Algorithm optimality.** Is one P2P live streaming algorithm from the literature “best” for all environments, or do different algorithms have measurably different strengths and weaknesses?

2. **Adaptation overhead.** If different algorithms should be used, how effectively can an adaptive framework transition between protocols on-the-fly?

3. **MAB performance.** Since performance of different algorithms in a given context are unknown on the forehand, how do different MAB algorithms fare when learning in a new network state?

#### 4.3.1 Experimental Setup

As a preliminary step, we run a simulation of two settings of churn (high and low) and two settings of upload bandwidth distribution (high and low). For each setting we test just below
Figure 4.4: Stream quality in different environments. The clear stream ratio and latency trade-offs of the algorithms in different system states. Push-pull algorithms prevail in both low churn and low upload bandwidth settings.

50 P2P live streaming algorithm compositions measuring clear stream percentage and average latency.

To simulate churn, we draw session lengths from a LogNormal distribution with a fixed variance and means for the high and low churn settings, respectively. The high and low upload bandwidths are simulated by drawing from a Normal distribution with means of 12.6Mbit and 2.6Mbit respectively. Each test simulates 10 minutes on a network of 200 peers using the PeerSim [MJ09] simulator with message latencies uniformly distributed between 200 and 400 msecs. Clients buffer for 2 seconds before starting playback.

We composed different algorithms by combining the predominant P2P live streaming ideas from the literature for each component of the streaming algorithm as follows. First, we use 6 different mesh-based overlays:

- **Coolstreaming4** The Coolstreaming overlay [ZLLY05] with \( M = 4 \). The \( M \) parameter controls the average degree of a node, with larger \( M \) yielding larger node degrees to increase churn resilience at the cost of higher overheads.

- **Coolstreaming7** The Coolstreaming overlay with \( M = 7 \).

- **Araneola4** The Araneola overlay [MK04] with average node degree \( L = 4 \). Similarly to \( M \), in Araneola, every node will have either \( L \) or \( L + 1 \) neighbors.
• **Araneola7** The Araneola overlay with $L = 7$.

• **RandomMinSize4** A baseline protocol which asks the tracker for 4 more neighbors when the number of neighbors of a node falls below 4.

• **BSCAMP** A version of the SCAMP [GKM03] algorithm with bidirectional links. A simple protocol that maintains $\log(#\text{nodes}) + 1$ neighbors per node on average.

For each of these overlays, we test two implementations of pull-based chunk dissemination components: Coolstreaming [ZLLY05] and Chainsaw [PKT+05].

We experiment with each of the combinations in two settings: as a stand-alone pull based algorithm, and as a part of a push-pull scheme suggested by mTreebone [WXL07] where the combination is used for the pull part as a fall-back while the peers are also organized in a different tree overlay where chunks are distributed in a push-based algorithm.

We also evaluate two settings for handling cases when a missing chunk needs to be played at a client – either by skipping the missing chunk or by waiting for it to arrive. An overview of the different algorithm compositions with which we experiment is listed in Figure 4.3.

We ran each experiment 10 times with different seeds and report the median measurements in the graphs.

### 4.3.2 Different Algorithms Perform Better in Different Settings

The first evaluation question is to compare how different algorithms perform in diverse environmental conditions. In Figure 4.4, we show the leading algorithms in terms of latency and clear stream ratio for four system states. Note that there are no pull-based algorithms in Figures 4.4(b) or 4.4(d), since they were outperformed by push-pull algorithms for all parameter settings. As can be seen in Figure 4.4, algorithms behave differently depending on the state of the system, confirming that **there is no free lunch when it comes to choosing P2P live streaming protocols**.

We note some other general observations from the experiment. In low bandwidth settings, push-pull based algorithms outperform pull-based algorithm with both lower latency and higher clear stream percentage, with the RandomMinSize overlay for the pull component in the push-pull schema leading the group. In these settings, algorithms which wait for missing chunks incur $100 - 300$ msec higher latency than those where missing chunks are skipped.

When the upload bandwidth is high while churn is low, push-pull algorithms outperform pull algorithms with minute differences between skip and wait settings. Nevertheless, under high churn, pull algorithms take the lead – trading their clear stream ratio for about 2 seconds of extra latency. In Figure 4.4(c), we see that Coolstreaming7 overlay has the larger clear stream ratio than the Araneola7 overlay. This advantage stems from Coolstreaming7 having more peers who play no chunks, thus the peers are not being counted in the clear stream ratio metric.

**Take-away:** We conclude that push-pull based combinations work better in low bandwidth or low churn settings and overall provide better latency. We see how pushing chunks on a tree overlay both improves latency and saves bandwidth. Conversely, pull-based algorithms perform better with high upload bandwidth and high churn. Also, we found that the choice of chunk
Figure 4.5: Protocol switching costs. Comparison of the continuity index for a setting where SMARTSTREAM switches between two algorithms. We change the number of rounds (K) the nodes retain state for both the current and previous algorithms.

dissemination strategy makes small to no difference on performance. Depending on the specific overlay, in 50%-70% of the cases, there was only a single possible neighbor from whom a missing chunk could be requested, making the choice of a specific pull-based algorithm largely irrelevant.

4.3.3 The Cost of Switching Algorithms

SMARTSTREAM relies on the ability to switch between algorithms underlying the stream dissemination. When a switch occurs, a special message is propagated specifying the configuration of the new algorithm and the first chunk that would be disseminated using the new algorithm. Upon receiving such message, a node starts running the algorithm and building the overlay. When a node received all chunks from the previous algorithm, it can stop running the old algorithm and disconnect, but its neighbors may lag behind. Thus, an important parameter of the switching algorithm is the time interval during which a node maintains state of the old algorithm after making the switch to the new one: K.

If K is set too small, many peers will be disconnected and the overall performance will be degraded. Conversely, a high value of K can congest the network. As we can see in Figure 4.5, K = 20 is the sweet spot in our settings. For K = 30, the continuity index never fully recovers from the changes.

4.3.4 Best Algorithm for Environment Conditions

We next subject SMARTSTREAM to a dynamic environment to assess the adaptivity of its protocol choices. After an initial training period, SMARTSTREAM is able to learn which algorithm best suits what environment state. In Figure 4.6, we test the performance of the algorithms in a dynamic environment. In this experiment, the churn rate is dialed high and the mean upload bandwidth setting of new peers is kept at 12.6Mbit. Starting at the 600 second
Figure 4.6: **Continuity of adaptive vs. static approaches under dynamic bandwidth.** Comparison of the continuity index of SMARTSTREAM versus each of the leading algorithms of every state. The upload bandwidth of peers is 12.6Mbit until the 600 second mark and new joining peers have 2.6Mbit upload bandwidth afterwards. The churn is set to high. Each point in the graph is a weighted average of the continuity index over the preceding 100 seconds.

Figure 4.7: **Continuity of adaptive vs. static approaches under dynamic bandwidth.** Comparison of the continuity index of each of the leading algorithms of every state with SMARTSTREAM’s performance. The upload bandwidth of new joining peers is 12.6Mbit for until the 600 second mark, then 2.6Mbit for 1200 seconds, then again 12.6Mbit between 1800 and 2400 second marks and again 2.6Mbit for the last 1200 seconds. The churn is set to high. Each point in the graph is a weighted average of the preceding 100 seconds.

mark, peers who join the overlay have their mean upload bandwidth set to only 2.6Mbit, thus gradually changing the state of the environment. We compare the continuity index of each of the algorithms that performed best on each state against our SMARTSTREAM. From the figure, it is evident that SMARTSTREAM’s performance follows the leading algorithm when the state changes without incurring significant switching costs.

Further, in Figure 4.7, we examine several state changes. The initial state consists of both high churn rates and high upload bandwidth, then toggling the configuration to high churn rate with low upload bandwidth. The toggling is repeated two times in succession. We observe that the push-pull based algorithm maintains a stable continuity index throughout the run, whereas the pull-based algorithm performs better in the high upload bandwidth settings and then struggles under low upload bandwidth. Again, SMARTSTREAM tracks the better algorithm with minor impact on the continuity index during changes of the dissemination algorithm.
4.3.5 MAB Reward Function

In order for SMARTSTREAM to optimally choose an algorithm, we must estimate the current continuity index in an online fashion. In the next experiment, we run several different compositions of P2P live streaming algorithms that were described in Subsection 4.3.1. For each algorithm, we record the three performance metrics (CR, CRA, CI) during runtime as data to help approximate the true continuity index. We additionally recorded the true continuity index for each segment of the experiment for comparison. We used the Weka library [HFH+09] to train a linear regression classifier, and ran 10-fold cross-validation to compare the results.

Figure 4.8 shows that the local continuity index is the largest predictor of the global continuity index. Using all three metrics further improves the accuracy.

4.3.6 Convergence of MAB algorithms

We have implemented UCB1 and $\varepsilon$-Greedy MAB algorithms in SMARTSTREAM. We next experiment with their ability to choose the best algorithm in a high churn rate but low upload bandwidth setting, where push-pull algorithms have the upper hand. Each test simulated a 2400 second stream, and the test was repeated 5 times. The MAB algorithm was invoked every 100 seconds. Each algorithm was run twice, once when the leading algorithm was the default choice and once when the second-strongest algorithm was default, denoted with R. The different parameters of the $\varepsilon$-Greedy MAB algorithms were chosen after tuning.

We utilized three different $\varepsilon$ functions:

- **EG-C** - maintain a constant value of $\varepsilon$.

- **EG-L** - $\varepsilon$ decays in a linear fashion: $\varepsilon = 1 - Lt$ where $t$ is the number of rounds that have passed.
Figure 4.9: Adaptivity. Continuity index of SMARTSTREAM working with different multi-armed bandit algorithms while adapting to new environmental configuration. Each subfigure shows the two static algorithms in the high churn low upload bandwidth setting and one type of MAB function with the default choice being either the leading algorithm or the second-strongest.

- **EG-IL** - $\varepsilon$ decays in an inverse linear fashion: $\varepsilon = \frac{1}{t}$ where $t$ is the number of rounds that have passed.

Figure 4.9 shows that the constant and linear functions of the $\varepsilon$-Greedy MAB algorithm mostly stick to the default algorithm, but eventually switch. In Figure 4.9(c) we see how EG-IL chooses the most competitive streaming algorithm faster than other $\varepsilon$-Greedy MAB algorithms. For UCB1, however, the difference in rewards is too small (Figure 4.9(d)), causing the framework to keep switching between the different algorithms.

**Take-away:** MAB algorithms can efficiently choose the best algorithm, but need to be carefully chosen and tuned.

4.4 Related Work

A vast number of P2P live streaming algorithms have been suggested in the academic literature: a recent survey [ZH12a] outlines more than 30 different takes on the problem. Several surveys [ZH12a, MRG07, HLR08] classify P2P live streaming algorithms by their chunk exchange algorithm types. On the one hand, pull-based systems exchange chunk availability bitmaps and request chunks from overlay neighbors. On the other, push-based systems forward received chunks to all other neighbors usually relying on a tree (or multi-tree)-shaped overlay or other
techniques to minimize transmission of duplicate chunks. Notable examples of pull-based algorithms include: Coolstreaming [ZLLY05], Chainsaw [PKT’05] and PULSE [PPKB07], while multi-tree push-based systems include SplitStream [CDK+03], ChunkySpread [VYF06] and BitTorrent Live [Coh12]. Hybrid approaches such as mTreebone [WXL07], combine push and pull-based systems to reap benefits from both types of algorithms.

Liang et al. [LGL09] implement several P2P live streaming designs and compare them in various settings of peer upload bandwidth, server upload bandwidth and buffering delay. They conclude that when the resource index is high, i.e., the peers and server both have high upload bandwidth, even the simplest random algorithm works well enough. When the upload bandwidth is scarce or restricted, however, more sophisticated algorithms outperform the naïve approaches.

The authors of TRANSIT [WRRH14] propose a system where each peer can decide whether it operates in pull or push mode by setting the length of requested flows from neighbors. Shorter flows resemble a pull-based system and longer flows resemble a push based system. In SMARTSTREAM, all peers can gradually switch to a different algorithm either to try a new algorithm if the measured state has changed and better performance can be achieved by using another algorithm.

Self-adaptive systems have been discussed extensively within various engineering disciplines, whereas adaptive software systems are a more recent development. There is significant understanding of software systems that follow a single control loop, usually from the lens of control theory, whereas adaptivity of systems comprised of multiple control loops – as in the case of P2P live streaming systems – are less generally understood [DLGM+13].

4.5 Discussion and Conclusion

The accelerating growth in video traffic on the Internet highlights the need for effective mechanisms for disseminating live streams. Our work is driven by the question of whether P2P live streaming can be made to adapt to a changing environment, addressing problems that can arise during large-scale deployment.

We proposed a framework, SMARTSTREAM, in which the dissemination algorithms are gradually switched to ones that appear to have better performance on important metrics for the live stream quality. The performance is measured by a function over desirable metrics, such as maintaining a high continuity index and low latency. We began by showing that different algorithms prevail in different environments, justifying why switching distribution algorithms should even be considered. We took a centralized approach to dynamically choose between dissemination algorithms and show that if the rough break-down of challenging environmental settings is known ahead of time, SMARTSTREAM is able to closely match the performance of the best algorithm without incurring a high switch cost.

To find the most suitable algorithm for each environmental state, we proposed to use multi-armed bandit (MAB) algorithms to navigate the exploration versus exploitation trade-off when
choosing the next candidate protocol. MAB algorithms rely on a well-defined reward for every time period following a choice. We studied different reward functions and derive the function that ideally reflects the true stream continuity index. We then showed that the MAB algorithms we studied converge quickly to a good streaming protocol. In SMARTSTREAM, MAB algorithms are used quite generally without any dependence on the specific problem domain. This way, new P2P live streaming algorithms, metrics and components can be implemented and added to the framework without changing the MAB logic.

We then analyzed the required building blocks for an adaptive P2P live streaming system. Our approach relies on several basic solutions for simplicity – more sophisticated components may yield even better results. For example, genetic algorithms and other forms of local search may be able to identify the best P2P streaming algorithm for each environment state, and even define the challenging states. Further, contextual MAB algorithms could potentially accelerate the convergence speed. We defer these improvements to future work.

In conclusion, we have showed that switching between P2P live streaming algorithms on-the-fly is not only achievable but appropriate in many settings, with the performance benefits outweighing minor degradation during a switch. Further, we believe the learning theory approach we took to the P2P live streaming problem generalizes to other large-scale distributed systems, suggesting that learning theory principles could be incorporated as part of the core to allow the systems to become more adaptive and responsive to their environments.
Chapter 5

An Advertising Mechanism for P2P Networks

In this chapter we set to study economic advertisement mechanisms. Specifically, we assume a model in which advertisements (ads) are spread inside an existing non-dedicated P2P network. Whenever a user accesses a certain content, the peer that served the content will also integrate one or more advertisements from the set of ads it currently has. The advertiser is then notified, and in return pays both to the peer that served the ad and to all other peers that participated in the dissemination of the ad.

Realizing this concept entails several challenges. For example, the dissemination mechanism should disseminate the ads to parts of the network where they are likely to be useful, without knowing who the users that view these ads are. Further, there is a need for a payment model that would encourage the dissemination of ads as well as peer participation in their dissemination. In particular, the payment obtained by each peer should be proportional to its participation time in the P2P network, and ideally, should be super-linear. That is, consistently active peers should be paid much more than sporadically active ones.

The ad dissemination mechanism should work as an incentive mechanism for an existing P2P system. Hence, it cannot modify the P2P overlay for its needs (similar to the work done in [NCT05]), as this may hurt the performance of the real application running on top of the P2P system. Instead, it must rely on the existing P2P overlay and do the best it can given this overlay.

We present several such approaches to ad dissemination, part of which are random and others are based on machine learning techniques, and investigate their performance in combination with several payment models. We also develop a light weight propagation encoding scheme that prevents peers from cheating about their participation in ad forwarding paths. We report on the insights gained from our study regarding when it is preferable to use which techniques and how the payment models affect the dissemination. Further, our results indicate that indeed P2P advertisement schemes are a promising direction for building incentives mechanisms in P2P networks, as elaborated on in this chapter.\footnote{This research was part of the e-Wolf project, e-Wolf is a P2P social network with privacy-preserving as one of its main objectives, whose prototype is available in open source at https://github.com/gilga1983/ewolf.}
The contributions of this work are as follows:

- A new model of economic incentives in a P2P network.
- A light weight propagation encoding scheme that prevents peers from cheating about their participation in ad forwarding paths.
- An algorithm to find seeding peers in a graph (elaborated further)
- Insights regarding when it is preferable to use which dissemination techniques and how the payment models affect the dissemination.

This chapter is based on the publication [FL13].

The rest of this chapter is organized as follows: An overview of our solution appears in Section 5.1, while Section 5.2 discusses the various payments models and dissemination strategies. The dissemination tracking scheme is introduced in Section 5.3, and the simulation performance results are presented in Section 5.4. Related work is discussed in Section 5.5. Finally, we conclude with a discussion and future work in Section 5.6.

5.1 Overview

5.1.1 Goals

As indicated in the Introduction, the mechanism we are looking for should be effective both as an incentive mechanism for the network’s peers as well as attractive for the advertisers. As an incentive mechanism, it should reward the owners of donated peers proportionally to the availability of these peers. Ideally, the reward should be super-linear with the availability in order to encourage keeping a donated machine connected to the network for long periods of time. At the same time, the communication overhead imposed by the protocols should be kept low and they should be computationally and space efficient, in order not to interfere with the main network activities.

As for the advertisers, the mechanism should ensure that every ad reaches every peer that currently serves a potential target user for the ad. In other words, the mechanism should obtain extremely high reach rates. At the same time, the mechanism should ensure that the maximal advertiser’s budget is not exceeded. Also, the mechanism should account for rogue nodes that may try to attack the system or try to increase their revenue through fraud.

5.1.2 Basic Concepts

In our work, we assume the existence of a P2P network of nodes. The nodes of the network have neighbors and can send messages to any other node in the network. All nodes can have client devices connected to them. A client device provides a single user with access to the network through a sole connection to a node of the P2P network.

Each user has specific characteristics such as: age, gender, marital status etc. An advertiser that is interested in disseminating advertisements on the P2P network must provide machines to
function as *advertiser nodes*. Each advertiser node is in charge of disseminating to its portion of the network. Depending on the size of the network, an advertiser must ensure that the advertiser nodes can handle their allocated network portion sizes. The advertiser nodes receive *advertisements* (with bid, target audience, budget and such) from advertisers. An advertiser node stores each advertisement until the budget allocated for the advertisement runs out. The advertiser nodes are in charge of disseminating the advertisements in their portion of the P2P network, tracking impressions and clicks as well as sending payment notices to nodes that participated in the dissemination. The advertiser node disseminates a small message, called *ADM* - advertisement dissemination message, that describes the features of the advertisement (target audience, bid, budget etc.). When a node decides that the advertisement is suitable for one of its clients (by asking the client), the node requests the actual content of the advertisement from the advertiser node. Also, when receiving an ADM, the receiving node decides which of the neighbors it will forward the ADM to. When a user views or clicks on the advertisement, other nodes that contributed to its dissemination and serving are getting paid.

### 5.1.3 Advertiser node

The advertiser node needs to disseminate the ad to all nodes in its portion of the P2P network that are likely to post it to their clients. Call these nodes the *target nodes*. However, the advertiser node may not know a-priori who are the target nodes, and even if it does, for scalability reasons it may not be practical for the advertiser node to contact each of them directly. Rather, we are looking for a solution in which the advertiser would contact only a small set of peers, which we call the *seeding peers*. These peers initiate the dissemination process to the rest of the network.
As requests for ads and click reports flow back into the advertiser node, it can gradually learn about target nodes as well as growing parts of the P2P overlay in its portion of the P2P network, and collect statistics. In particular, all our schemes are based on rounds where each round comprises of disseminating an ad from the advertiser node and then obtaining feedback about its reach, impressions, and clicks. The advertiser node maintains a round based history of seeding peers and for each advertisement a score is being kept, which comprises of the click through rate (CTR), number of clicks-per-day, etc. Using the gathered data along with the advertisement properties, an advertiser node can choose seeding peers for new advertisements. Using the principle of time locality, the advertiser peer can assume that (most) target nodes of recent ad dissemination rounds remain target nodes for the following rounds. Hence, intuitively, the advertiser nodes picks as seeding peers a small number of nodes whose distances to the target nodes are minimal. From these seeding peers, the ads will be propagated on the P2P overlay and will likely meet their target nodes. Yet, since the advertiser node may not be aware of all target nodes, the dissemination process described in the next section enables discovery of additional target nodes and is not limited to the already known ones. Recall that we try to optimize the dissemination on the existing P2P overlay because in our model, the advertisement mechanism is part of an existing P2P system and thus cannot alter the P2P overlay.

The availability of the seeding peers is also recorded and is factored in the score to give preference to nodes that are available the most. For each impression, the node that serves it sends a content request message to the advertiser node. Whenever a user clicks on an ad, the user is taken to the advertiser node that redirects him to the advertiser’s page. Since all clicks go through the advertiser node, the advertiser node can employ click fraud detection mechanisms to filter fraudulent clicks [IJMT05]. If the clicks-per-day for the advertisement is too low, the advertiser node may choose different seeding peers and send the advertisement again, or, notify the advertiser. If the ad budget is nearly depleted, all content requests return with a flag that notifies the asking nodes that future requests will be ignored and not be paid for.

Below, we elaborate on the exact details of these mechanisms.

**Seeding peer selection**

As mentioned before, an advertiser node records all requests for advertisements. In order for an advertiser node to select seeding peers, it creates a graph depicting the latest state of its P2P overlay portion known to it along with the target nodes that have served advertisements recently. From this graph, the advertiser node selects $s$ seeding peers, where $s$ is a tunable parameter. Nodes that were chosen as seeding peers, but were unavailable are deleted from the graph for a period of time reverse proportional to their overall availability (i.e. a node that is available $1/51$ rounds will have $x50$ more penalty time than a node that is available $50/51$ rounds). Based on the resulting graph, the advertiser node chooses the seeding peers as the ones minimizing some cost function related to reaching all target nodes. We consider the following optimization function for seeding peer selection: Min sum of distances - select the seeding peers such that the sum of distances from each target node to the closest seeding peer would be minimal.
The problem of finding minimal sum of distances is somewhat similar to \( K \)-Medoids [KRTHI87]. However, the \( K \)-Medoids algorithm would minimize the distance between all the nodes to \( k \) corresponding seeding peers, while we want to minimize only distances from target nodes to their corresponding seeding peers. We have implemented a variation of the \( K \)-Medoids algorithm described in [PJ09] to accommodate these differences, as listed below. On each round, every advertiser node uses the modified algorithm with several \( k \)s and chooses the \( k \) that resulted in the lowest score; the seeding peers of this advertisement node for that particular round are the resulting seeding ones computed with the chosen \( k \). In order to account for the communication costs of nodes, in the algorithm below we assign a high cost for messages sent between the advertiser node and the seeding peers.

Suppose that there are \( n \) nodes in the graph of the partial P2P overlay. Let \( d_{ij} \) be the distance between every two nodes \( i \) and \( j \). \( d_{ij} \) is the amount of hops it takes in the graph (or P2P overlay) to reach from \( i \) to \( j \) (or vice versa). Suppose that, without loss of generality, the first \( s \) nodes are target nodes. The new algorithm is composed of the following three steps:

1. Step 1: (Select initial seeding peers)
   
   (a) Calculate the distance between every pair of all nodes.
   
   (b) Calculate \( v_j \) for node \( j \) as follows:
   
   \[
   v_j = \sum_{i=1}^{n} \frac{d_{ij}}{\sum_{l=1}^{s} d_{il}}, \quad j = 1, \ldots, n
   \]
   
   (c) Sort \( v_j \)’s in ascending order. Select \( k \) nodes having the first \( k \) smallest values as initial seeding peers.
   
   (d) Obtain the initial cluster result by assigning each target node to the nearest seeding peer.
   
   (e) Calculate the sum of distances from all target nodes to their assigned seeding peers.

2. Step 2: (Update seeding peers)
   
   (a) Find a new seeding peer of each cluster, which is the node minimizing the total distance to target nodes in its cluster.
   
   (b) Update the current seeding peer in each cluster by replacing with the new seeding peer.

3. Step 3: (Assign target nodes to seeding peers)
   
   (a) Assign each target node to the nearest seeding peer and obtain the cluster result.
   
   (b) Calculate the sum of distance from all target nodes to their seeding peers. If the sum is equal to the previous one, then stop the algorithm. Otherwise, go back to the Step 2.
5.1.4 Advertisement serving

For each ADM that reaches a node with clients, the node asks each client if the advertisement is suitable for the client. The node requests the advertisement from the corresponding advertiser node if at least one client is suitable for the advertisement (i.e. the client is part of the target audience of the advertisement). However, the node periodically removes the lowest bidding advertisements to save space.

Each time the user pulls new content from the P2P network, advertisements with the highest bid are chosen to be shown to the user and the respective advertiser nodes are notified of the impression. It is possible to develop a mechanism that adds a relevancy factor to each advertisement, thus showing the most relevant advertisement rather than the most valuable. The relevancy can be calculated using the social information the user shares with the P2P based social network, such as, interests, viewed content, etc..

5.2 Design Decisions

5.2.1 Payment Models

When a user views or clicks on an advertisement, the advertiser node is being notified. We employ the well known GSP mechanism [EOS05] to calculate the amount that the advertiser is paying for the impression or click. All nodes participating in the dissemination should get a part of that amount. We define four different payment models:

**Equal Share** - we divide the amount equally among all participating nodes. Since each participating node performed the same amount of work (send a message to a neighbor) all nodes should get equal share of the reward.

**Equal Referral Share** - the last node gets half of the amount, and the rest is divided equally between the referring nodes. Using this model, the amount that the serving node is receiving is not affected by the dissemination path length.

**Balloon challenge** [TCG11] - the last node gets half of the amount, and each node before it on the dissemination path gets half of what the next node got. Using this model, the proximity to the serving node is rewarded.

**Bounded share** [BDOZ11] - we choose an upper bound N and pay the amount/N to the first N-1 nodes on the dissemination path. The rest of the amount goes to the last node. Using this model, an advertiser can limit the length of the dissemination path and also each node can know exactly the amount it would get for a referral.

For each of these payment models we can invoke different payment schemes:

**Pay for all** - we can pay for every impression. Since every impression is profitable for the advertiser, all nodes participating in the dissemination are getting paid for every impression.
Pay per useful neighbor - all nodes but the serving node sent a single message regardless of the amount of impressions that the advertisement generated. To reflect this, in this scheme we pay the disseminating nodes once for each useful neighbor. A useful neighbor of a node is a neighbor that received an ADM that eventually produced an impression (either by its own client or by disseminating the ADM further). For each advertisement, every node is paid only once per useful neighbor. The serving node is still paid for every impression since it sends messages for every impression. The amount paid for a useful neighbor can be according to the first impression that this neighbor has produced.

Pay max revenue per useful neighbor - another option is to pay for the first impression, but if another impression would have paid more for the same neighbor, then the node gets paid for the difference. In this scheme, each node gets paid for one impression - the one with the maximal revenue. This scheme employs the same reasoning as the one before, but does not take into account the order in which the impressions occur.

The same schemes can be applied when paying for clicks. Pay per useful neighbor scheme depends on the order in which the impression happen and thus, inserts more uncertainty to the system. Due to lack of space, in the scope of this work, we look into Equal Share, Equal Referral Share and Balloon Challenge models with Pay for all and Pay max revenue per useful neighbor schemes. As mentioned above, it seems that Pay per useful neighbor would not add much over Pay max revenue per useful neighbor, but is much more sensitive and is thus not investigated further. The Bounded share model has an extra tuning parameter. Thus, we have left its exploration and comparison to future work.

To get a feel for the different combinations of payment models and payment schemes, consider the scenario depicted in Figure 5.2 in which a seeding peer has disseminated an ADM with a total payment of 1$, which has reached 3 target nodes - A, B and C, each with a single connected client. Suppose that each client has generated 10 impressions. The different models and schemes will generate the following revenues:

• With the Pay for all payment scheme:
– Equal Share - the seeding peer would receive $10 \times 1/2$ for target node C, $10 \times 1/3$ for target node A and $10 \times 1/4$ for target node B.

– Equal Referral Share - the seeding peer would receive $10 \times 1/2$ for target node C, $10 \times 1/4$ for target node A and $10 \times 1/6$ for target node B.

– Balloon challenge - the seeding peer would receive $10 \times 1/4$ for target node C, $10 \times 1/8$ for target node A and $10 \times 1/16$ for target node B.

• With the Pay max revenue per useful neighbor payment scheme:

– Equal Share - the seeding peer would receive $1/2$ for target node C and $1/3$ for target node A. No payment would be received for node B, since target nodes A and B were reached by the same useful neighbor, and the pay for A is higher than the pay for B.

– Equal Referral Share - the seeding peer would receive $1/2$ for target node C and $1/4$ for target node A.

– Balloon challenge - the seeding peer would receive $1/4$ for target node C and $1/8$ for target node A.

### 5.2.2 Dissemination Strategies

Whenever a node receives a new ADM, the receiving node decides to which of the neighbors it will forward the ADM. Lacking prior studies on dissemination strategies for our setting, we first consider the following simple strategies:

**Flood** - a trivial solution is whenever a node receives a new ADM, it simply propagates it to all neighboring nodes in the P2P network. Flooding is very expensive, but guarantees maximal possible reach.

**Random with parameter** $r$ - another option is to send the messages to part of the neighbors probabilistically. For each neighbor, the probability to forward an ADM is $r$.

The above strategies treat all neighbors the same. They are very simple and mostly serve as a reference to compare with more sophisticated ones. In particular, learning strategies could inspect past ADMS passed to a specific neighbor and decide based on the revenue of those advertisements if the current ADM should be passed to that neighbor. If there is not enough history for a specific neighbor, the advertisement is passed to that neighbor to gather information. Only recent history is considered when making the decision whether to pass the ADM. This is to make sure that only the latest network state is taken into account. We propose three learning strategies:

**Recent income** - a simple strategy that passes an ADM to a neighbor if a recent ADM passed to that neighbor has yielded any revenue. However, This strategy treats all ADMS equally and does not take the characteristics of each advertisement into account.
**AdPrice with parameter** $k$ - a strategy that for every neighbor $p$ employs machine learning techniques on recent advertisements sent to $p$ to predict the revenue for the current ADM. The revenue prediction is then compared to the average revenue of recent ADMs sent to $p$. The ADM is passed if the revenue prediction divided by the average revenue is higher than $k$.

**Probabilistic AdPrice with parameter** $prob$ - a strategy that for every neighbor $p$ employs machine learning techniques on recent advertisements sent to $p$ to predict the revenue for the current ADM. The ADM is passed with probability of $prob + \text{revenue prediction divided by average revenue of recent ADMs sent to } p$.

As an example, suppose a node $p$ has to decide whether to pass an ADM to a neighbor $q$. The node $p$ predicts a revenue of $rev$ if the ADM is passed to $q$. The average revenue for $p$ of advertisements recently passed to $q$ is $\text{avgRev}$. The different strategies will operate as follows:

- **Flood** - will pass the ADM.
- **Random with parameter** $r$ - will pass the ADM with probability $r$.
- **Recent income** - will pass the ADM if $\text{avgRev} > 0$.
- **AdPrice with parameter** $k$ - will pass the ADM if $\text{rev/avgRev} > k$.
- **Probabilistic AdPrice with parameter** $prob$ - will pass the ADM with probability $(\text{rev/avgRev}) + prob$.

### 5.3 Dissemination tracking

#### 5.3.1 The Mechanism

When a node receives an ADM, the ADM should include the route this ADM has traveled. This information is needed for the advertiser so he can pay all participating nodes. However, if we simply pass this information as is, then any node on the route can delete all preceding nodes before it from this info, thus getting more credit for all the clicks and impressions of that advertisement further down the line. Hence, a mechanism is needed that will guarantee that no node would be able to alter the route of the ADM.

To that end, the advertiser node includes in the ADM an array, called *path array*, with a length of $k$. In each cell of the array, the advertiser node generates a random one time pad (OTP) that will be used to encrypt a node ID, an index of a cell, and a bit specifying whether this cell is already used. The advertiser marks $m < k$ cells as used and sends the message to the seeding peers specifying an index of a free cell to use (the advertiser keeps record of this as well). Upon receiving the ADM and an index of the cell to use, each node $P$ does the following:

1. Chooses randomly a free cell from the array in the ADM for the next node to use.
2. Encrypts the ID of $P$ and the chosen cell for the next node in the cell received from the sending node (by XORing with the OTP in that cell).

3. Marks that cell as used.

4. Chooses neighbors to send the ADM to, and sends them the ADM specifying the unoccupied path array cell chosen for them.

When requesting ad content to present to the user, the node also sends the path array to the advertiser node. The advertiser node decrypts all occupied cells in the path array that the advertiser node did not mark as occupied.

The cells in the path array are linked so that the advertiser would know the order of the dissemination. Some payment models (Balloon challenge) may require the exact order of the node. Also, when a node is paid for an advertisement, it should know the successor that is responsible for that payment in order to evaluate which nodes are good candidates to disseminate the next advertisement.

### 5.3.2 Security Analysis

A malicious node may take on different roles to try and gain profit using the advertisement mechanism.

**Imposing as an advertiser node**

All messages sent by legitimate advertiser nodes can be signed by the advertisers making it unfeasible to impose as an advertiser node. We assume that there cannot be a malicious advertiser.

**A malicious referring node**

When receiving an ADM, a node cannot know in which cell its predecessor stored its info in since all the cells containing the path information are encrypted. Also, the node does not know the order between its predecessors. Since the advertiser marked $m$ cells as used, the $p$th ($p > 1$) node will have a $1/(p + m - 1)$ chance of guessing the position of its predecessor (or any other specific cell).
If a node receives and ADM that the node has already participated in its dissemination, the node can learn the path that ADM has travelled since it has sent it. But the node cannot use that information for any malicious purpose.

When receiving an ADM, a malicious node can alter the contents of an occupied cell in the path array. In this case, there is a slight chance that the altered cell will hold a different legitimate ID; otherwise, the altered cell will hold non-existent ID. If an advertiser node receives a path array with non-existent ID, it can ask all the participating nodes to send the path array and the ID of the node that they had sent the ADM to. That way, the advertiser node will find out the missing ID and also will narrow down the nodes that are suspicious of malicious behavior to two - the first node that reported the array with an altered cell and the node before it. Both nodes can be notified of this so that the legitimate node could punish the malicious one (for example by not forwarding ADMs to it). To handle the former case (altered cell holds legitimate ID), an advertiser node may once in a while ask the participating nodes to send the path array and the ID of the node that they had sent the ADM to, even if the path array holds only legitimate IDs.

If 2 nodes (A and B) are working together to cheat the system, every time A receives an advertisement, A can write B also in the array. There is a chance that B is already in the array. In that case, the advertiser can again ask all the participating nodes to send the path array and the ID of the node that they had sent the ADM to, and, narrow down the nodes that are suspicious of malicious behavior to two. Otherwise, (if B is not present) the advertiser node can not tell that cheating has occurred. To prevent this scenario, the path array can be altered to hold ID of node that the ADM was received from, ID of current node and ID of node that the ADM is sent to. That way, a node can not add any other IDs before or after it in the path array. Another option is for A to check if B is already in the array. This communication would require A to send a message to B and wait for a response, while sending the ADM to B would potentially have the same effect and use only one message. In other words, this type of cheating costs more than following the rules.

A malicious serving node

A malicious serving node can request for advertisements even though there are no users interested in the advertisement (or no users at all) connected to the node. It that case, the advertiser node should notice that the CTR of the node is marginally lower than the CTR of other nodes and suspect the node of malicious requests. Suspected Nodes can be punished (for instance, have all their requests rejected for some time). As mentioned before, all clicks also go through the advertiser node, so, fraudulent clicks can also be detected.

5.4 Performance Evaluation

5.4.1 Model and setup

The simulation of the different elements is divided into dissemination rounds. Each round starts with the advertiser nodes sending new ADMs to the respective seeding peers. The bids
for the ADMs are generated using information from [Web]. During the round, the ADMs are being disseminated and impressions are being simulated (a constant amount of impressions for the 3 highest bidding advertisements for every online user). When the round ends, the advertising peers send a payment notice to all nodes that are entitled to any payment specifying the advertisement for which the node is being paid. If the payment is for a referral, the referred node is also specified.

In order to model churn, every node receives on start up a number of consecutive rounds that the node should be responsive for until it fails, called session length. These lengths are assigned to the nodes randomly from a Weibull distributed variable, which was reported to represent well real P2P networks’ churn [SR06, SENB09], with parameter $k = 0.5$ and a specific mean. Further, in order to avoid a situation in which all nodes are up on the first round, every node also receives on start up the first round in which it should fail. After a round in which a node was in a failed state, the node rejoins the network as a brand new node with new users, no history and the same session length.

The various dissemination strategies, payment models and the tracking mechanism are fully implemented in Java and the source code is available at 2. Only the messages between the peers are simulated using the PeerSim simulator3. We simulate a P2P network with 512 nodes and the network is wired as a hypercube, which functions as a portion of a P2P network and thus has only one advertiser node per advertiser. 5 such nodes act as advertiser nodes for target audiences of various sizes. The history length of the learning strategies is set to 20 rounds.

5.4.2 Definitions

- **Round reach rate** is defined to be the ratio of users receiving an advertisement out of all users that are the target audience for that advertisement for a specific round.

- **Round miss rate** is defined to be the ratio of users not receiving an advertisement out of all users that are the target audience for that advertisement for a specific round.

- **Average reach rate** is the average round reach rate.

- **Average miss rate** is the average round miss rate.

- **History length** - the amount of latest rounds the learning strategies take into account when making predictions.

5.4.3 Results

**Random $r$ parameter tuning** We have tested the random strategy with different $r$ parameters. We have used the Balloon Challenge payment model with pay for all impressions payment scheme. As we can see from Figure 5.4, we get low ($< 0.05$) round miss rate starting from $r = 0.4$. There is a steady increase in the amount of advertisements sent with the increase of

---

2https://sourceforge.net/projects/adflow/
3http://peersim.sourceforge.net/
Figure 5.4: Overall miss rate, ads sent, revenue and revenue per ad as a function of the value of the r parameter of the random strategy

Also, we note a decrease in the revenue per ad with the increase of r. The total revenue is increasing with the reach up to r = 0.3. Then, when the reach is high (> 0.9), the total revenue decreases. This happens because the Pay for all payment scheme pays for referrals proportionately to the path lengths.

AdPrice p parameter tuning For all values of p < 1 that we have tested, the behavior is nearly identical; it can be seen as the second set of graphs (vertically) in Figure 5.5 (AdPriceS-trategyp0.0). That is, initially the miss rate is high and the message cost is relatively low. Yet, as time goes by and the learning improves, the miss rate drops to almost 0 but the message cost increases. When p >= 1, the round miss rate becomes much higher than with any of the other schemes we tested, more than 0.2 (graph omitted for brevity).

Average miss rate versus messages sent In Figure 5.5, we analyze closely the behavior of the different dissemination strategies. The payment model is Balloon Challenge with pay for all impressions payment scheme. The Random strategies send approximately the same amount of messages for every advertisement. They also maintain the same round miss rate with little variance from the average. The behavior of Recent income and AdPrice strategies is cyclic. Every cycle the average amount of messages sent per advertisement increases and the round miss rate decreases. The size of the cycles is exactly the history length of each strategy. However, the amount of messages sent per advertisement increase logarithmically. As shown in Figure 5.5, for every round miss rate, there are random and learning strategies that can eventually provide
that round miss rate. Yet, in doing so, the learning strategies eventually send fewer messages per advertisement than the random ones while providing the same round miss rate.

**Impact of churn** We have tested the AdPrice0.0 and Random0.6 strategies with different churn settings. We have ran tests with session length means of 75, 150 and 300 rounds. Figure 5.6 shows that both strategies maintain their average miss rate behavior. However, when increasing the session length mean (decreasing the churn), the random strategy sends out more messages per advertisement while the AdPrice strategy sends the same amount.

**Impact of payment model** We have tested the randomStrategy0.6 with different payment models. In all payment modes, the average miss rate and the amount of messages sent per advertisement are similar to Balloon Challenge pay model with pay for all payment scheme shown in Figure 5.5. In Figure 5.7, we present an exponential fit to the revenue results. The variance of the exponential fit is very low (0.00001-0.02) and is lower than fitting to a linear plot (which is in turn lower than fitting to a logarithmic plot). We can observe the desired result of nodes receiving payment proportionally to their availability. In particular, methods whose
As can be further seen in Figure 5.7, payment models that pay for all impressions pay more for availability and their base of the exponent is higher. Equal reference and Balloon challenge models behave in a similar manner. Both models pay half of the amount to the serving node. When paying for all impressions, Equal reference model pays a little more since Balloon challenge model does not distribute all the amount between the disseminating nodes \( \frac{1}{2^{\text{disseminationLength}}} \) of the pay amount is always left undistributed). However, when paying only for maximal impression, Balloon challenge model pays more for availability than Equal reference model. Using that scheme, Equal reference model pays every node for the shortest dissemination path that the node participated in, while Balloon challenge pays for the closest target node. Equal share model pays marginally more for availability than both Balloon challenge and Equal reference models in both payment schemes. When employing the AdPrice strategy, the payments per availability are a bit lower than those in Figure 5.7 due to the time it takes for the strategy to reach a comparable round miss rate. Further, the variance of the fit is larger (0.000005-0.03). This exponential behavior is enforced by the seeding peers that take into account nodes that are unavailable when choosing potential seeding peers.
5.5 Related Work

Several P2P networks with social aspects have been introduced, e.g. [BSVD09, PGW+08]. However, their work have not focused on advertisements as a possible incentive mechanism.

In [DS11] and the followup work [SD12], the authors disseminate messages to target audience. However, the dissemination is carried out over the social links rather than the P2P links. Also, in order to disseminate, the users share information with their immediate social neighbors. Lastly, the authors of these works did not simulate churn.

In [CSXZ09], the authors discuss instant and location-aware commercials. The paper presents an opportunistic gossiping model for disseminating instant advertisements, including a probability function for determining advertisement forwarding probability at different locations. In their work, they were able to provide high delivery rate of advertisements while keeping the delivery time and the number of messages low. However, [CSXZ09] addresses dissemination between a network of mobile devices. Hence, they used distance and velocity information to optimize the gossiping model, information that cannot be used to optimize our P2P dissemination strategies.

[FM08] introduces a new social gossip protocol and its analysis. As a recommendation travels from one user to the next, its relevance decreases. Therefore, when a certain hop-count limit is reached, the relevance goes to zero and the message dissemination stops. The
adoption criterion of accepting only \( f + 1 \) disjoint gossip paths protects the network from spam recommendations. The main contribution is a practical path verification protocol whose computation and storage complexities are polynomial in \( n \). In our work, we assume that no spam advertisements are passed (they could be signed by the advertisers). So, we accept the first ad received as genuine.

The goal of [RBCS10] is to incentivate sharing and discourage free-riding in P2P networks. However, their model is designed to run over any type of graph structure that can be sub-grouped and managed by a super-peer, which is not the case in our work.

A P2P publish/subscribe technique called Pub-2-Sub [TP09] can be used to disseminate advertisements in a P2P network. Pub-2-Sub assigns to each node a unique binary string called a virtual address so that the virtual addresses of all the nodes form a prefix tree. Based on this tree, each node is assigned a unique zone partitioned from the universe of binary strings. Then, later queries and publications are hashed to binary strings and, based on their overlapping with the node zones, subscription and notification paths are chosen appropriately and deterministically. Unfortunately, Pub-2-Sub can work well only when the network is stable and cooperative, since any churn (peers leaving and joining) triggers a restructure of parts of the prefix tree. Also, rogue nodes (if placed high enough in the prefix tree) can affect many peers.

A well conceived non economical incentive mechanism for message relaying of service requests in a P2P network is described in [LYS09]. In that mechanism, promised rewards are passed along the message propagation process and after a service provider was reached - a rewarding process is propagated backwards on the same route. However, in our model, the advertiser disseminates the advertisements and is not a service provider whom the users are trying to reach.

### 5.6 Discussion

In this chapter, we have provided a realistic model of a P2P social network advertisement dissemination mechanism. We have designed and implemented a dissemination method so that the dissemination is carried out while cheating is difficult.

We have presented a heuristic for finding a minimum sum of distances between the seeding peers and the target nodes. We have defined several payment models and compared between them. We have also introduced and thoroughly tested different dissemination methods.

When trying to make general observations about the benefit of the learning dissemination schemes vs. the random dissemination ones, we can point out the following: For every required minimal average miss rate, we can eventually achieve this miss rate with a learning strategy while sending fewer messages per round than the random strategy that achieves the required average miss rate. Further, the benefit of the learning strategies over the random ones increases as the churn rates decrease. This is because when the network changes too fast, by the time something is learned about the network, it is no longer relevant. Hopefully, with a paying advertisements based incentive mechanism, the churn rates in a P2P network will indeed not be too high.
Finally, we have shown mechanisms that reward nodes in an exponential proportion to their availability. This was an important goal of our work, since it shows that ads can serve as a real incentive for users to keep their donated machines connected to the network for long periods of time.

In our work, users connect directly to peers in the P2P overlay. Since we assume the users use mobile devices, the information that can be learned by a peer about connected users is already limited. In order to enhance the protection of the privacy of the users, the clients can connect to the peer nodes using an anonymization mechanism such as Tor\(^4\).

\(^4\)https://www.torproject.org/
Chapter 6

Conclusions

In this work, we set out to advance the field of P2P live streaming by introducing structured algorithm building and decomposition. In Chapter 2, we started by implementing MOLSTREAM, an open source modular framework for rapid prototyping, testing and performance tuning of P2P live streaming protocols. MOLSTREAM defines four basic components which can be used to construct a multitude of P2P live streaming protocols. Indeed we have implemented and tested various known protocols in MOLSTREAM and demonstrate how MOLSTREAM facilitates comparing between them.

In Chapter 3, we have presented a further refinement in the form of the ingredients concept and an implementation of the STREAMAID framework for constructing extensible middleware for P2P live streaming. Ingredients allow encapsulation of the smallest design decisions while remaining flexible and extendible. We disentangled several such protocols into their most basic design decisions, uncovering in the process commonality between several systems. Our large-scale experiments illustrate the power of the abstraction and the flexibility of STREAMAID.

Next, in Chapter 4, we proposed a framework in which the dissemination algorithms are gradually switched to ones that appear to have better performance on important metrics for the live stream quality. Using our extensive protocol library, we generated many P2P live streaming algorithms, and we reach a conclusion that the is no one algorithm that beats them all – in different system states, different algorithms prevail. We devise a MAB based scheme and show how our adaptive framework is able to closely match the performance of the best algorithm without incurring a high switch cost.

In Chapter 5, we have provided a realistic model of a P2P social network advertisement dissemination mechanism to be used as an economic incentive. We have designed and implemented a dissemination method so that the dissemination is carried out while cheating is difficult. We show how in some settings, the payment is super linear in the session length, making this an effective incentive mechanism.

Future version of our adaptive framework can use more sophisticated methods of algorithm selection, such as contextual MAB or genetic algorithms.

Developing our advertisement dissemination mechanism, in the future, we would like to examine different optimization functions for seeding peer selection and possibly different
heuristics for each one of them. We would like to test how the different payment models affect the dissemination. More work can be done to ensure the exponential payments for availability, i.e., that nodes can take availability into account when dissemination ADMs. In addition, advertiser nodes could decide on different “punishments” for unavailable nodes. Different network overlays can also affect the mechanism. For instance, the overlay achieved by Kademlia [MM02] is not a hypercube. Hence, we would like to test the effect of the overlay on these mechanisms. Finally, we intend to further explore the tradeoffs between desired miss rates and communication overhead.
Bibliography


[MSR+14] Miguel Matos, Valerio Schiavoni, Etienne Riviere, Pascal Felber, and Renato Oliveira. Laystream: composing standard gossip protocols for live...


לבסוף, אנו חוחרים כיצדدخل מספר פרוטוקולים נוספים המופעלים ב לנו, בייחוד שימוש בשימוש ב Multi-armed bandit芳香 לשphalt ציוד השיתוף בה הם מופעלים

ובשתים את הפרוטוקולים בשיתוף회

ולכל אחד שיתוף שיתוף ב Multi-armed bandit芳香 לשphalt ציוד השיתוף

ובשתים את הפרוטוקולים בשיתוף להשתתף בשיתוף

מתוך אתADIUS

https://github.com/alibov/StreamAid

, והוא זמין כפרויקט קוד פתוח ב Java

ככדב

בשתים אתрош ב

ולכל אחד שיתוף שיתוף ב Multi-armed bandit芳香 לשphalt ציוד השיתוף

ובשתים את הפרוטוקולים בשיתוף להשתתף בשיתוף

מתוך אתADIUS

https://github.com/alibov/StreamAid

, והוא זמין כפרויקט קוד פתוח ב Java

ככדב

בשתים אתрош ב

ולכל אחד שיתוף שיתוף ב Multi-armed bandit芳香 לשphalt ציוד השיתוף

ובשתים את הפרוטוקולים בשיתוף להשתתף בשיתוף

מתוך אתADIUS

https://github.com/alibov/StreamAid

, והוא זמין כפרויקט קוד פתוח ב Java

ככדב

בשתים אתрош ב

ולכל אחד שיתוף שיתוף ב Multi-armed bandit芳香 לשphalt ציוד השיתוף

ובשתים את הפרוטוקולים בשיתוף להשתתף בשיתוף

מתוך אתADIUS

https://github.com/alibov/StreamAid

, והוא זמין כפרויקט קוד פתוח ב Java

ככדב

בשתים אתрош ב

ולכל אחד שיתוף שיתוף ב Multi-armed bandit芳香 לשphalt ציוד השיתוף

ובشتימב בשיתוף להשתתף בשיתוף

מתוך אתADIUS

https://github.com/alibov/StreamAid

, והוא זמין כפרויקט קוד פתוח ב Java

ככדב

בשתים אתрош ב

ולכל אחד שיתוף שיתוף ב Multi-armed bandit芳香 לשphalt ציוד השיתוף

ובשתים את הפרוטוקולים בשיתוף להשתתף בשיתוף

מתוך אתADIUS

https://github.com/alibov/StreamAid

, והוא זמין כפרויקט קוד פתוח ב Java

ככדב

בשתים אתрош ב

ולכל אחד שיתוף שיתוף ב Multi-armed bandit芳香 לשphalt ציוד השיתוף

ובשתים את הפרוטוקולים בשיתוף להשתתף בשיתוף

מתוך אתADIUS

https://github.com/alibov/StreamAid

, והוא זמין כפרויקט קוד פתוח ב Java

ככדב

בשתים אתрош ב

ולכל אחד שיתוף שיתוף ב Multi-armed bandit芳香 לשphalt ציוד השיתוף

ובשתים את הפרוטוקולים בשיתוף להשתתף בשיתוף

מתוך אתADIUS

https://github.com/alibov/StreamAid

, והוא זמין כפרויקט קוד פתוח ב Java

ככדב

בשתים אתрош ב

ולכל אחד שיתוף שיתוף ב Multi-armed bandit芳香 לשphalt ציוד השיתוף

ובשתים את הפרוטוקולים בשיתוף להשתתף בשיתוף

מתוך אתADIUS

https://github.com/alibov/StreamAid

, והוא זמין כפרויקט קוד פתוח ב Java

ככדב

בשתים אתрош ב

ולכל אחד שיתוף שיתוף ב Multi-armed bandit芳香 לשphalt ציוד השיתוף

ובשתים את הפרוטוקולים בשיתוף להשתתף בשיתוף

מתוך אתADIUS

https://github.com/alibov/StreamAid

, והוא זמין כפרויקט קוד פתוח ב Java

ככדב

בשתים אתрош ב

ולכל אחד שיתוף שיתוף ב Multi-armed bandit芳香 לשphalt ציוד השיתוף

ובשתים את הפרוטוקולים בשיתוף להשתתף בשיתוף

מתוך אתADIUS

https://github.com/alibov/StreamAid

, והוא זמין כפרויקט קוד פתוח ב Java

ככדב

בשתים אתрош ב

ולכל אחד שיתוף שיתוף ב Multi-armed bandit芳香 לשphalt ציוד השיתוף

ובשתים את הפרוטוקולים בשיתוף להשתתף בשיתוף

מתוך אתADIUS

https://github.com/alibov/StreamAid

, והוא זמין כפרויקט קוד פתוח ב Java

ככדב

בשתים אתрош ב

ולכל אחד שיתוף שיתוף ב Multi-armed bandit芳香 לשphalt ציוד השיתוף

ובשתים את הפרוטوك
תקציר

ה-sidebar vieille.jpg כמרכזי וחפים מה-IP הוא ה-64% מכלי ההעברה. ב-2018, יצורה בפליורוד באнтерניטיסי שיפור ב-79% מכלי ההעברה אינטרניטיסי. סופי עין: בערך ב-20%, ייעינת ברק짢י בערך ב-20%. כתといっても, פונקציונליות פונקציה וברזים הם של ה-IP, ומשתמשים במשתמשים אלו עם ה-IP. тоже, מימוש ו-IP, בין מרחבי עולם (peer-to-peer), הפונקציה של ה-IP מזווית בبقى. גם, שיפור ב-20% למשתמשים אגנזה משותפת בין-奭ה, בפונקציה בстраива, מקהל פונקציה של ה-IP. ידוע, אם נשיית ברק姽ה, בין-奭ה, ציור משותף משותפים פונקציה בᵔ.壮观ו. P2P

ניתן ב-P2P, מתאימה לתוכנה OStream, פיתוח של המודול הל-ברזים, המודול P2P allows for segmentation between P2P, בשפה של תכנית המודולナル מחשובים. בשפה של תכנית המודולNAL מחשובים, בשפה של תכנית המודולNAL מחשובים, בשפה של תכנית המודולNAL מחשובים, בשפה של תכנית המודולNAL מחשובים, בשפה של תכנית המודולNAL מחשובים, בשפה של תכנית המודולNAL מחשובים, בשפה של תכנית המודולNAL מחשובים, בשפה של תכנית המודולNAL מחשובים, בשפה של תכנית המודולNAL מחשובים, בשפה של תכנית המודולNAL מחשובים, בשפה של תכנית המודולNAL מחשובים, בשפה של תכנית המודולNAL מחשובים, בשפה של תכנית המודולNAL מחשובים, בשפה של תכנית המודולNAL מחשובים, בשפה של תכנית המודולNAL מחשובים, בשפה של תכנית המודולNAL מחשובים, בשפה של תכנית המודולNAL מחשובים, בשפה של תכנית המודולNAL מחשובים, בשפה של תכנית המודולNAL מחשובים, בשפה של תכנית המודולNAL מחשובים, בשפה של תכנית המודולNAL מחשובים, בשפה של תכנית המודולNAL מחשובים, בשפה של תכנית המודולNAL מחשובים, בשפה של תכנית המודולNAL מחשובים, בשפה של תכנית המודולNAL מחשובים, בשפה של תכנית המודולNAL מחשובים, בשפה של תכנית המודולNAL מחשובים, בשפה של תכנית המודולNAL מחשובים, בשפה של תכנית המודולNAL מחשובים, בשפה של תכנית המודולNAL מחשובים, בשפה של תכנית המודולNAL מחשובים, בשפה של תכנית המודולNAL מחשובים, בשפה של תכנית המודולNAL מחשובים, בשפה של תכנית המודולNAL מחשובים, בשפה של תכנית המודולNAL מחשובים, בשפה של תכנית המודולNAL מחשובים, בשפה של תכנית המודולNAL מחשובים, בשפה של תכנית המודולNAL מחשובים, בשפה של תכנית המודולNAL מח симвולי לא עליון בקבק אוחז בעקבי.
The research was conducted under the guidance of Professor Roy Friedman, and the results of the research were published in articles written by the researcher in conferences and journals throughout the period of his doctoral studies, and more recent versions of these are:


המחקר בוצע בתנאיי של פרופסור רועי פרידמן, בכ开荒ת למדעי המחשב.

חלק מנחוני המחקר וחברי המועצה מצטיינים בתחום המחקר והوثائق למחקרה בנסתם ובכתביה עת.

**תודה**

תודהแรกה למנהלי ספרה, פרופסור רועי פרידמן. תודה על ימיים וימי שעוניו של יום ויום התמחות.

אומר גם תודה לסטודנטים רבים של פרופסור ימי ויגפוזון, על תרומתם המשמעותית למחקרנו בתחום פעולות החברתיות.

בנוסף, תודה לברור שlicable וחברי המועצה, על תמיכה השארית שהRequestId להחזרה על כל אחד מהם. אני מקווה שיתיכהו גם כן בקשר נוח מזמיןackets אשר שלא הסופרים בראות כלם את עונתי פעולות פיתוח.

לנספים, אומר גם תודה למשתמשיبوابة של בית הספר ויום למ_fhimi והיוו את הדרכה ścianה עבור שירותים במחשבים בני משכון

ומעובר.
הפצת תוכן וייל בSegoe ברשתות עמיות עמוקות

היבור על מחקר

לשכם חלקי של הדרישות לבלבלת התוכן
דוקטור לפילוסופיה

אלכסנדר ליבוב

הונש ל bölümü הטכנולוגיה – מכון טכנולוגי לישראל
כסל innov - החיפה נובמבר 2015
הפצת תוכן יעילת ברשתות עמיתות על глובלי

אלכסנדר ליבוב