Live Streaming in P2P and Hybrid P2P-Cloud Environments for the Open Internet

AMIR H. PAYBERAH

Doctoral Thesis in Information
and Communication Technology
Stockholm, Sweden 2013

KTH VETENSKAP OCH KONST

ROYAL INSTITUTE
OF TECHNOLOGY

ROYAL INSTITUTE
OF TECHNOLOGY

Live Streaming in P2P and Hybrid P2P-Cloud Environments for the Open Internet

AMIR H. PAYBERAH

Doctoral Thesis in
Information and Communication Technology
Stockholm, Sweden 2013

TRITA-ICT/ECS AVH 13:05
ISSN 1653-6363
ISRN KTH/ICT/ECS/AVH-13/05-SE
ISBN 978-91-7501-686-3

KTH School of Information and
Communication Technology
SE-164 40 Kista
SWEDEN

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av teknologie licentiatesexamen i datalogi Torsdag den 13 Juni 2013 klockan 13:00 i sal E i Forum IT-Universitetet, Kungl Tekniskahögskolan, Isajordsgatan 39, Kista.

Swedish Institute of Computer Science
SICS Dissertation Series 60
ISRN SICS-D–60–SE
ISSN 1101-1335.

© Amir H. Payberah, June 13, 2013

Tryck: Universitetsservice US AB

Abstract

Peer-to-Peer (P2P) live media streaming is an emerging technology that reduces the barrier to stream live events over the Internet. However, providing a high quality media stream using P2P overlay networks is challenging and gives raise to a number of issues: (i) how to guarantee quality of the service (QoS) in the presence of dynamism, (ii) how to incentivize nodes to participate in media distribution, (iii) how to avoid bottlenecks in the overlay, and (iv) how to deal with nodes that reside behind Network Address Translators gateways (NATs).

In this thesis, we answer the above research questions in form of new algorithms and systems. First of all, we address problems (i) and (ii) by presenting our P2P live media streaming solutions: SEPIDAR, which is a multiple-tree overlay, and GLIVE, which is a mesh overlay. In both models, nodes with higher upload bandwidth are positioned closer to the media source. This structure reduces the playback latency and increases the playback continuity at nodes, and also incentivizes the nodes to provide more upload bandwidth.

We use a reputation model to improve participating nodes in media distribution in SEPIDAR and GLIVE. In both systems, nodes audit the behaviour of their directly connected nodes by getting feedback from other nodes. Nodes who upload more of the stream get a relatively higher reputation, and proportionally higher quality streams.

To construct our streaming overlay, we present a distributed market model inspired by Bertsekas auction algorithm, although our model does not rely on a central server with global knowledge. In our model, each node has only partial information about the system. Nodes acquire knowledge of the system by sampling nodes using the *Gradient overlay*, where it facilitates the discovery of nodes with similar upload bandwidth.

We address the bottlenecks problem, problem (iii), by presenting CLIVE that satisfies real-time constraints on delay between the generation of the stream and its actual delivery to users. We resolve this problem by borrowing some resources (*helpers*) from the cloud, upon need. In our approach, helpers are added on demand to the overlay, to increase the amount of total available bandwidth, thus increasing the probability of receiving the video on time. As the use of cloud resources costs money, we model the problem as the minimization of the economical cost, provided that a set of constraints on QoS is satisfied.

Finally, we solve the NAT problem, problem (iv), by presenting two NAT-aware peer sampling services (PSS): GOZAR and CROUPIER. Traditional gossip-based PSS breaks down, where a high percentage of nodes are behind NATs. We overcome this problem in GOZAR using one-hop relaying to communicate with the nodes behind NATs. CROUPIER similarly implements a gossip-based PSS, but without the use of relaying.

To Fatemeh, my beloved wife,
to Farzaneh and Ahmad, my parents, who I always adore,
and to Azadeh, Aram, and Kaveh my lovely sister and brothers.

Acknowledgements

I am deeply grateful to Professor Seif Haridi, my advisor, for giving me the opportunity to work under his supervision. I appreciate his invaluable help and support during my work. His deep knowledge in various fields of computer science, fruitful discussions, and enthusiasm have been a tremendous source of inspiration for me.

I would like to express my deepest gratitude to Dr. Jim Dowling for his excellent guidance and caring. I feel privileged to have worked with him and I am grateful for his support. He worked with me side by side and helped me with every bit of this research.

I would never have been able to finish my dissertation without the help and support of Fatemeh Rahimian, who contributed to many of the algorithms and papers in this thesis.

I would also like to thank Professor Alberto Montresor, Professor Vladimir Vlassov, Dr. Sarunas Girdzijauskas, Dr. Ali Ghodsi, Professor Christian Schulte, and Dr. Johan Montelius for their valuable feedbacks on my work during the course of my graduate studies. I am also grateful to Dr. Sverker Janson for giving me the chance to work as a member of CSL group at SICS. I acknowledge the help and support by Thomas Sjöland, the head of software and computer systems unit at KTH.

I would like to thank Cosmin Arad for providing KOMPICS, the simulation environment that I used in my work. I also thank Hanna Kavalionak, Tallat Mahmood Shafaat, Ahmad Al-Shishtawy, Roberto Roverso, Raul Jimenez, Flutra Osmani, Niklas Ekström, Martin Neumann, and Alex Averbuch for the fruitful discussions and the knowledge they shared with me. Besides, I am grateful to the people of SICS that provided me with an excellent atmosphere for doing research.

Finally, I am most grateful to my parents for helping me to be where I am now.

Contents

Chapter 1

Introduction

Live media streaming over the Internet is getting more popular every day. The conventional solution to provide this service is the client-server model, which allocates servers and network resources to each client request. However, providing a scalable and robust client-server model, such as Youtube, with more than one billion hits per day [1], is very expensive. There are few companies, who can afford to provide such an expensive service at large scale. An alternative solution is to use *IP multicast*, which is an efficient way to multicast a media stream over a network, but it is not used in practice due to its limited network-level support by Internet Service Providers.

Another approach is to use the *application level multicast*, which utilizes *overlay networks* to distribute large-scale media streams to a large number of users (*nodes*). A *Peer-to-Peer (P2P) overlay* is a type of overlay network in which each node simultaneously functions as both a client and a server. In this model, nodes that have all or part of the requested media can forward it to the requesting nodes. Since each node contributes its own resources, the capacity of the whole system grows when the number of nodes increases. Hence, P2P overlays can provide media streaming services at large scale, but with a relatively lower cost for the service provider than that of the client-server model, as network traffic and data storage/processing costs are pushed out to peer nodes.

The high scalability and low cost of P2P overlays, have lowered the barrier to stream live events over the Internet, and thus, have revolutionized media streaming technology. The question remains is how successful this new trend of technology is at providing a good quality of service (QoS) to end users. The QoS is defined in terms of two metrics in live streaming: *playback continuity*, and *playback latency.* To have a high playback continuity, or smooth media playback, nodes should receive data blocks of the stream with respect to certain timing constraints; otherwise, either the quality of the playback is reduced or its continuity is disrupted. Likewise, to have a low playback latency, nodes should receive points of the media that are close in time to the most recent part of the media delivered by the provider.

For example, in a live football match, people do not like to hear their neighbours celebrating a goal, several seconds before they see the goal happening.

Streaming live media with a high QoS, i.e., high playback continuity, and low playback latency, over a P2P overlay raises a number of issues:

- How do we guarantee the QoS in the presence of dynamism? P2P overlays are dynamic, meaning that nodes join/leave/fail continuously and concurrently in a process knows as *churn*. The network capacity also changes over time.

- How do we incentivize nodes to participate in media distribution? Nodes should be incentivized to contribute and share their resources in a P2P overlay. Otherwise, opportunistic nodes, called *free-riders*, can take advantage of the system without contributing to media distribution.

- How do we avoid bottlenecks in a P2P streaming overlay? Bottlenecks in the available upload bandwidth inside the P2P overlay network may limit the QoS experienced by users.

- How do we overcome the Network Address Translation gateways (NATs) problem? The presence of NATs in the Internet is a problem for P2P overlays. Nodes that reside behind NATs do not support direct connectivity by default, and other nodes cannot initiate connections to them.

1.1 Contribution

In this work, we answer the above questions in the form of new algorithms and systems. Some of the systems we developed address more than one of our research problems.

Sepidar and Glive. We address the two problems of the churn and free-riding by presenting our P2P live media streaming solutions: Sepidar [2] and Glive [3]. In Sepidar, we build multiple approximately minimal height overlay trees for content delivery, whereas, in Glive, we build a mesh overlay, such that the average path length between nodes and the media source is approximately minimum. In these structures, i.e., multiple-tree and mesh, each node receives data from multiple nodes, called its *partners*. If some partners of a node fail, the node can continue to receive the stream, as long as it has other partners to get data from them.

In both models, the nodes with higher available upload bandwidth are positioned closer to the media source, for two main reasons: (i) these nodes can serve relatively more nodes, thus reducing the average number of hops from nodes to the media source, and (ii) this model incentivizes nodes to provide more upload bandwidth, as nodes that contribute more upload bandwidth will be located closer to the media source, and consequently have relatively higher playback continuity and lower latency.

We use a reputation model to address the free-riding problem in Sepidar and Glive. We solve this problem in Sepidar through nodes auditing the behaviour of their child nodes in trees, while in Glive we implement a scoring mechanism that ranks the nodes, based on the received feedback from other nodes. In both systems, nodes who upload more of the stream get a relatively higher score or reputation. Nodes with higher rank will receive relatively improved video streams.

To construct our streaming overlays, we present a distributed market model inspired by the auction algorithm [4, 5]. Our distributed market model [6] differs from the classical implementations of the auction algorithm, in that we do not rely on a central server with a global knowledge of all participants. Instead, each node, as an auction participant, has only partial information about the system. Nodes continuously exchange their information, in order to collect more knowledge about other participating nodes. In our systems, nodes acquire knowledge of the system by sampling nodes using the gossip-generated *Gradient overlay* network [7, 8]. The Gradient overlay facilitates the discovery of nodes with similar upload bandwidth.

Clive. We present Clive, to satisfy soft real-time constraints on delay between the generation of the stream and its actual delivery to users, in case of bottlenecks in the available upload bandwidth inside the P2P overlay network. Our solution to this problem is assisting the P2P streaming network with a cloud computing infrastructure to guarantee a certain level of the QoS. For this purpose, we borrow some resources (*helpers*) from the cloud, upon need.

A helper could be an *active* computational node that participates in the streaming protocol, or it could be a *passive* storage node that just provides content on demand. The helpers increase the total upload bandwidth available in the system, thus, potentially reduce the playback latency. Both types of helpers could be rented on demand from an IaaS (Infrastructure as a Service) cloud provider, e.g., Amazon AWS. Considering the capacity and the cost of helpers, the problem to be solved becomes minimizing the economical cost of helpers, provided that a set of constraints on the QoS is satisfied.

Gozar and Croupier. We solve the NAT problem, by presenting two NAT-aware peer sampling services (PSS): Gozar [9] and Croupier [10]. Gossip-based PSS [11], which is a building block for our systems, provides a node with a uniform random samples of live nodes, where the sample size is typically much smaller than the system size.

In the Internet, where a high percentage of nodes are behind NATs, traditional gossip-based PSS' break down. We overcome this problem in Gozar by providing a distributed NAT-traversal to enable connectivity to nodes behind NATs (*private nodes*) using existing nodes not behind NATs (*public nodes*) as relay/rendezvous servers. We, then, go further in Croupier by removing relay/rendezvous nodes and building a gossip-based PSS without the use of relaying or hole-punching. As a result, we decrease the complexity and overhead of our protocol and increase its robustness to churn and failure.

However, this thesis does not cover the security issues and the problem of nodes colluding to receive the video stream for free in this thesis. To summarize, our contributions in this thesis include:

- a distributed market model to construct P2P streaming overlays, a tree-based overlay, Sepidar, and a mesh-based overlay, Glive, and two reputation-based solutions to overcome the free-riding problem in them,
- a cloud-assisted P2P live streaming system, Clive, that guarantees a higher bound on playback latency, if there exists bottlenecks in the available upload bandwidth inside the P2P overlay, by renting cloud resources,
- two NAT-aware gossip-based PSS' that provide uniform random samples in the presence of NATs using one-hop relaying in Gozar and without relaying in Croupier.

1.2 Publications

The list of papers published in this work are:

1. Amir H. Payberah, Hanna Kavalionak, Alberto Montresor, Jim Dowling, and Seif Haridi, *Lightweight Gossip-based Distribution Estimation*, The 15th IEEE International Conference on Communications (ICC), Budapest, Hungary, June 2013.
2. Amir H. Payberah, Jim Dowling, Fatemeh Rahimian and Seif Haridi, *Distributed Optimization of P2P Live Streaming Overlays*, Journal of Springer Computing, Vol. 94, No. 8, pp. 621–647, June 2012.
3. Amir H. Payberah, Hanna Kavalionak, Vimalkumar Kumaresan, Alberto Montresor, and Seif Haridi, *CLive: Cloud-Assisted P2P Live Streaming*, The 12th IEEE International Conference on Peer-to-Peer Computing (P2P), pp. 79–90, Tarragona, Spain, September 2012.
4. Jim Dowling and Amir H. Payberah, *Shuffling with a Croupier: Nat-Aware Peer-Sampling*, The 32nd IEEE International Conference on Distributed Computing Systems (ICDCS), pp. 102–111, Macau, China, June 2012.
5. Amir H. Payberah, Jim Dowling and Seif Haridi, *GLive: The Gradient overlay as a market maker for mesh-based P2P live streaming*, The 10th IEEE International Symposium on Parallel and Distributed Computing (ISPDC), pp. 153–162, Cluj-Napoca, Romania, July 2011.
6. Amir H. Payberah, Jim Dowling and Seif Haridi, *Gozar: NAT-friendly Peer Sampling with One-Hop Distributed NAT Traversal*, The 11th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS), pp. 1–14, Reykjavik, Iceland, June 2011.

7. Amir H. Payberah, Jim Dowling, Fatemeh Rahimian and Seif Haridi, *Sepidar: Incentivized Market-Based P2P Live Streaming on the Gradient Overlay Network*, The IEEE International Symposium on Multimedia (ISM), pp. 1–8, Taichung, Taiwan, December 2010.

8. Amir H. Payberah, Jim Dowling, Fatemeh Rahimian and Seif Haridi, *gradienTv: Market-based P2P Live Media Streaming on the Gradient Overlay*, The 10th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS), pp. 212–225, Amsterdam, Netherlands, June 2010.

The list of publications of the same author but not related to this work are:

1. Fatemeh Rahimian, Sarunas Girdzijauskas, Amir H. Payberah and Seif Haridi, *Subscription Awareness Meets Rendezvous Routing*, The 4th IARIA International Conference on Advances in P2P Systems (AP2PS), Barcelona, Spain, September 2012.

2. Fatemeh Rahimian, Sarunas Girdzijauskas, Amir H. Payberah and Seif Haridi, *Vitis: A Gossip-based Hybrid Overlay for Internet-scale Publish/Subscribe*, The 25th IEEE International Parallel & Distributed Processing Symposium (IPDPS), pp. 746–757, Anchorage, Alaska, USA, May 2011.

3. Hakan Terelius, Guodong Shi, Jim Dowling, Amir H. Payberah, Ather Gattami and Karl Henrik Johansson, *Converging an Overlay Network to a Gradient Topology*, The 50th IEEE Conference on Decision and Control and European Control Conference (CDC-ECC), pp. 7230–7235, Orlando, Florida, USA, December 2011.

1.3 Outline

The rest of this document is organized as follows:

- In Chapter 2, we present the required background for this thesis project. We review the main concepts of P2P media streaming and introduce a framework for classifying and comparing different P2P streaming solutions. Moreover, we go through the basic concepts behind peer sampling services and introduce the Gradient overlay. Furthermore, we show the effects of NATs on the behaviour of P2P applications, and explore existing NAT traversal solutions. Finally, we present a short introduction on the auction algorithms.

- In Chapter 3, we present our P2P live streaming systems using our distributed market model. In this chapter, we show how we use the Gradient overlay to improve the convergence time of our systems. Additionally, we present our free-rider detector mechanisms.

- In Chapter 4, we go through the details of our cloud-assisted P2P live streaming, and explain how we can guarantee the QoS in term of playback latency, when there are bottlenecks in the overlay network.
- In Chapter 5, we present our gossip-based PSS' and show how they provide uniform random samples of nodes at all nodes in a system, even when a high percentage of them are behind NAT.
- In Chapter 6, we conclude the thesis.

Chapter 2

Background and Related Work

In this chapter we present the main background work that is relevant for this thesis. First of all, we review the main concepts of P2P media streaming systems. Later, we present the basics of peer sampling services and the Gradient overlay as the core blocks of our systems. In addition, we show the connectivity problem among nodes in the Internet and present the common NAT traversal solutions. Finally, we cover the auction algorithm as a method we used in our systems to solve assignment problems.

2.1 P2P media streaming

In this section, we present the main questions on designing P2P media streaming systems and introduce a framework to organize existing P2P streaming systems. We also review some of the solutions to incentivize nodes to contribute in data dissemination.

2.1.1 P2P streaming overlays

There are two fundamental questions in building an overlay for P2P streaming:

1. How to construct and maintain a P2P streaming overlay?
2. How to distribute content to the nodes in a P2P streaming overlay?

Constructing and maintaining a P2P streaming overlay. The first question in P2P streaming systems is how to construct and maintain a content distribution overlay, or in other words, how nodes can discover other supplying nodes [12]. Some possible answers to this question are:

- Centralized method

- Hierarchical method
- Flooding method
- DHT-based method
- Gossip-based method

The *centralized* method is a solution used mostly in early P2P streaming systems. In this method, the information about all nodes, e.g., their address or available bandwidth, is kept in a centralized directory and the centralized directory is responsible to construct and maintain the overall topology. CoopNet [13] and DirectStream [14] are two sample systems that use the central method. Since the central server has a global view of the overlay network, it can handle nodes joining and leaving very quickly. One of the arguments against this model is that the server becomes a single point of failure, and if it crashes, no other node can join the system. The scalability of this model, also, is another problem. However, these problems can be resolved if the central server is replaced by a set of distributed servers.

The next solution for locating supplying nodes is using a *hierarchical* method. This approach is used in several systems, such as Nice [15], ZigZag [16], and Bulk-Tree [17]. In Nice and ZigZag, for example, a number of *layers* are created over the nodes, such that the lowest layer contains all the nodes. The nodes in this layer are grouped into some *clusters*, according to a property defined in the algorithm, e.g., the latency between nodes. One node in each cluster is selected as a *head*, and the selected head for each cluster becomes a member of one higher layer. By clustering the nodes in this layer and selecting a head in each cluster, they form the next layer, and so on, until it ends up in a layer consisting of a single node. This single node, which is a member of all layers is called the *rendezvous* point.

Whenever a new node comes into the system, it sends its join request to the rendezvous point. The rendezvous node returns a list of all connected nodes on the next down layer in the hierarchy. The new node probes the list of nodes, and finds the most proper one and sends its join request to that node. The process repeats until the new node finds a position in the structure where it receives its desired content. Although this solution solves the scalability and the single point of failure problems in the central method, it has a slow convergence time.

The third method to discover nodes is the controlled *flooding*, which is originally proposed by Gnutella [18]. GnuStream [19] is a system that uses this idea to find supplying nodes. In this system, each node has a *neighbour set*, which is a partial list of nodes in the system. Whenever a node seeks a provider, it sends its query to its neighbours. Each node forwards the request to all of its own neighbours except the one who has sent the request. The query has a time-to-live (TTL) value, which decreases after each rebroadcasting. The broadcasting continues until the TTL becomes zero. If a node that receives the request satisfies the node selection constraints, it will reply to the original sender node. This method has two main drawbacks. First, it generates a significant amount of network traffic and second, there is no guarantee for finding appropriate providers.

An alternative solution for discovering the supplying nodes is to use Distributed Hash Tables (DHT), e.g., Chord [20] and Pastry [21]. SplitStream [22] and [23] are two samples that work over a DHT. In these systems, each node keeps a routing table including the address of some other nodes in the overlay network. The nodes, then, can use these routing tables to find supplying nodes. This method is scalable and it finds proper providers rather quickly. It guarantees that if proper providers are in the system, the algorithm finds them. However, it requires extra effort to manage and maintain the DHT.

The last approach to find supplying nodes is the *gossip-based* method. Many algorithms are proposed based on this model, e.g., NewCoolstreaming [24], DONet/-Coolstreaming [25], PULSE [26], gradienTv [27] and [28] use a gossip-generated random overlay network to search for the supplying nodes. We use the gossip-generated Gradient overlay [7, 8] for node discovery in Sepidar and Glive. In the gossip-based method, each node periodically sends its data availability information to its *neighbours*, a partial view of nodes in the system, to enable them find appropriate suppliers, who possess data they are looking for. This method is scalable and failure-tolerant, but because of the randomness property of neighbour selection, sometimes the appropriate providers are not found in reasonable time.

Distributing contents in a P2P streaming overlay. In order to distribute streaming contents in a P2P overlay, we should decide:

1. What overlay topology is built for data dissemination?
2. What algorithm is used for data dissemination?

Many different overlay topologies have been used for data dissemination in P2P media streaming systems. The main topologies used for this purpose are:

- Tree-based topology
- Mesh-based topology
- Hybrid topology

The tree-based topology is divided to *single-tree* and *multiple-tree* structures. Early data delivery overlays use a single-tree topology, where data blocks are pushed over a tree-shaped overlay with a media source as the root of the tree. Nice [15], ZigZag [16], Climber [29] and [30] are examples of such systems. The low latency of data delivery is the main advantage of this approach. Disadvantages, however, include the fragility of the tree structure upon the failure of interior nodes and the fact that all the traffic is only forwarded by them.

The multiple-tree structure is an improvement on single-tree overlays, which was proposed for the first time in SplitStream [22]. In this model, the stream is split into substreams and each tree delivers one substream. Sepidar, CoopNet [13], gradienTv [27], Orchard [31], and ChunkySpread [32] are some solutions belonging to this class.

Although multiple-tree overlays improve some of the shortcomings of single-tree structures, they are still vulnerable to the failure of interior nodes. Rajaee et al. have shown in [33] that mesh overlays have consistently better performance than tree-based approaches for scenarios where there is churn and packet loss. The mesh structure is highly resilient to node failures, but it is subject to unpredictable latencies due to the frequent exchange of notifications and requests [12]. Glive, DONet/Coolstreaming [25], PULSE [26], Gossip++ [34], Chainsaw [35], and [28] are the systems that use a mesh-based overlay for data dissemination.

Another solution for data dissemination is a hybrid model that combines the benefits of the tree-based structure with the advantages of the mesh-based approach. Example systems include NewCoolStreaming [24], CliqueStream [36], mTreebone [37], Prime [38], and [23].

The second question in the content distribution is what algorithm should be used for data dissemination. The two most common answers to this question are:

- Push-based method
- Pull-based method

The *push-based* content distribution is a solution mostly used in tree structures. ZigZag [16] and SplitStream [22], as instances of single-tree and multiple-tree structures, respectively, use the push-based model for data dissemination. The push model in mesh-based overlays may generate lots of redundant messages, since nodes may receive the same data block from different neighbours. Although, Fortuna et al. in [28] resolved the redundancy problem of the push model in mesh overlays, the *pull-based* method is still the dominant data distribution model in mesh overlays.

In the pull-based model, nodes exchange their data block availability information and request each required data block explicitly from a neighbour that possesses that data block. Sepidar and Glive use push and pull data distribution models, respectively. The systems that use hybrid tree-mesh topologies, e.g., NewCoolStreaming [24], CliqueStream [36], and mTreebone [37], usually use both push and pull model at the same time.

A classification framework. We classify the existing P2P media streaming systems in two dimensions, each representing one aspect of the problem. The result is shown in Table 2.1. Each row in this table shows an approach to overlay construction, while each column shows a different data dissemination solution. Due to the lack of space, we just show a few systems in each cell.

	Single-tree (push)	Multiple-tree (push)	Mesh (push)	Mesh (pull)	Tree-Mesh (push-pull)
Centralized	DirectStream [14] HyMoNet [39]	CoopNet [13]			
Hierarchical	Nice [15] ZigZag [16] Climber [29]			BulkTree [17]	Prime [38]
Flooding				GnuStream [19]	
DHT		SplitStream [22]		CollectCast [40] Promise [41]	Pulsar [23] CliqueStream [36]
Gossip		Sepidar [2] gradienTv [27] Orchard [31] ChunkySpread [32]	Napa-Wine [28]	Glive [3] Coolstreaming [25] PULSE [26] Chainsaw [35] Bitos [44] DagStreamt [45]	Bullet [42] mTreebone [37] GridMedia [43]

Table 2.1: A framework to classify P2P media streaming systems.

2.1.2 Incentive mechanisms

A common problem in P2P streaming systems is *free-riding.* In P2P content distribution networks nodes should be incentivized to share their resources and contribute to data dissemination; otherwise, opportunistic nodes, called *free-riders*, can use the system without contributing any resources. This could have a serious impact on the quality of service of the P2P streaming system, leading to scalability issues and service degradation [46, 47]. The existing solutions to address the free-riding problem can be categorized as follows:

- Monetary-based
- Reciprocity-based
- Reputation-based

In the *monetary-based* scheme, users pay virtual currency to get content from other nodes. Each node plays a dual role of a content consumer and provider. A node, as a rational player, wants to maximize its profit, i.e., the quality of its received stream, but simultaneously reduces its costs, i.e., the amount of resources it contributes to the system. A popular modeling tool to study strategic interactions among such rational players is the game theory [46]. Some systems that use the game theory to overcome free-riders are [48–50].

Reciprocity-based mechanisms are similar to the *tit-for-tat* strategy in BitTorrent [51]. Here, nodes measure the amount of received stream from their neighbors, and keep the history of them. A node periodically decides to upload content to its neighbours, based on the local information about which neighbours have uploaded more to it in the past. PULSE [26], and Bitos [44] are two systems that use the reciprocity-based mechanism.

Another mechanism to resolve the free-riding problem is the *reputation-based* model. Nodes, in this model, receive scores based on their contribution to data dissemination. The higher score a node has, the higher reputation it achieves, and consequently the higher priority it has for receiving data. Nodes' reputations are constructed based on feedbacks from other nodes in the system that have interacted with them. Sepidar, Glive, BarterCast [52], EigenTrust [53], Give-to-Get [54], and BAR gossip [55] are a number of P2P streaming systems that use the reputation-based model.

2.2 Peer sampling service

Peer sampling services (PSS) have been widely used in large scale distributed applications, such as information dissemination [56], aggregation [57], and overlay topology management [8, 58]. Gossiping algorithms are the most common approach to implementing a PSS [9, 10, 59–63]. In gossip-based PSS', the protocol execution at each node is divided into periodic cycles. In each cycle, every node selects a node from its partial view and exchanges a subset of its partial view with the selected node. Both nodes subsequently update their partial views using the received node descriptors.

2.2.1 Gossip-based peer sampling service

Based on M. Jelasity et al. classification [11], implementations of gossip-based PSS' vary based on a number of different policies:

1. *Node selection*: determines how a node selects another node to exchange information with. It can be either selected randomly (*rand*), or based on the node's age (*tail*).

2. *View propagation*: determines how to exchange views with the selected node. A node can send its view with or without expecting a reply, called *push-pull* and *push*, respectively.

3. *View selection*: determines how a node updates its view after receiving the nodes' descriptors from another node. A node can either update its view randomly (*blind*), or keep the youngest nodes (*healer*), or replace the subset of nodes sent to the other node with the received descriptors (*swapper*).

In a gossip-based PSS, the sampled nodes should follow a uniform random distribution. Moreover, the overlay constructed by a PSS should preserve *indegree distribution*, *average shortest path* and *clustering coefficient*, close to a random network [11, 63]. The indegree distribution shows the distribution of the input links to nodes. The path length for two nodes is measured as the minimum number of hops between two nodes, and the average path length is the average of all path lengths between all nodes in the system. The clustering coefficient of a node is the number of links between the neighbors of a node divided by all possible links among them.

The *Gradient overlay* is a class of P2P overlays that uses a gossip-based PSS to arrange nodes using a local utility function at each node [7, 8]. The nodes in the Gradient overlay are ordered in descending utility values away from a core of the highest utility nodes. In other words, the highest utility nodes are found at the core of the Gradient overlay, and nodes with decreasing utility values are found at increasing distance from the center.

To build our streaming systems, Sepidar and Glive, we acquire knowledge of the network by sampling nodes from the Gradient overlay. The Gradient maintains two sets of neighbours using gossiping algorithms: a *similar-view* and a *random-view.* The similar-view of a node is a partial view of the nodes whose utility values are close to, but slightly higher than the utility value of this node. Nodes periodically gossip with each other and exchange their similar-views. Upon receiving a similar-view, a node updates its own similar-view by replacing its entries with those nodes that have closer (but higher) utility value to its own utility value. In contrast, the random-view constitutes a random sample of nodes in the system, and it is used both to discover new nodes for the similar-view and to prevent partitioning of the overlay.

2.2.2 NAT-aware peer sampling service

In networks where all nodes can directly communicate with each other, a gossip-based PSS' can ensure that node descriptors are distributed uniformly at random over all partial views [63]. However, in the Internet, where a high percentage of nodes are behind NATs and firewalls, traditional gossip-based PSS' become biased [64]. Nodes cannot establish direct connections to nodes behind NATs or firewalls (*private nodes*), and as a result private nodes become under-represented in partial views. Conversely, nodes that do support direct connectivity (*public nodes*) become over-represented in partial views. Kermarrec et al. also evaluated the impact of NATs on traditional gossip-based PSS' in [64], and showed that the network becomes partitioned when the number of private nodes exceeds a certain threshold.

There are two main techniques that are used to communicate with private nodes: *hole punching* [65] and *relaying* [66]. Hole punching can be used to establish direct connections that traverse the private node's NAT, and relaying can be used to send a message to a private node via a third party relay node that already has an established connection with the private node. In general, hole punching is preferable when large amounts of traffic will be sent between two nodes and when a slow connection setup time is not a problem. Relaying is preferable when a connection setup time should be short (less than one second) and small amounts of data will be sent over the connection.

Traditionally, gossip-based PSS' do not support connectivity to private nodes. However, as nodes are typically sampled from a PSS in order to connect to them, there are natural benefits to including NAT traversal as part of a PSS. The first PSS that addresses the problem of NATs was ARRG [67]. In ARRG, each node maintains an open list of nodes with whom it has had a successful gossip exchange in the past. When a node view exchange fails, it selects a different node from this open list. The open list, however, biases the PSS, since the nodes in the open list are selected more frequently for gossiping.

Nylon [64], is another NAT-aware PSS that uses all existing nodes in the system (both private and public nodes) as rendezvous servers (RVPs). A RVP provides connectivity to private nodes by facilitating hole-punching the private node's NAT. In Nylon, two nodes become the RVP of each other whenever they exchange their views. If a node selects a private node for gossip exchange, it hole-punches a direct connection to the private node using a chain of RVPs until the chain reaches the private node. The chains of RVPs in Nylon are unbounded in length, making Nylon fragile in networks with churn, as well as increasing overhead at intermediary nodes. Their chain of RVPs also performs poorly over high latency links, which are frequently found on the Internet [68].

In other work on NAT-aware gossiping, Renesse et al. [69] presented an approach to fairly distribute relay traffic over public nodes. In their system, each node balances the number of gossip requests it accepts to the number of gossip exchanges it has sent itself. Nodes that have already accepted enough gossip requests, forward them in a manner similar to Nylon, using chains of nodes as relay servers.

In our system Gozar, we replaced RVP chains with one-hop relaying to all private nodes. Private nodes discover and maintain a redundant set of public nodes that act as relay nodes on their behalf. Nodes shuffled with private nodes by relaying messages via at least one of the private node's relay nodes, where the addresses of the relay nodes are cached in node descriptors. Through redundant relay nodes and quickly expiring node descriptors, connectivity to private nodes is maintained and latency is kept low, even under churn. In Croupier, we introduce a novel mechanism for exchanging partial views in NATed networks to build a PSS without the use of relaying.

2.3 The assignment problem

We consider the following two problems as *assignment problems* [70]: (i) constructing the streaming overlays in Sepidar and Glive, and (ii) building a distributed NAT traversal. In this section, we shortly explain the assignment problem in general, and sketch a possible solution based on the auction algorithm [4].

Suppose there are n persons and n objects, and we want to find a pairwise matching among them. A matching between person i and object j is shown as a pair (i, j), and is associated with a benefit a_{ij}. We want to assign all persons to objects so as to maximize the total benefit. The set of persons and objects are denoted by $\mathcal{P}$ and $\mathcal{O}$, respectively.

We define an *assignment* $\mathcal{S}$ as a set of pairs (i, j) such that:

1. For all $(i, j) \in \mathcal{S}$, $i \in \mathcal{P}$ and $j \in \mathcal{O}$.

2. For each $i \in \mathcal{P}$, there is at most one pair $(i, j) \in \mathcal{S}$.

3. For each $j \in \mathcal{O}$, there is at most one pair $(i, j) \in \mathcal{S}$.

A *complete assignment* $\mathcal{A}$ is an assignment containing n pairs, such that each $i \in \mathcal{P}$ is assigned to a different $j \in \mathcal{O}$. Our goal is to find a complete assignment $\mathcal{A}$ over all assignments $\mathcal{S}$ that maximizes the total benefit. We can formulate this problem in the Integer Linear Programming (ILP) framework [5], as the following:

$$\text{maximize} \sum_{i=1}^{n} \sum_{\{j|(i,j)\in\mathcal{A}\}} a_{ij}x_{ij} \tag{2.1}$$

subject to

$$\sum_{\{j|(i,j)\in\mathcal{A}\}} x_{ij} = 1, \quad \forall i = 1, 2, \cdots, n \tag{2.2}$$

$$\sum_{\{i|(i,j)\in\mathcal{A}\}} x_{ij} = 1, \quad \forall j = 1, 2, \cdots, n \tag{2.3}$$

$$x_{ij} \in \{0, 1\}, \; \forall i = 1, 2, \cdots, n, \forall j = 1, 2, \cdots, n \tag{2.4}$$

where $x_{ij} = 1$ if a person i is assigned to an object j, and $x_{ij} = 0$, otherwise. The Constraint 2.2 requires that every person i is assigned to one object, and the Constraint 2.3 requires to ensure that each object is assigned to one person.

A popular solution to solve assignment problems is the auction algorithm [4, 5]. The auction algorithm models a real world auction, where people bid for the objects that brings them the highest profit, and the highest bids win. In our problem, n persons compete to be assigned to an abject among the set of n available objects. Like ordinary auctions, the bidders progressively increase their bid for the objects in a competitive process.

Each object j is associated with a *price* p_j, which is zero in the beginning, and is increased in auction iterations after accepting new *bids* from persons. A person i measures the *net profit*, v_{ij}, of each object j as the following:

$$v_{ij} = a_{ij} - p_j \tag{2.5}$$

The auction algorithm proceeds in iterations, and in each iteration it creates one assignment $\mathcal{S}$, such that the net profit of each connection under the assignment $\mathcal{S}$ is maximized. In each iteration, the algorithm updates the price of all objects in the assignment $\mathcal{S}$. If all persons of an assignment $\mathcal{S}$ are assigned, we have a complete assignment $\mathcal{A}$, and the algorithm terminates. Otherwise, the algorithm starts the next iteration by finding objects that offer maximal net profit for unassigned persons. Note that in the beginning of each iteration, the net profit of each assignment (Equation 2.5) under the assignment $\mathcal{S}$ should be maximum.

An iteration in the auction algorithm contains two phases: a *bidding phase* and an *assignment phase*:

- **Bidding phase:** In the bidding phase, each unassigned person i under the assignment $\mathcal{S}$ finds the object j^* that has the highest net profit:

  $$v_{ij^*} = \max_{j\in\mathcal{O}} v_{ij} \tag{2.6}$$

  To measure the amount of the bid, the person i finds the second best object j', and its net profit, $w_{ij'}$:

  $$w_{ij'} = \max_{j\in\mathcal{O}} v_{ij} \tag{2.7}$$

  Considering $\delta_{ij^*} = v_{ij^*} - w_{ij'}$ as the difference between the highest net profit and the second one, the person i raises the price of a preferred object j^* by the bidding increment δ_{ij^*}, and sends its bid, b_{ij^*}, to j^*:

  $$b_{ij^*} = p_{j^*} + \delta_{ij^*} \tag{2.8}$$

- **Assignment phase:** The object j, which receives the highest bid from i^*, removes the connection to the person i' (if there was any connection to i' in the beginning of the iteration), and assigns to i^*, i.e., the connection (i^*, j) is added to the current assignment $\mathcal{S}$. The object j also updates its own price to the received bid from the person i^*, i.e., $p_j = b_{i^*j}$.

Lemma 1. If $\delta_{ij} > 0$, the auction process will terminate.

Proof. If an object j receives m bids during m iterations, its price p_j increases by $\sum_{k=1}^{m} \delta_{ij}^{(k)}$, where $\delta_{ij}^{(k)}$ represents the added price at the iteration k. Therefore, over the iterations, the object j becomes more and more "expensive" and consequently its net profit decreases. This implies that an object can receive bids only for a limited number of iterations, while some other objects still have not received any bids. Hence, after some iterations, n distinct objects will receive at least one bid. Bertsekas shows in [5] that an auction algorithm with n persons, where the set of person-object pair is limited, terminates once n distinct objects receive at least one bid. □

However, if $\delta_{ij} = 0$, it may happen that several persons compete for the same set of objects without raising the price, thereby they may stuck in an infinite loop of bidding and assignment phases. To solve this problem, each person that bids for an object, should rise the price by a small value ϵ by biding $b_{ij^*} = p_{j^*} + \delta_{ij^*} + \epsilon$. The details of how ϵ is selected is out of the scope of our work and can be found in [5].

Chapter 3

P2P Live Streaming

Live streaming using overlay networks on the Internet requires distributed algorithms that strive to use the nodes' resources efficiently in order to ensure that the viewer quality is good. To improve user viewing experience, systems need to maximize the playback continuity of the stream at nodes, and minimize the playback latency between nodes and the media source. Nodes should be incentivized to contribute resources through improved relative performance, and nodes that attempt to free-ride, by not contributing resources, should be detected and punished. In order to improve system performance in the presence of asymmetric bandwidth at nodes, it is also crucial that nodes can effectively utilize the extra resources provided by the better nodes.

In this chapter, we present our P2P streaming systems, Sepidar [2] and Glive [3], that meet these requirements. In Sepidar, we build multiple approximately minimal height streaming overlay trees, where the nodes with higher available upload bandwidth are positioned higher in the tree as they can support relatively more child nodes. Minimal height trees help reduce both the probability of streaming disruptions and the average playback latency at nodes. In this system, the media stream is split into a set of substreams, and each tree delivers one substream. Multiple substreams allow more nodes to contribute bandwidth and enable trees to be more robust [22]. Likewise, in Glive, we build a mesh overlay, such that the average path length between nodes and the media source is approximately minimum. In Glive, we divide the media stream into a sequence of *blocks*, and each node pulls the required blocks of the stream from a set of nodes in the mesh.

To build our streaming overlays, we first describe an Integer Linear Programming (ILP) formulation of the topology building problem and provide a centralized solution for it based on the auction algorithm [4], and later we propose a distributed market model to solve the problem at large scale. In our distributed market, we do not rely on a central server with a global knowledge of all participants, and each node has only partial information about the system.

To improve the speed of convergence of the streaming overlays, nodes execute the market model in parallel using samples taken from the *Gradient overlay* [8]. The Gradient is a gossip-generated overlay network, where nodes are organized into a gradient structure with the media source at the center of the Gradient and nodes with decreasing relative upload bandwidth found at increasing distance from the center. When nodes sample from their neighbours in the Gradient, they receive nodes with similar upload bandwidths. In a converged overlay, the sampled nodes will be located at similar distance, in terms of number of hops, from the media source. Although we only consider upload bandwidth for constructing the Gradient and our streaming overlays, the model can be easily extended to include other characteristics such as node uptime, load and reputation.

We also address the free-riding problem, as one of the problems in P2P streaming systems, in our systems. Nodes are not assumed to be cooperative; nodes may execute protocols that attempt to download data blocks without forwarding it to other nodes. We resolve this problem in Sepidar through parent nodes auditing the behaviour of their child nodes in trees. We also address free-riding in Glive by implementing a scoring mechanism that ranks the nodes. Nodes who upload more of the stream have relatively higher score. In both solutions, nodes with higher rank will receive a relatively improved quality. We do not, however, address the problem of nodes colluding to receive the video stream for free.

3.1 Problem description

In this work, we want to build a P2P overlay for live media streaming and adaptively optimize its topology to minimize the average *playback latency* and improve timely delivery of the stream. Playback latency is the difference between the playback time (*playback point*) at the media source and at a node.

Intuitively, nodes with higher upload bandwidth should be located closer to the media source. Since, these nodes can serve relatively more nodes, such a structure reduces the average distance from nodes to the media source, and consequently decreases the playback latency. A similar overlay structure is used in a few other systems. For example, LagOver [71] is an information dissemination overlay that organizes nodes according to their resource constraints and the maximum acceptable latency to receive the information from the source. In this section, we first describe this problem in the tree-based approach, and then present the required modification to apply it for the mesh-based approach.

In our tree-based model, the media stream is split into a number of substreams or *stripes*, and each stripe is divided into *blocks* of equal size without any coding. Every block has a sequence number that indicates its playback order in the stream. A node retrieves the stripes independently, from any other node that can supply them. We define the number of *download-slots* and *upload-slots* of a node as the number of stripes that a node is able to simultaneously download and forward, respectively. The set of all download-slots and upload-slots in an overlay are also denoted by $\mathcal{D}$ and $\mathcal{U}$, respectively. Similarly, the set of download-slots and upload-

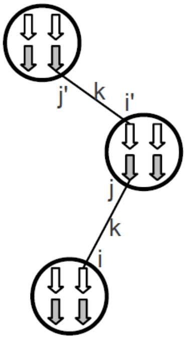


Figure 3.1: The connection between download-slots i and i' and upload-slots j and j' for strip k. The white arrows are download-slots and the gray arrows are the upload-slots.

slots of a node p are shown by $\mathcal{D}(p)$ and $\mathcal{U}(p)$.

A node is called the *owner* of its slots (either upload-slots or download-slots), and the function $owner(i)$ returns the node that owns the slot i. Any two slots i and j are *similar*, if they are owned by the same node, i.e., $owner(i) = owner(j)$. For each download-slot i, the set of all download-slots similar to i is the *similarity class* of i, and denoted by $\mathcal{M}_{\mathcal{D}}(i)$. Likewise, the similarity class of an upload-slot j is the set of upload-slots owned by the owner of j and is shown by $\mathcal{M}_{\mathcal{U}}(j)$.

Without loss of generality, we assume every node owns the same number of download-slots, equal to the number of stripes, and a potentially different number of upload-slots. In order to provide a full media to all the nodes, (i) every download-slot needs to be assigned to an upload-slot, (ii) each upload-slot should be assigned to at most one download-slot, (iii) similar download-slots, i.e., download-slots at the same node, must download distinct stripes, and (iv) nodes should not have loop back connections from their download-slots to their own upload-slots.

This problem can be defined as an *assignment problem* [70]. A connection between a download-slot i and an upload-slot j for a stripe k is shown as a triple (i, j, k), and it is associated with a *cost* c_{ijk} (Figure 3.1). The cost can be defined based on different metrics, e.g., the latency to the source, the number of hops to the source or the locality of the nodes, that is a connection between two nodes in the same Autonomous System (AS) has lower cost compare to a connection between two nodes in different ASs. Formally the cost is defined as the following:

$$c_{ijk} = \begin{cases} c_{i'j'k} + d_{ij} & \text{if } owner(j) = owner(i'),\ i' \in \mathcal{D}, \text{ and } j' \in \mathcal{U} \\ 0 & \text{if } owner(j) = \text{source} \\ \infty & \text{if there is no path from } owner(i) \text{ to the source} \end{cases} \tag{3.1}$$

where d_{ij} is the added cost in a connection between a download-slot i and an upload-slot j. For example, if the cost is defined as the latency of the node to the source, then d_{ij} will be the connection latency between the $owner(i)$ and the $owner(j)$, and if the cost is the number of hops to the source, then $d_{ij} = 1$.

With $\mathcal{V}$ being the set of all stripes, we define an *assignment* $\mathcal{S}$ as a set of triples (i, j, k), such that:

1. For all $(i, j, k) \in \mathcal{S}$, $i \in \mathcal{D}$ and $j \in \mathcal{U}$ and $k \in \mathcal{V}$.
2. For each $i \in \mathcal{D}$, there is at most one triple $(i, j, k) \in \mathcal{S}$.
3. For each $j \in \mathcal{U}$, there is at most one triple $(i, j, k) \in \mathcal{S}$.
4. For each $k \in \mathcal{V}$, there is at most one triple $(i, j, k) \in \mathcal{S}$ for all $i \in \mathcal{M}_{\mathcal{D}}(i)$.

The last constraint implies that the download-slots in one similarity class cannot download the same stripe. In other words, each download-slot in a node should download a distinct stripe. Note that with the above definition, it is possible to have cyclic connections among nodes in an assignment.

A *complete assignment* $\mathcal{A}$ is an assignment with the above definition that contains exactly $|\mathcal{D}|$ triples, i.e., all download-slots are assigned. To have a complete assignment, the total number of download-slots should be less than or equal to the total number of upload-slots, i.e., $|\mathcal{D}| \leq |\mathcal{U}|$. The playback latency of a node depends on the maximum latency of the node in different stripe trees. Therefore, to improve the playback latency we should minimize the latency of a node for all stripes simultaneously. Hence, we use the average distance of a node at all stripe trees as the cost function. Considering a complete assignment $\mathcal{A}$, the cost of a node p is defined as:

$$\mathcal{C}_{\mathcal{A}}(p) = \frac{1}{|\mathcal{V}|} \sum_{i \in \mathcal{D}(p)} \sum_{j \in \mathcal{U}} \sum_{k \in \mathcal{V}} c_{ijk} \cdot x_{ijk} \tag{3.2}$$

where $x_{ijk} = 1$ if a download-slot i is assigned to an upload-slot j for a stripe k, and $x_{ijk} = 0$, otherwise. Since putting the nodes with higher upload-slots closer to the source can reduce the average distances of all the nodes to the source [27, 28], we bias the cost of each node p by the number of its upload-slots:

$$\mathcal{C'}_{\mathcal{A}}(p) = \frac{1}{|\mathcal{V}|} \sum_{i \in \mathcal{D}(p)} \sum_{j \in \mathcal{U}} \sum_{k \in \mathcal{V}} \frac{c_{ijk}}{m_i} \cdot x_{ijk} \tag{3.3}$$

where $m_i = |\mathcal{U}(owner(i))| = |\mathcal{U}(p)|$ denotes the number of upload-slots that the owner of i has. Then, the average cost of all the nodes in a complete assignment $\mathcal{A}$ is measured as:

$$\begin{aligned}
\mathbb{C}_{\mathcal{A}} =& \frac{1}{|\mathcal{N}|} \sum_{p \in \mathcal{N}} \mathcal{C'}_{\mathcal{A}}(p) \\
=& \frac{1}{|\mathcal{N}| \cdot |\mathcal{V}|} \sum_{p \in \mathcal{N}} \sum_{i \in \mathcal{D}(p)} \sum_{j \in \mathcal{U}} \sum_{k \in \mathcal{V}} \frac{c_{ijk}}{m_i} \cdot x_{ijk} \\
=& \frac{1}{|\mathcal{N}| \cdot |\mathcal{V}|} \sum_{i=1}^{|\mathcal{D}|} \sum_{j \in \mathcal{U}} \sum_{k \in \mathcal{V}} \frac{c_{ijk}}{m_i} \cdot x_{ijk}
\end{aligned} \tag{3.4}$$

where $\mathcal{N}$ is the set of all the nodes. Therefore, the problem to be solved turns out to be finding a complete assignment $\mathcal{A}$ among all the possible complete assignments, which minimizes the total cost $\mathbb{C}_{\mathcal{A}}$. Since, the term $\frac{1}{|\mathcal{N}|\cdot|\mathcal{V}|}$ is a constant we ignore it in the optimization process.

Putting all the above constraints together, we formulate the problem in the ILP framework [5], as the following:

$$\text{minimize} \quad \sum_{i=1}^{|\mathcal{D}|} \sum_{\{j|(i,j,k)\in\mathcal{A}\}} \sum_{\{k|(i,j,k)\in\mathcal{A}\}} \frac{c_{ijk}}{m_i} \cdot x_{ijk} \tag{3.5}$$

subject to

$$\sum_{\{j|(i,j,k)\in\mathcal{A}\}} \sum_{\{k|(i,j,k)\in\mathcal{A}\}} x_{ijk} = 1, \quad \forall i \in \mathcal{D} \tag{3.6}$$

$$\sum_{\{i|(i,j,k)\in\mathcal{A}\}} \sum_{\{k|(i,j,k)\in\mathcal{A}\}} x_{ijk} \leq 1, \quad \forall j \in \mathcal{U} \tag{3.7}$$

$$\sum_{\{i\in\mathcal{M}_{\mathcal{D}}(i)|(i,j,k)\in\mathcal{A}\}} \sum_{\{j|(i,j,k)\in\mathcal{A}\}} x_{ijk} = 1, \quad \forall k \in \mathcal{V} \tag{3.8}$$

$$x_{ijk} \in \{0,1\}, \ \forall i \in \mathcal{D}, j \in \mathcal{U}, k \in \mathcal{V} \tag{3.9}$$

The first constraint requires that every download-slot i is assigned to exactly one upload-slot. The second constraint ensures that each upload-slot is assigned to at most one download-slot. It also stated that if the number of upload-slots are greater than the number of download-slots, some of the upload-slots remain unassigned. The third constraint ensures that the download-slots in a similarity class download distinct stripes.

Our model of the system should consider dynamism, while solving this assignment problem. A good solution, therefore, should assign download-slots to upload-slots as quickly as possible. Centralized solutions to this problem are possible for small system sizes. For example, if all the nodes send the number of their upload-slots to a central server, the server can use an algorithm that solves a linear sum assignments, e.g., the auction algorithm [4], the Hungarian method [72], or more recent high-performance parallel algorithms [70]. For large scale systems, however, a centralized solution is not appropriate, since it can become a bottleneck. In the next section, we briefly sketch a possible solution with the auction algorithm [4]. Later, in Section 3.3 we present a distributed model of the auction algorithm that solves this problem at large scale.

3.2 Centralized solution

We use the auction algorithm for $|\mathcal{D}|$ download-slots that compete for being assigned to some upload-slot among the set of $|\mathcal{U}|$ available upload-slots. A matching between a download-slot i and an upload-slot j for stripe k is associated with a *profit* a_{ijk}, and the goal of the auction is to maximize the total profit for all the matchings, which is:

$$\sum_{i=1}^{|\mathcal{D}|} \sum_{j \in \mathcal{U}} \sum_{k \in \mathcal{V}} a_{ijk} \cdot x_{ijk} \tag{3.10}$$

where x_{ijk} is defined in Equation 3.2, and a_{ijk} is calculated as:

$$a_{ijk} = \frac{m_i}{c_{ijk}} \tag{3.11}$$

Note that m_i and c_{ijk} are already defined in Section 3.1. Hereafter, we refer to m_i as *money*. Equation 3.11 simply says that a connection with a lower cost is more desirable, and the more money a download-slot has, the more profit it gets by creating a connection to a lower cost upload-slot.

Each download-slot has a certain amount of money, with which it finds a matching that maximizes its profit. Each upload-slot j is associated with a *price* p_j. The price of an unassigned upload-slot is zero, and is increased in auction iterations after accepting new *bids* from download-slots (the bidding process will be described later in this section). We define the *net profit* of an upload-slot as its profit minus its current price. A download-slot i measures the net profit, v_{ijk}, of each upload-slot j for a stripe k as the following:

$$\begin{aligned} v_{ijk} =& a_{ijk} - p_j \\ =& \frac{m_i}{c_{ijk}} - p_j \end{aligned} \tag{3.12}$$

As mentioned in Section 2.3, the auction algorithm proceeds in iterations, and it terminates when all the download-slots are assigned. The bidding phase and assignment phase in each iteration is as follows:

- **Bidding phase:** In this phase each unassigned download-slot i under the assignment $\mathcal{S}$ finds the upload-slot j^* with highest net profit for a stripe k, where k is not assigned to any other download-slots $i \in \mathcal{M}_{\mathcal{D}}(i)$:

$$v_{ij^*k} = \max_{j \in \mathcal{U}} v_{ijk} \tag{3.13}$$

The download-slot i also finds the second best upload-slot j' for stripe k, such that j' is not owned by the owner of j^*, i.e., $j' \notin \mathcal{M}_{\mathcal{U}}(j^*)$. The second best net profit, $w_{ij'k}$, equals:

$$w_{ij'k} = \max_{j \in \mathcal{U} - \mathcal{M}_{\mathcal{U}}(j^*)} v_{ijk} \tag{3.14}$$

Considering δ_{ij^*k} as the difference between the highest net profit and the second one, i.e., $\delta_{ij^*k} = v_{ij^*k} - w_{ij'k}$, the download-slot i computes the bid, b_{ij^*k}, for the upload-slot j^*:

$$b_{ij^*k} = p_{j^*} + \delta_{ij^*k} \tag{3.15}$$

In this process at each iteration the download-slot i raises the price of a preferred upload-slot j^* by the bidding increment δ_{ij^*k}.

- **Assignment phase:** The upload-slot j, which received the highest bid from i^*, removes the connection to the download-slot i' (if there was any connection to i' at the beginning of the iteration), and assigns to i^*, i.e., the connection (i^*, j, k) is added to the current assignment $\mathcal{S}$. The upload-slot j also updates its own price to the received bid from the download-slot i^*, i.e., $p_j = b_{i^*jk}$.

As shown in Section 2.3, the auction process terminates if $\delta_{ijk} > 0$. Although the presented auction algorithm was shown for the multiple-tree approach, we can easily use it to build a mesh overlay. In contrast to the multiple-tree approach, in the mesh-based overlay, we do not split the stream into stripes. The video is divided into a set of $\mathcal{B}$ blocks of equal size without any coding. Every block $b \in \mathcal{B}$ has a sequence number that indicates its playback order in the stream. Nodes can pull each block independently, from any other node that can supply it. Each node has a *partner list*, which is a small subset of nodes in the system. A node can create a bounded number of download connections, equals to its number of download-slots, to partners and accept a bounded number of upload connections, equals to its number of upload-slots, from partners over which blocks are downloaded and uploaded, respectively.

Unlike the tree-based approach that assigns download-slots to upload-slots of nodes for each stripe, here, we need to find the assignments for each block. Biskupski et al. in [73] show that a block disseminated through a mesh overlay follows a tree-based diffusion pattern for each block. Therefore, the objective is to minimize the cost function for every block b, such that a shortest path tree is constructed over the set of available connections for every block. We define the cost of connection c_{ijb} from a download-slot i to an upload-slot j for a block b as the minimum distance, e.g., the number of hops, from the owner of upload-slot j to the media source.

Since the auction algorithm is centralized, it does not scale to many thousands of nodes, as both the computational overhead of solving the assignment problem and communication requirements on the server become excessive, breaking our real-time constraints [70]. In the next section, we present a distributed market model as an approximate solution to this problem.

3.3 Distributed solution

In this section, we present a distributed market model to construct multiple-tree and mesh overlays for media streaming.

3.3.1 Multiple-tree overlay

Our distributed market model is based on minimizing costs (Equation 3.5) through nodes iteratively bidding for upload-slots. We define a node q as the *parent* of a *child* p, if an upload-slot of q is bounded to a download-slot of p. Nodes in this system compete to become children of nodes that are closer to the media source, and parents prefer children nodes who offer to forward the highest number of copies of the stripes. A child node explicitly requests and pulls the first block it requires in a stripe from its parent, and the parent, then, pushes to the child subsequent blocks in the stripe, as long as it remains the child's parent. Children proactively switch parents when they get more net benefit by changing their parents.

We use the following three properties, calculated at each node, to build the multiple-tree overlay:

1. *Money*: the total number of upload-slots at a node. A node uses its money to bid for a connection to another node's upload-slot for each stripe.

2. *Price*: the minimum money that should be bid when trying to establish a connection to an upload-slot. The price of a node that has an unused upload-slot is zero, otherwise the node's price equals the lowest money of its already connected children. For example, if node p has three upload-slots and three children with monies 2, 3 and 4, the price of p is 2. Moreover, the price of a node that has a *free-riding* child, a node not contributing in data dissemination, is zero.

3. *Cost*: the cost of an upload-slot at a node for a particular stripe is the distance between that node and the media source (root of the tree) for that stripe. Since the media stream consists of several stripes, nodes may have different costs for different stripes. The closer a node is to the media source for a stripe, the more desirable parent it is for that stripe. However, other metrics, such as the nodes' locality, can be taken into account for measuring the cost. For example, if two nodes have the same distance to the source, the cost of choosing any of them by the nodes in the same Autonomous System (AS) is lower than that of the nodes in a different AS. Nodes constantly try to reduce their costs over all their parent connections by competing for connections to the nodes closer to the media source.

Our market model can be best described as an approximate auction algorithm. Similar to the centralized solution in Section 3.2, for each stripe, child nodes place bids for upload-slots at the parent nodes with the highest net profit, e.g., closest nodes to the media source. Note that the money of a node is not used up after bidding and can be reused to bid for other connections. Therefore, if a node can afford a high net profit parent for one stripe, it can also afford other good parents for other stripes.

Nodes increase their price by receiving new bids, and the more expensive a node is, the lower net profit it has. Thus, a parent node, which had a high net profit in one iteration, turns out to be a low profit node after receiving a number of bids. Hence, the seeking nodes will try to bid for other nodes with a higher net profit. This implies that in an overlay with $|\mathcal{D}|$ download-slots, if there is no churn in the system, eventually $|\mathcal{D}|$ distinct upload-slots receive at least one bid, and consequently the algorithm terminates by assigning all the download-slots to upload-slots. However, in a dynamic network, where nodes continuously join and leave the system, our algorithm keeps running and optimizes the connections in the overlay.

A parent node sets a price of zero for an upload-slot when at least one of its upload-slots is unassigned. Therefore, the first bid for an upload-slot will always win, enabling children to immediately connect to available upload-slots. When all of a parent's upload-slots are assigned, it sets the price for an upload-slot to the money of its child with the lowest number of upload-slots, i.e., the lowest money. If a child with more money than the current price for an upload-slot bids for an upload-slot, it will win the upload-slot and the parent will replace its child with the lowest money with the new child. A child that has lost an upload-slot has to discover new nodes and bid for their upload-slots.

One crucial difference with the auction algorithm is that our market model is decentralized; nodes have only a partial (changing) view of a small number of nodes, called *partners*, in the system with whom they can bid for upload-slots. Moreover, in contrast to the auction algorithm, the price of upload-slots does not always increase - it can be reset to zero, if a child node is detected as a free-rider. A node is free-rider if it is not correctly forwarding all the stripes it promises to supply. As such, it is a *restartable auction*, where the auction is restarted because a bidder did not have sufficient funds to complete the transaction.

To construct the overlay, nodes periodically send their money, cost, price, and *buffer map* to their partners. The buffer map shows the last blocks that a node has in its buffer for different stripes. For each stripe k, a node p periodically checks if it has a node in its partners that has (i) a lower cost than its current parent, (ii) a price less than its money and (iii) blocks ahead of its block in stripe k. As the method `FindParent` shows in Algorithm 1, if such a node is found, it is added to a list of candidate parents for stripe k. Next, the node p chooses a node q from the candidates that provides the highest net profit for strip k, i.e., $\frac{p.money}{q.cost_k} - q.price$. If two nodes have the same net profit, it selects the one with higher money.

The handler `AssignRequest` shows, if a node q that receives a connection request from node p for stripe k, has a free upload-slot, it accepts the request, otherwise, if p's money is greater than the price of q, q abandons its child that has the lowest money, and accepts p as a new child. The disconnected node has to find a new

Algorithm 1 Parent selection methods.

```
// Find candidate parents for stripe $k$ at node $p$.
procedure FindParent $\langle k \rangle$
    $candidates = \emptyset$
    if $p.stripe_k.parent$ = null then
        $p.stripe_k.parent.cost \leftarrow \infty$
    end if
    for all $n$ in $p.partners$ do
        if $n.stripe_k.cost < p.stripe_k.parent.cost$
        and $n.price < p.money$
        and $n.BM(stripe_k) \geq p.BM(stripe_k)$ then          ▷ $BM$: Buffer Map
            $candidates$.add($n$)
        end if
    end for
    return $candidates$
end procedure

// Handling the assign request from node $p$ for stripe $k$ at node $q$.
upon event ⟨AssignRequest | $k$⟩ from $p$
    if $q.uploadSlots$ has free entries then
        assign a free upload slot to $p$
        Send $ParentResponse(AssignAccepted, k)$ to $p$
    else
        if $q$ has $freeridingChild$ then          ▷ $q.price = 0$
            $lowestMoneyChild \leftarrow freeridingChild$
        else          ▷ $q.price$ = the lowest money of the children
            $lowestMoneyChild \leftarrow$ the child with the lowest money
        end if
        if $p.money > q.price$ then
            assign an $uploadSlot$ to $p$
            Send $Release(k)$ to $lowestMoneyChild$
            Send $ParentResponse(AssignAccepted, k)$ to $p$
        else
            Send $ParentResponse(AssignRejected, k)$ to $p$
        end if
    end if
end event
```

parent. If q's price is greater than or equal to p's money, q declines the request. If q has a free-riding child, q abandons that node as the child with the lowest money.

Handling free-riders. *Free-riders* are nodes that supply less upload bandwidth than claimed. To detect free-riders, we introduce the *free-rider detector* component with *eventual strong completeness* property. By eventual strong completeness property, we mean that, a node that does not have free upload-slots eventually detects all its free-riding children. Nodes identify free-riders through transitive auditing using their children's children (*grandchildren*). To do this, a parent q periodically sends an audit request, about its child p, to p's claimed children. Whenever a grandchild receives a message from q, it checks if p is its parent, and has properly forwarded the stripe(s) it has promised to supply. The grandchild, then, sends back either a positive or negative audit response to q that shows whether these conditions are satisfied or not. However, this model does not solve the *collusion* problem, if a set of nodes cooperate to cheat.

We now show how the eventual strong completeness property is satisfied for the free-rider detector. Assume a node p claims it has u upload-slots, such that m of them are assigned to other nodes and n of them are free upload-slots, i.e., $u = m + n$. Then, p's parent, q, periodically sends audit requests to p's m claimed children. Before the next iteration of sending audit requests, q calculates the sum of (i) the number of audit responses not received before a timeout, (ii) the number of negative audit responses, and (iii) the n free upload-slots. If the sum is more than $M\%$ of u, p is *suspected* as a freerider, where M is a threshold for free-rider suspicion. If p becomes suspected in N consecutive iterations, it is *detected* as a free-rider. For example, if N equals 2, a node is detected as a free-rider if it is suspected on two consecutive iterations of the free-rider detector. The higher is the value of N, the more accurate but slower is the detection.

In a converged tree, for nodes not in the two bottom levels (the trees' leaves), we expect that at least $M\%$ of their upload-slots are meeting their contracted obligation to correctly supply a stripe over that upload-slot. For example, if M is 90%, then node p is suspected as a free-rider, if 10% or more of its upload-slots are either not connected to child nodes or connected to child nodes but do not supply the stream at the requested rate.

After detecting a node as a free-rider, the parent node q, decreases its own price (q's price) to zero and as a *punishment* considers the free-rider node p as its child with the lowest money. On the next bid from another node, q replaces the free-rider node with the new node. Therefore, if a node claims it has more upload-slots than it actually supplies, it will be detected and punished. In a converged tree, many members of the two bottom levels may have no children, because they are the leaves of the trees, thus, the nodes in these levels are not suspected as free-riders.

3.3.2 Mesh overlay

To build a mesh overlay, we keep the definition of the price as it is in Section 3.3.1, but we redefine the money and cost as the following:

1. *Money*: the total number of blocks uploaded to children during the last 10 seconds.

2. *Cost*: the cost of a node is the average distance of the node to the media source via its shortest path from each of its download-slot. We can also add the locality property to the cost, as mentioned in the tree-based approach.

Like the multiple-tree approach, each node periodically sends its money, cost, price and the buffer map to all its *partners*, which are its neighbours in the mesh. The buffer map in the mesh approach shows the list of available blocks in a node buffer. For each of its download-slots, a child node p sends a bid request to those nodes that (i) have a lower cost than the existing parents assigned to download-slots in p, (ii) their price is less than p's money, and (iii) their blocks are ahead of blocks of node p.

Algorithm 2 Handling the parent response message from node q at node p.

```
upon event ⟨ParentResponse | $msg$⟩ from $q$
    if $msg$ is AssignAccepted then
        if $p.downloadSlot$ has free entries then
            $p.parents.add(q)$                                  ▷ add $q$ to the parent list
        else
            $z \leftarrow$ the lowest net profit parent          ▷ the worst parent
            if $(\frac{p.money}{q.cost} - q.price) > (\frac{p.money}{z.cost} - z.price)$ then
                $p.parents.remove(z)$
                $p.parents.add(q)$
                Send $RemoveMeFromYourChildren$ to $z$
            else
                Send $RemoveMeFromYourChildren$ to $q$
            end if
        end if
    end if
end event
```

A parent node, who receives a bid request, accepts it if (i) it has a free upload-slot, or (ii) it has assigned an upload-slot to another node with a lower amount of money. The pseudo-code is similar to the `AssignRequest` in Algorithm 1, with a small difference that in the mesh approach there is no notion of stripes. If a parent re-assigns a connection to a node with more money, it abandons the old child who must then bid for a new upload connection. The parent behaviour in case of having free-rider child is explained later in this section.

When a child node receives the acceptance message from another node, it assigns one of its download-slots to an upload-slot of the parent. However, since a node may send more connection requests than its number of download-slots, it might receive more acceptance messages than it needs (Algorithm 2). In this case, if the child has a free download-slot, it accepts the parent; otherwise, it checks all its assigned parents and finds the one with the lowest net profit or the *worst parent.* If the net profit of the connection to the worst parent is lower than the new parent, the child node releases the connection to the worst parent and accepts the new one; otherwise it ignores the received message.

Handling free-riders. Whenever a node assigns a download-slot to an upload-slot of another node, it sends the address of its current children to its parent. It subsequently informs its parents of any changes in its children. Thus, a parent node knows about its grandchildren, i.e., children's children.

We implement a *scoring* mechanism to detect free-riders, and thus motivate nodes to forward blocks. Each child assigns a score to each of its parents that shows the amount of blocks they have received from their parents in the last 10 seconds. When a child requests and receives a non-duplicate block from a parent within the last 10 seconds, it increments the score of that parent. Hence, the more blocks a parent node sends to its children, the higher score it has among its children. We chose 10 seconds as it is the same as the choking period in BitTorrent [51] and does not unnecessarily punish nodes because of variance in the rate of block forwarding.

Each node periodically sends a score request to its grandchildren, and the grandchildren nodes send back a score response containing the scores of the original node's children. The node sums up the received scores for each child. Free-rider nodes forward a lower number of blocks, thus they have lower scores compared to others.

When a node with no free upload-slots receives a connection request, it sorts its children based on their latest scores. If an existing child has a score less than a predefined threshold, s, then that child is identified as a free-rider. The parent node abandons the free-rider nodes and accepts the new node as its child. If there is more than one child with score less than s, then the lowest score is selected. If all the node's children have a score higher than s, then as explained in the previous section, the parent accepts the connection, if the connecting node has more money than the lowest money of its existing children. When the parent accepts such a connection, it then abandons the child with the lowest money. The abandoned child then has to search for and bid for a new connection to a new parent.

Data dissemination. Each parent node periodically sends its buffer map and its *load* to all its assigned children. The load shows the ratio of the number of blocks that a node has forwarded to the number of its upload connections. A child node, uses the information received from its parents to schedule and pull the required blocks in different iteration. We define a *sliding window* that shows the number of blocks that a child node can request in each iteration. If the playback point of a node is t, and the sliding window size is n, the node can request the blocks from t to $t + n$ in each iteration.

One important question in pulling blocks is the order of requests. There are a number of studies [74, 75] on block selection policies. The main constraint in data dissemination in live media streaming is that the blocks should be received before their playback time. Therefore, a node should pull the missing block with the closest playback time first, that is, blocks should be pulled in-order. Another potential strategy, as used by BitTorrent [51], is to pull the rarest blocks in the system, as this is known to increase aggregate network throughput [44].

We have designed a download policy that attempts to marry the benefits for playback latency of in-order downloading with the improved network throughput of rarest-block policy. We divide the sliding window into two sets: an *in-order set* and a *rare set*. The first m blocks in the sliding window are the blocks in the in-order set and the rest of the blocks of the sliding window are the rare set blocks. As the names of these sets imply, blocks from the in-order set are requested in order and the least popular block (from among the node's partners) is chosen from the rare set. A node selects a block from the in-order set with probability $h\%$ and from the rare set with $(100 - h)\%$, where h is a system parameter. If multiple parents can provide a block, the child node chooses the parent that has the lowest load.

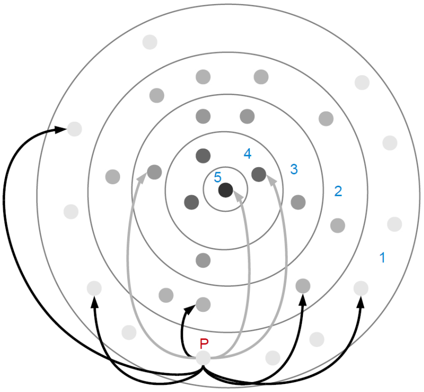


Figure 3.2: Different market-levels of a system, and the similar-view and fingers of p.

3.3.3 The Gradient overlay

The problem with a decentralized implementation of the auction algorithm is the communication overhead in nodes discovering the node with the upload-slot of highest net profit. The centralized auction algorithm assumes that the cost of communicating with all nodes is close to zero. In a decentralized system, however, communicating with all nodes requires flooding, which is not scalable. An alternative approach to compute an approximate solution is to find good upload-slots based on random walks or sampling from a random overlay. However, such solutions typically have slow convergence time, as we show in Section 3.4.

It is important that nodes' partial views enable them to find good matching parents quickly. We use the Gradient overlay [7, 8] to provide nodes with a constantly changing partial view of other nodes that have a similar number of upload-slots. Thus, rather than have nodes explore the whole system for better parent nodes, the Gradient enables nodes to limit exploration to the set of nodes with a similar number of upload-slots.

The Gradient overlay is an overlay network that arranges nodes using a local utility function at each node, such that nodes are ordered in descending *utility values* away from a core of the highest utility nodes [7, 8]. The highest utility nodes are found at the center of the Gradient topology, while nodes with decreasing utility values are found at increasing distance from the center. The Gradient is built by both gossiping and sampling from a random overlay network. Each node maintains a set of neighbours called a *similar-view* containing a small number of nodes whose utility values are close to, but slightly higher than, the utility value of the node. Nodes periodically gossip to exchange and update their similar-views.

Node references stored in the similar-view contain the utility value for the neighbours. In our systems, the utility value of a node is calculated using two factors: (i) a node's upload bandwidth, i.e., node's money and (ii) a disjoint set of discrete utility values that we call *market-levels*. A market-level is defined as a range of network upload bandwidths. For example, in Figure 3.2, we define 5 example market-levels:

mobile broadband (64-127 *Kbps*) with utility value 1, slow DSL (128-511 *Kbps*) with utility value 2, DSL (512-1023 *Kbps*) with utility value 3, fiber (>1024 *Kbps*) with utility value 4, and the media source with utility value 5. A node measures its upload bandwidth, e.g., using a server or trusted neighbour, and calculates its utility value as the market-level that its upload bandwidth falls into. For instance, a node with 256 *Kbps* upload bandwidth falls into slow DSL market-level, so its utility value is 2. Nodes may also choose to contribute less upload bandwidth than they have available, causing them to join a lower market-level.

A node prefers to fill its similar-view with nodes from the same market-level or one level higher. A feature of this preference function is that low-bandwidth nodes only have connections to one another. However, low bandwidth nodes often do not have enough upload bandwidth to simultaneously deliver all stripes/blocks in a stream. Therefore, in order to enable low bandwidth nodes to utilize the spare upload-slots of higher bandwidth nodes, nodes maintain a *finger list*, where each *finger* points to a node in a higher market-level (if one is available). We illustrate the market-levels and fingers in Figure 3.2. Each ring represents a market-level, the black links show the links within the similar-view and the gray links are the fingers to nodes in higher market-levels.

In order for nodes to be able to explore to find new nodes with which to execute our market model, a node constantly updates its neighbours within its market-level. Algorithm 3 is executed periodically by a node p to maintain its similar-view using its *random-view*. The random-view of a node is a random sample of the nodes in the system, which is updated by a peer sampling service, e.g., Cyclon [63].

Algorithm 3 describes how on every round, p increments the age of all the nodes in its similar-view. It removes the oldest node, q, from its similar-view and sends a subset of nodes in its similar-view to q. Node q responds by sending back a subset of its own similar-view to p. Node p then merges the view received from q with its existing similar-view by iterating through the received list of nodes, and preferentially selecting those nodes in the same market-level as p or at most one level higher. If the similar-view is not full, it adds the node, and if a reference to the node to be merged already exists in p's similar-view, p just refreshes the age of its reference. If the similar-view is full, p replaces one of the nodes it had sent to q with the selected node. Moreover, p also merges its similar-view with its own local random-view, in the same way described above. Upon merging, when the similar-view is full, p replaces a node whose utility value is higher than p's utility value plus one.

The fingers to higher market-levels are also updated periodically. Node p goes through its random-view, and for each higher market-level, picks a node from that market-level if there exists such a node in the random-view. If there is not, p keeps the old finger. Using the Gradient overlay as the market maker, the partners of a node are chosen from the similar-view and finger-list. In other words, in Algorithm 1 (line 7), we should replace `partners` with `(similarView ∪ fingers)`.

Algorithm 3 The Gradient overlay construction algorithm.

```
// Run by each node $p$ in each gossiping round.
procedure Round $\langle\rangle$
    $this.similarView.updateAge()$
    $q \leftarrow this.similarView.selectOldest()$
    $this.similarView.remove(q)$
    $pSubView \leftarrow this.similarView.subset()$        ▷ a random subset from $p$'s $similarView$
    Send $ShuffleRequest(pSubView)$ to $q$
end procedure

// Handling the shuffle request.
upon event ⟨ShuffleRequest | $pSubView$⟩ from $p$
    $qSubView \leftarrow this.similarView.subset()$        ▷ a random subset from $q$'s $similarView$
    $UpdateView(this.simialrView, this.randomView, qSubView, pSubView)$
    Send $ShuffleResponse(qSubView)$ to $p$
end event

// Handling the shuffle response.
upon event ⟨ShuffleResponse | $qSubView$⟩ from $q$
    $UpdateView(this.simialrView, this.randomView, pSubView, qSubView)$
end event

// Updating the view.
procedure UpdateView ⟨$simialrView, randomView, sentView, receivedView$⟩
    for all $node_i$ in $receivedView$ do
        if $U(node_i) = U(this)$ or $U(node_i) = U(this) + 1$ then
            if $similarView.contains(node_i)$ then
                $similarView.updateAge(node_i)$
            else if $similarView$ has free entries then
                $simialrView.add(node_i)$
            else
                $node_j \leftarrow sentView.poll()$        ▷ get and remove one entry from $sentView$
                $similarView.remove(node_j)$
                $simialrView.add(node_i)$
            end if
        end if
    end for
    for all $node_i$ in $randomView$ do
        if $U(node_i) = U(this)$ or $U(node_i) = U(this) + 1$ then
            if $similarView$ has free entries then
                $simialrView.add(node_i)$
            else
                $node_j \leftarrow (node_k \in similarView$ such that $U(node_k) > U(this) + 1)$
            end if
            if $(node_j \neq null)$ then
                $similarView.remove(node_j)$
                $simialrView.add(node_i)$
            end if
        end if
    end for
end procedure
```

3.4 Experiments

In this section, we compare the performance of Sepidar and Glive with the state-of-the-art NewCoolstreaming [24] under simulation.

3.4.1 Experimental setup

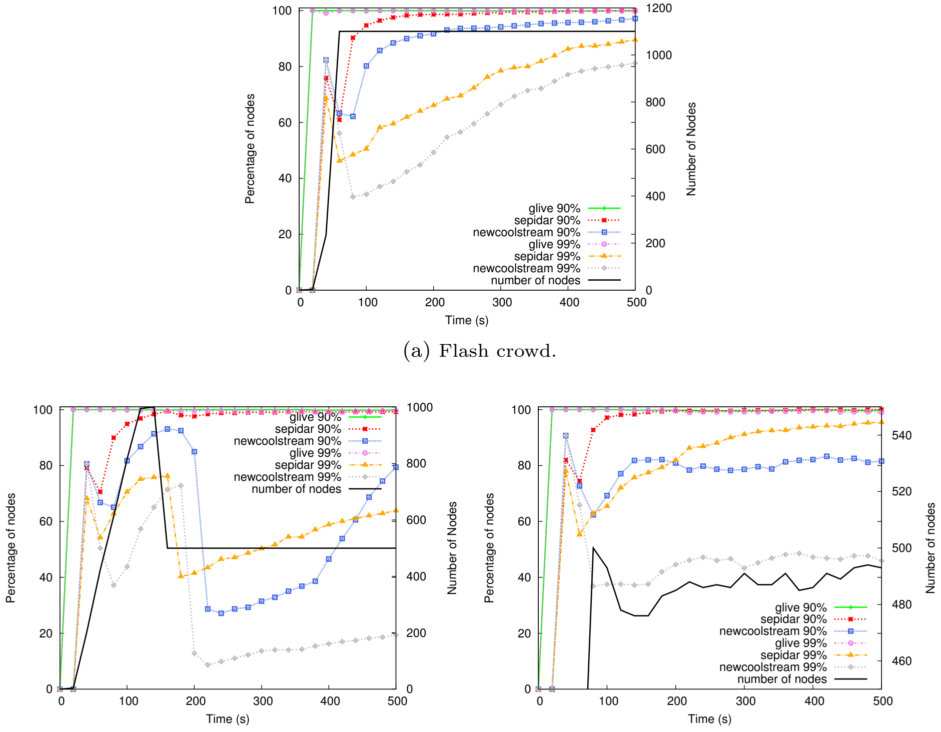
We have used Kompics [76, 77] to implement Sepidar, Glive and NewCoolstreaming. Kompics is a framework for building P2P protocols and it provides a discrete event simulator for simulating them using different bandwidth, latency and churn models. We have implemented NewCoolstreaming based on the system descriptions from [24].

In our experimental setup, we set the streaming rate to $512Kbps$, which is divided into blocks of $16Kb$. Nodes start playing the media after buffering it for 15 seconds, which compares favorably to the 60 seconds of buffering used by state-of-the-art (proprietary) SopCast [78]. The size of similar-view in Sepidar and Glive and the partner list in NewCoolstreaming is 15 nodes. We assume all the nodes have enough download bandwidth to receive the stream with a full rate, and also all the nodes have the same number of download-slots, which is set to 8. To model upload bandwidth, we assume that each upload-slot has available bandwidth of $64Kbps$ and that the number of upload-slots for nodes is set to $2i$, where i is picked uniformly at random from the range 1 to 10. This means that nodes have upload bandwidth between $128Kbps$ and $1.25Mbps$. As the average upload bandwidth of $640Kbps$ is not much higher than the streaming rate of $512Kbps$, nodes have to find good matches as parents in order for good streaming performance. The media source is a single node with 40 upload-slots, providing five times the upload bandwidth of the stream rate. This setting is based on SopCast's requirement that the media source has at least five times the upload capacity of the stream rate [78].

In our simulations, we assume 11 market-levels, such that the nodes with the the same number of upload-slots are located at the same market-level. For example, nodes with two upload-slots ($128Kbps$) are the members of the first market-level, nodes with four upload-slots ($256Kbps$) are located in the second market-level, and the media source with 40 upload-slots ($2.5Mbps$) is the only member of the 11th market-level. Latencies between nodes are modeled using a latency map based on the King data-set [68]. We use the hop count in our experiments to measure the cost function.

In the mesh-based solution, we assume the size of sliding window for downloading is 32 blocks, such that the first 16 blocks are considered as the in-order set and the next 16 blocks are the blocks in the rare set. A block is chosen for download from the in-order set with 90% probability, and from the rare set with 10% probability. In Sepidar, we set $N = 2$ and $M = 50\%$ for the free-rider detector component, and in Glive, we set the threshold of the score, s, to zero.

In this experiment, we measure *playback continuity* and *playback latency*, which combined together reflect the QoS experienced by the overlay nodes. The playback continuity show the percentage of blocks that a node received before their playback time. We consider two metrics related to playback continuity: where nodes have a playback continuity of (i) greater than 90% and (ii) greater than 99%. The playback latency shows the difference in seconds between the playback point (the playback time) of a node and the playback point at the media source.



(a) Flash crowd.

(b) Catastrophic failure.

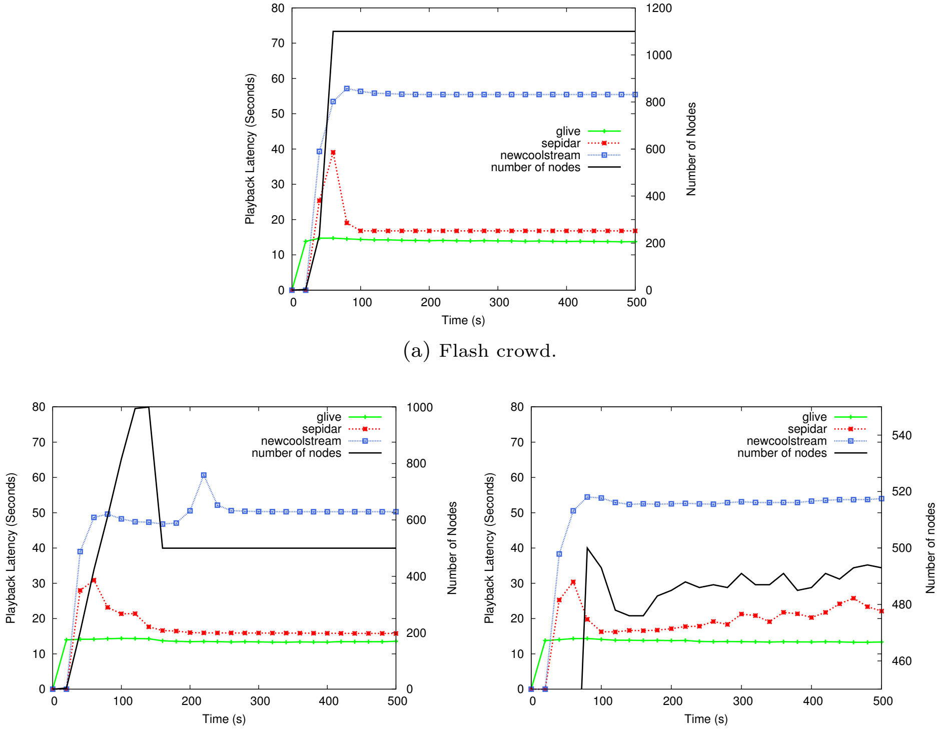
(c) Churn.

Figure 3.3: Playback continuity of the systems in different scenarios.

3.4.2 System performance evaluation

In this section, we compare the playback continuity and playback latency of Sepidar, Glive and NewCoolstreaming in the three scenarios: *flash crowd*, *catastrophic failure*, and *churn*: (i) in the flash crowd, first, 100 nodes join the system following a Poisson distribution with an average inter-arrival time of 100 milliseconds. Then, 1000 nodes join following the same distribution with a shortened average inter-arrival time of 10 milliseconds, (ii) in the catastrophic failure 1000 nodes join the system following a Poisson distribution with an average inter-arrival time of 100 milliseconds. Then, 500 existing nodes fail following a Poisson distribution with an average inter-arrival time 10 milliseconds, and (iii) in the churn scenario 500 nodes join the system following a Poisson distribution with an average inter-arrival time of 100 milliseconds, and then till the end of the simulations nodes join and fail continuously following the same distribution with an average inter-arrival time of 1000 milliseconds.

Figure 3.3 shows the percentage of the nodes that have playback continuity of at least 90% and 99%. We see that all the nodes in Glive receive at least 90% of all the blocks very quickly in all scenarios, while it takes more time in Sepidar. That is because in Sepidar, in the beginning, nodes spend time constructing the trees, while in Glive the nodes pull blocks quickly as soon as at least one of their download-slots is assigned. As we see in Figure 3.3, both Glive and Sepidar outperform NewCoolstreaming in playback



(a) Flash crowd.

(b) Catastrophic failure.

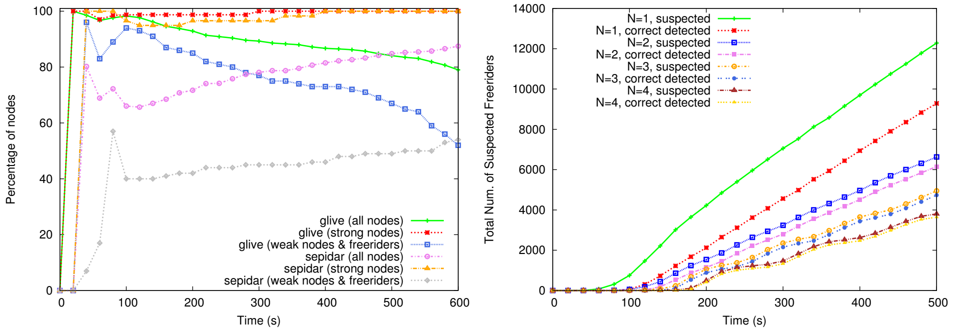
(c) Churn.

Figure 3.4: Playback latency of the systems in different scenarios.

continuity for the whole duration of the experiment in all scenarios. Glive and Sepidar use the Gradient overlay for node discovery. The Gradient overlay arranges nodes based on their number upload bandwidth capacity, and so the neighbours of a node are those with the same upload bandwidth capacity, or slightly higher. This helps the high capacity nodes to quickly discover the media source. In contrast, NewCoolstreaming uses a random overlay, and it takes more time for nodes to find appropriate parents. The result is a higher number of changes in parent connections, causing lower playback continuity in NewCoolstreaming compared to Glive and Sepidar.

As we see in Figure 3.3, the difference between Glive and Sepidar increases, when we measured the percentage of the nodes that receive 99% of the blocks in time. Again, the tree structure used in Sepidar causes this difference. Although, Sepidar has a multiple-tree structure, which is resilient to the failures, it has a lower playback continuity than Glive when nodes crash. In a multiple-tree structure, a node typically receives the blocks of each stripe independently, but if a parent providing a stripe fails, then it loses the blocks from that stripe, while the node is trying to find a new parent for that stripe. However, this problem does not apply to the mesh overlay, because the nodes pull the blocks independently of each other. Therefore, if a node loses one of its parents, it can pull the required blocks from other parents.

Figure 3.4 shows the playback latency of the systems in different scenarios. As we can see, Glive keeps its playback latency relatively constant, close to 15 seconds, which is



(a) Playback continuity in the free-rider scenario.

(b) SEPIDAR freerider detection settings.

Figure 3.5: The systems behaviour in the existence of freeriders.

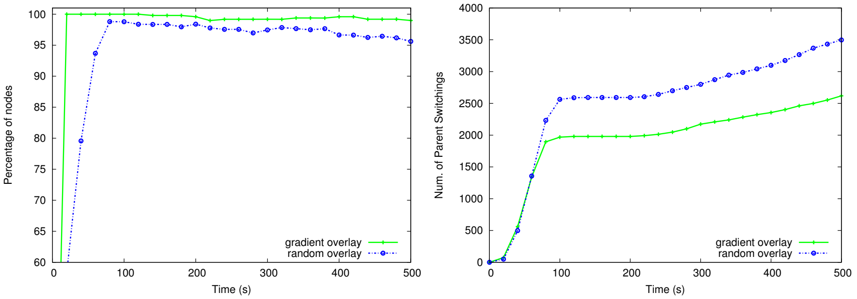
the initial buffering time. The playback latency of SEPIDAR also converges to 15 seconds, but it takes longer to converge than GLIVE. The reason for this delay is, again, the time needed to construct the trees. The playback latency of GLIVE and SEPIDAR, are both less than NewCoolstreaming. In NewCoolstreaming, the higher playback latency is a result of nodes only reactively changing parents when their playback latency is greater than a predefined threshold.

3.4.3 Free-rider detection evaluation

Here, we compare the playback continuity of GLIVE and SEPIDAR in the *free-rider scenario*. In this scenario, 1000 nodes join the system following a Poisson distribution with an average inter-arrival time of 100 milliseconds, such that 30% of the nodes are free-riders, and the total number of upload-slots in the system is less than the total number of download-slots, i.e., $|\mathcal{U}| < |\mathcal{D}|$. The free-riders can be found in any market-level. Figure 3.5a shows the percentage of the nodes that receive 99% of the blocks before their playback time. It shows this value for all the nodes in the system, including the *strong nodes* (top 10% of upload bandwidth nodes), and the free-riders and the *weak nodes* (the bottom 10% of upload bandwidth nodes).

Figure 3.5a shows that all the strong nodes in both systems receive all the blocks in time, however, GLIVE converges faster than SEPIDAR. In GLIVE, we are using the scoring mechanism to find the nodes who contribute less bandwidth than they claim when bidding for connections, while SEPIDAR uses a free-rider detector module that identifies nodes that do not meet their contractual requirement to forward the stream to their child nodes. In GLIVE, in the beginning, a high percentage of weak nodes and free-riders receive all the blocks in time, which shows that free-riders have not been detected yet. That is because nodes need time to update and validate the scores of their parents, and, thus, identify free-riders. Meanwhile, the free-riders use the resources of the system. However, after enough time has passed and the nodes' scores have been updated, the free-riders are detected. Thus, after about 100 seconds the percentage of the free-riders who have a high playback continuity decreases.

As Figure 3.5a shows, after about 600 seconds from the beginning of the experiment, in both GLIVE and SEPIDAR the free-riders and weak nodes receive roughly the same quality



(a) 99% of playback continuity of Glive.

(b) Number of parent switches in Sepidar.

Figure 3.6: The systems behaviour in the Gradient overlay and random overlay.

of stream, that is, they have the same percentage of playback continuity. As the playback continuity of the weak nodes and free-riders keeps decreasing in Glive, we can also see that the playback continuity decreases for all nodes in Glive. After 500 seconds, playback continuity even decreases below Sepidar.

Importantly, as we can see in Figure 3.5a, the existing free-riders in the system have a very low effect on the playback continuity of the strong nodes in Sepidar and Glive. Strong nodes have consistently higher playback continuity than weak nodes and free-riders. This is due to the fact that weak nodes have a lower amount of money compared to strong nodes, which makes them take longer to find good parents. Also, the punishment of free-riders negatively affects their playback continuity. As such, nodes are strongly incentivized to contribute more upload bandwidth through receiving improved relative performance.

Figure 3.5b shows the CDF of the total number of detected free-riders, and the number of nodes that are correctly detected as a free-rider in Sepidar for different settings. As we see, with smaller N, the fraction of nodes that are correctly detected as free-riders decreases. There is a trade-off between accuracy and the speed of the detection. Smaller N gives us faster detection but with less accuracy. In this experiment we assume $M = 50\%$.

3.4.4 Neighbour selection evaluation

In this experiment, we compare the convergence speed of our market model for the Gradient overlay and random overlays. We use the churn scenario in this experiment, as this is the most typical environment for P2P streaming systems on the Internet. Our market model is run using (i) samples taken from the Gradient overlay, where the sampled nodes have similar number of upload-slots, and (ii) samples taken from a random network, where the sampled nodes have random number of upload-slots.

As nodes in the Gradient overlay receive bids from a set of nodes with almost the same money, the difference between received bids is less than the expected difference for the random network. Figure 3.6a shows that in Glive in the case of using the Gradient overlay, more nodes can quickly receive high playback continuity. As such, the Gradient overlay can be said to be a more efficient market maker for our distributed market model than a random overlay. Figure 3.6b shows the CDF of number of parent switches in Sepidar for both overlays against time, and we can see that in the Gradient overlay, the system has a substantially lower number of parent switches.

Chapter 4

Cloud-Assisted P2P Live Streaming

One of the main challenges in P2P live streaming is to provide a good quality of service (QoS) in spite of the dynamic behavior of the network. For live streaming, QoS means high *playback continuity* and low *playback latency*. There is a trade-off between these two properties: it is possible to increase the playback continuity by adopting larger stream buffers, but at the expense of delay. On the other hand, improving playback latency requires that no bottlenecks are present in either the upload bandwidth of the media source and the aggregated upload bandwidth of all nodes in the *swarm*, i.e., the nodes forming the P2P streaming overlay [79, 80].

Increasing the bandwidth at the media source is not always an option, and even when possible, bottlenecks in the swarm have proven to be much more disruptive [79]. An interesting approach to solve this issue is the addition of auxiliary *helpers* to accelerate the content propagation. A helper could be an *active* computational node that participates in the streaming protocol, or it could be a *passive* service, e.g., a storage node, that just provides content on demand. The helpers increase the total upload bandwidth available in the system, thus, potentially reduce the playback latency. Both types of helpers could be rented on demand from an IaaS (Infrastructure as a Service) cloud provider, e.g., Amazon AWS. Considering the capacity and the cost of helpers, the problem consists in selecting the right type of helpers (passive or active), and provisioning their number with respect to the dynamic behavior of the users. If too few helpers are present, it could be impossible to achieve the desired level of QoS. On the other hand, renting helpers is costly, and their number should be minimized.

The P2P-cloud hybrid approach, termed *cloud-assisted* P2P computing, has already been pursued by a number of P2P content distribution systems. For example, Angelcast [80, 81] is a system that dynamically places active helpers in the swarm to optimize data delivery, where the servers cache and forward content to other nodes, and Cloudcast [82] employs a single passive helper and enforces strict limits on the number of (costly) interactions with it that originate from nodes. In addition to these solutions, CloudMedia [83] is another system that predicts the dynamic demands of the users of a P2P video on demand (VoD) system and provides elastic amounts of computing and bandwidth resources on fly with a minimum cost, and finally [84] presents a cloud-assisted system architecture for P2P media streaming among mobile nodes to minimize energy consumption. However, adapting the cloud-assisted approach to P2P live streaming is still an open issue. Live streaming differs from the content distribution for its soft real-time constraints and a higher dynamism in the network, as the users may be zapping between several channels and start or stop to watch a video at anytime [85, 86].

In this chapter, we present CLIVE [87], a novel cloud-assisted P2P live streaming system that guarantees a predefined QoS level by dynamically renting helpers from a cloud infrastructure. We model our problem as an optimization problem, where the constraints are given by the desired QoS level, while the objective function is to minimize the total economic cost incurred in renting resources from the cloud. We provide an approximate, on-line solution that is (i) adaptive to dynamic networks and (ii) decentralized.

CLIVE extends existing *mesh-pull* P2P overlay networks for video streaming [3, 25, 34], in which each node in the swarm periodically sends its data availability to other nodes, which in turn pulls the required chunks of video from the neighbors that have them. The swarm is paired with a CLIVE *manager* (CM), which participates with other nodes in a gossip-based aggregation protocol [57, 88] to find out the current state of the swarm. Using the collected information in the aggregation protocol, the CM computes the number of active helpers required to guarantee the desired QoS. CLIVE includes also a passive helper, whose task is to provide a last resort for nodes that have not been able to obtain their video chunks through the swarm.

A delicate balance between the amount of video chunks obtained from the passive helper and the number of active helpers in the system must be found. Either approaches are associated with an economical cost that depends on (i) the running time for active helpers, (ii) the storage space and number of data requests for passive helpers, and (iii) the consumed bandwidth for both.

4.1 Problem description

We consider a network consisting of a dynamic collection of *nodes* that communicate through message exchanges. Nodes could be *peers*, i.e., edge computers belonging to users watching the video stream, *helpers*, i.e., computational and storage resources rented from an IaaS cloud, and the media source that generates the video stream and starts its dissemination towards peers.

Each peer is uniquely identified by an ID, e.g., composed by an IP address and a port, required to communicate with it. We use the term *swarm* to refer to the collection of all peers. The swarm forms an *overlay network*, meaning that each peer connects to a subset of nodes in the swarm (called *neighbours*). The swarm is highly dynamic: new peers may join at any time, and existing peers may voluntarily leave or crash. Byzantine behavior is not considered in this work.

There are two types of helpers: (i) an *active helper* (AH), which is an autonomous virtual machine composed of one or more computing cores, volatile memory and permanent storage, e.g., Amazon EC2, and (ii) a *passive helper* (PH), which is a simple storage service that can be used to store (PUT) and retrieve (GET) arbitrary pieces of data, e.g., Amazon S3. We assume that customers of the cloud service are required to pay for computing time and bandwidth in the case of AHs, and for storage space, bandwidth and the number of PUT/GET requests in the case of PHs. This model follows the Amazon's pricing model [89, 90].

We assume the source generates a constant-rate bitstream and divides it into a number of *blocks*. A block b is uniquely identified by the real time $t(b)$ at which it is generated. The generation time is used to play blocks in the correct order, as they can be retrieved in any order, independently from previous blocks that may or may not have been downloaded. Peers, helpers and the source are characterized by different bounds on the amount of available download and upload bandwidth. A peer can create a bounded number of download connections and accept a bounded number of upload connections over which blocks are downloaded and uploaded. We define the number of *download-slots* and *upload-slots* of a peer p as its number of download and upload connections, respectively.

Thanks to the replication strategies between different data centers currently employed in clouds [91], we assume that the PH has an unbounded number of upload-slots and can serve as many requests as it receives. Preliminary experiments using PlanetLab and Amazon Cloudfront show that this assumption holds in practice, as adding as many clients as possible has not saturated the upload bandwidth. We assume that peers are approximately synchronized; this is a reasonable assumption, given that some cloud services, like Amazon AWS, are already synchronized and sometimes require the client machines to be synchronized as well.

The goal of CLIVE peers is to play the video with predefined *playback latency*, i.e., the time between the generation of the video and its visualization at the peer, and *playback continuity*, i.e., the percentage of blocks that are correctly streamed to users. To reach this goal, CLIVE is allowed to rent a PH and/or AHs from the cloud. Deciding about which and how much resources to rent from the cloud can be modeled as an optimization problem, where the objective function is to minimize the economic cost and the constraints are the following:

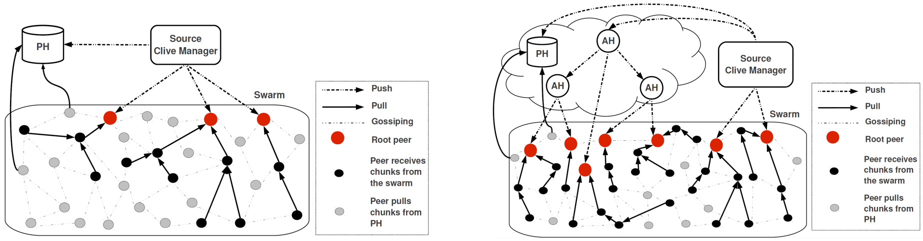
1. the maximum playback latency should be less than or equal to T_{delay}, meaning that if a block b is generated at time $t(b)$ at the source, no peers will show it after time $t(b) + T_{delay}$;
2. the maximum percentage of missing blocks should be less than or equal to P_{loss}.

Note that different formulations of this problem are possible, such as fixing a limit on the amount of money to be spent and trying to maximize the playback continuity. We believe, however, that a company, willing to stream its videos, should not compromise on the users' experience, but rather exploit peers whenever possible and fall back to the cloud when peers are not sufficient.

4.2 System architecture

The basic elements forming CLIVE have been already introduced: (i) the media source, (ii) a swarm of peers, (iii) a single passive helper (PH), and (iv) a number of active helpers (AH). Aim of this section is to discuss how a such diverse collection can be organized and managed. We present two architectural models, illustrated in Figures 4.1a and 4.1b. The *baseline* model (Figure 4.1a) can be described as a P2P streaming protocol, where peers revert to the PH whenever a block cannot be retrieved from other peers. The *enhanced* model (Figure 4.1b) builds upon the baseline, by considering AHs and by providing a distributed mechanism to provision their number and appropriately organizing them.

In the rest of the section, we first discuss the baseline model, introducing the underlying P2P video streaming protocol and showing how it can be modified to exploit a PH. Then,


PH
Source
Clive Manager
Swarm
Push
Pull
Gossiping
Root peer
Peer receives chunks from the swarm
Peer pulls chunks from PH
AH
AH
AH
PH
Source
Clive Manager
Swarm
Push
Pull
Gossiping
Root peer
Peer receives chunks from the swarm
Peer pulls chunks from PH

(a) The baseline model.

(b) The enhanced model.

Figure 4.1: Clive architecture.

we add the AHs into the picture and illustrate the diverse architectural options available when including them.

4.2.1 The baseline model

The baseline model can be seen as a P2P streaming service associated with a media server. We introduce this model as a baseline for comparison and validation of our enhanced architectural model. Note that the idea of augmenting a P2P video streaming application by renting cloud resources is general enough to be applied to several existing video streaming applications. We adopt a *mesh-pull* approach for data dissemination [12], meaning that peers are organized in an unstructured overlay and explicitly ask the missing blocks from their neighbors. Peers discover each other using a gossip-based peer-sampling service [9, 10, 62, 63]; then, the random *partial views* created by this service can be used by any of the existing algorithms to build the streaming overlay [3, 24, 28, 35].

In this model, neighboring peers exchange their data availability with each other, and the peers use this information to schedule and pull the required blocks. There are a number of studies [74, 75] on block selection policies, but here we use the *in-order* policy, as in Coolstreaming [25], where peers pull the missing blocks with the closest playback time first. The baseline model builds upon this P2P video streaming protocol by adding a PH (Figure 4.1a). The source, apart from pushing newly created video blocks to the swarm, temporary stores them on the PH. In order to guarantee a given level of QoS, each peer is required to have a predefined amount of blocks buffered ahead of its playback time, called *last chance window* (LCW), corresponding to a time interval of length T_{lcw}. If a given block has not been obtained from the swarm T_{lcw} time units before the playback time, it is retrieved directly from the PH.

4.2.2 The enhanced model

If the P2P substrate does not suffice, the baseline model represents the easiest solution, but as our experiments will show, this solution could be too expensive, as an excessive number of blocks could end up being retrieved directly from the PH. However, even if the aggregate bandwidth of the swarm may be theoretically sufficient to serve all blocks to all peers, the soft real-time constraints on playback latency may prevent to exploit entirely such bandwidth. No peer must lag behind beyond a specified threshold, meaning that after a given time, blocks will not be disseminated any more. We need to increase the amount of peers that receive blocks in time, and this could be done by increasing the amount of peers that are served as early as possible. The enhanced model pursues this goal by adding a number of AHs to the swarm (Figure 4.1b).

AHs receive blocks from the source or from other AHs, and push them to other AHs and/or to peers in the swarm. To discover such peers, AHs join the peer sampling protocol and obtain a partial view of the whole system. To do so, we use a modified version of Cyclon [63], such that peers exchange their number of upload-slots along with their ID. AH chooses a subset of peers, called *root peers* (Figure 4.1b), from their partial view and establish a connection to them, pushing blocks as soon as they become available. Root peers of an AH are not changed over time, unless they fail or leave the system, or AH finds a peer with more upload-slots than the existing root peers. A peer accepts to be a root peer only for one AH, to avoid to receive multiple copies of the same block.

The net effect is an increase in the number of peers that receive the video stream early in time. The root peers also participate in the P2P streaming protocol, serving a number of peers directly or indirectly. PH still exists in the enhanced model to provide blocks upon demand, but it will be used less frequently compared to the baseline model.

Architecturally speaking, an important issue is how to organize multiple AHs and how to feed blocks to them. There are two possible models: (i) *flat*, where the AHs receive all their blocks directly from the source and then push them to peers in the swarm, acting just as bandwidth multipliers for the source, and (ii) *hierarchical*, where the AHs are organized in a tree with one AH at the root; the source pushes blocks to the root, which pushes them through the tree.

The advantage of the flat model is that few intermediary nodes cause a limited delay between the source and the peers. However, the source bandwidth could end up being entirely consumed to feed the AHs; and more importantly, any communication to the cloud is billed, including the multiple ones from the source to the AHs. We, thus, decided to adopt the hierarchical model, also considering that communication inside the cloud is extremely fast, given the use of gigabit connections, and also free of charge [92].

4.3 System management

One important question in the enhanced model is: how many AHs to add? Finding the right balance is difficult; too many AHs may reduce the PH load, but cost too much, given that they are billed hourly and not only per bandwidth. Too few AHs also increases the PH load, and as we show in the experiments, increases the cost. The correct balance dynamically depends on the current number of peers in the swarm, and their upload bandwidth.

The decision on the number of AHs to include in the system is taken by the CLIVE *manager* (CM), a unit that is responsible for monitoring the state of the system and organizing the AHs. By participating in a decentralized aggregation protocol [57], the CM obtains information about the number of peers in the system and the distribution of upload-slots among them. Based on this information, CM computes the number of AHs that have to be active to minimize the economic cost. Then, depending on the current number of AHs, new AHs may be booted or existing AHs may be shutdown. The CM role can be played either directly by the source, or by one AH.

The theoretical number of AHs that minimize the cost is not so straightforward to compute, because no peer has a global view of the system and its dynamics, e.g., which peers are connected and how many upload-slots each peer has. Instead, we describe a heuristic solution, where each peer runs a small collection of gossip-based protocols, with the goal of obtaining approximate aggregate information about the system. CM joins these gossip protocols as well, and collects the aggregated results. It exploits the collected information to estimate a lower bound on the number of peers that can receive a block either directly or indirectly from AHs and the source, but not from PH. We call this set of peers as *infected peers.* CM, then, uses this information to detect whether the current number of AHs is adequate to the current size of the swarm, or if correcting actions are needed by adding/removing AHs.

In the rest of this section, we first explain how CM estimates the swarm size and the upload-slot distribution, and then we show how it calculates the number of infected peers using the collected information. We also present how CM manages the number of AHs, based on the swarm size and the number of infected peers, and finally we discuss about the impact of T_{lcw} on the system performance and the total cost.

4.3.1 The swarm size and upload slot distribution estimation

All peers in the system, including CM participate in the aggregate computation in Algorithm 4. The procedure `Round` is called periodically by all peers, as well as CM to estimate (i) the current size of the swarm, and (ii) the probability density function of upload-slots available at peers in the swarm.

The size of the current swarm, N_{swarm}, is computed, with high precision, through the aggregation protocol [57]. Nevertheless, knowing the number of upload-slots of all peers is infeasible, due to the large scale of the system and its dynamism. However, we can obtain a reasonable approximation of the probability density function of the number of upload-slots available at all peers.

Assume ω is the actual upload slot distribution among all peers. We adopt Adam2 [93] idea to compute $P_\omega : \mathbb{N} \to \mathbb{R}$, an estimate probability density function of ω. $P_\omega(i)$, then, represents the proportion of peers that have i upload slots w.r.t. the total number of peers, so that $\sum_i P_\omega(i) = 1$. Adam2 is a gossip-based algorithm that provides an estimation of the cumulative distribution function of a given attribute across all peers.

For our algorithm to work, we assume that each peer is able to estimate its own number of upload-slots, and the extreme values of such distribution are known to all and static. Otherwise, a simple mechanism proposed by Haridasan and van Renesse [94] can adjust the set of entries for the case where the extreme values of a variable are unknown. The maximum value is shown by `maxSlot` in Algorithm 4.

Our solution, summarized in Algorithm 4, is based on the gossip paradigm: execution is organized in periodic rounds, performed at roughly the same rate by all peers, during which a push-pull gossip exchange is executed [11]. During a round, each peer p sends a `ShuffleRequest` message to a peer q, and waits for the corresponding `ShuffleResponse` message from q. Information contained in the exchanged messages are used to update the local knowledge about the entire system, which is composed by the following information:

- a *partial view*, or *view* for short, of the network that represents a small subset of the entire population of peers,
- a *slot vector (SV)*, which is used to obtain an approximate and up-to-date information about the attribute distribution,
- a *local value (LV)*, which is used by peers to estimate the network size.

Algorithm 4 Estimating the swarm size and upload slot distribution

```
procedure Init ⟨⟩
    $maxSlot \leftarrow$ the maximum value of slots in system
    for $i \leftarrow 0$ to $maxSlot$ do
        if $i = this.slots$ then
            $this.SV[i] \leftarrow 1$
        else
            $this.SV[i] \leftarrow 0$
        end if
    end for
    if $this = CM$ then
        $this.LV \leftarrow 1$
    else
        $this.LV \leftarrow 0$
    end if
end procedure

// Run by each peer $p$ in each gossiping round.
procedure Round ⟨⟩
    $this.view.updateage()$
    $q \leftarrow this.view.selectOldest()$
    $this.view.remove(q)$
    $pSubView \leftarrow this.view.subset()$
    $pSubView.add(this)$
    Send $ShuffleRequest(pSubView, this.SV, this.LV, this.T_{lcw})$ to $q$
end procedure

// Handling the shuffle request.
upon event ⟨SHUFFLEREQUEST | $pSubView, pSV, pLV, pT_{lcw}$⟩ from $p$
    $qSubView \leftarrow this.view.subset()$
    Send $ShuffleResponse(qSubView, this.SV, this.LV, this.T_{lcw})$ to $p$
    for $i \leftarrow 0$ to $maxSlot$ do
        $this.SV[i] \leftarrow \frac{this.SV[i]+pSV[i]}{2}$
    end for
    $this.LV \leftarrow \frac{this.LV+pLV}{2}$
    $this.T_{lcw} \leftarrow \frac{this.T_{lcw}+pT_{lcw}}{2}$
    $UpdateView(this.view, qSubView, pSubView)$
end event

// Handling the shuffle response.
upon event ⟨SHUFFLERESPONSE | $qSubView, qSV, qLV, qT_{lcw}$⟩ from $q$
    for $i \leftarrow 0$ to $maxSlot$ do
        $this.SV[i] \leftarrow \frac{this.SV[i]+qSV[i]}{2}$
    end for
    $this.LV \leftarrow \frac{this.LV+qLV}{2}$
    $this.T_{lcw} \leftarrow \frac{this.T_{lcw}+qT_{lcw}}{2}$
    $UpdateView(this.view, pSubView, qSubView)$
end event

// Updating the view.
procedure UpdateView ⟨$view, sentView, receivedView$⟩
    for all $node_i$ in $receivedView$ do
        if $view.contains(node_i)$ then
            $view.updateAge(node_i)$
        else if $view$ has free space then
            $view.add(node_i)$
        else
            $node_j \leftarrow sentView.poll()$
            $view.remove(node_j)$
            $view.add(node_i)$
        end if
    end for

end procedure
```

Partial views are needed to maintain a connected, random overlay topology over the population of all peers to allow the exchange of information. We manage the views through the Cyclon peer sampling service [63]. Each view contains a fixed number of *descriptors*, composed by a peer ID and a timestamps. During each round, a peer p identifies the node q with the oldest descriptor in its view, based on the timestamps through `selectOldest` in Algorithm 4. The corresponding descriptor is removed, and a subset of p's view is extracted through procedure `randomSubset`. This subset is sent to q through a `ShuffleRequest` message. The peer that receives the `ShuffleRequest` responses with a `ShuffleResponse` message, that similarly contain a number of descriptors randomly selected from the local view.

Whenever peer p receives a view from q, it merges its own view with the q'view through procedure `UpdateView`. Peer p iterates through the received view, and adds the descriptors to its own view. If its view is not full, it adds the peer, and if a peer descriptor to be merged already exists in p's view, p updates its age, if it is newer. If p's view is full, p replaces one of the peers it had sent to q with the peer in received list. The `poll` method returns and removes the last peer from the list. The net effect of this process is the continuous shuffling of views, removing old descriptors belonging to crashed peers and epidemically disseminating new descriptors generated by active ones. The resulting overlay network, where the neighbors of a peer are the peers included in the partial view, closely resembles a random graph, characterized by extreme robustness and small diameter [63].

LV is a local float value at peers and CM. Initially, it equals zero in all peers, and equals one in CM. In addition to LV, each peer also maintains SV, which is a vector with $\texttt{maxSlot}+1$ entries, such that the index of each entry shows the number of slots, i.e., from 0 to `maxSlot`. Initially, all entries of SV at peer p are set to zero, except the entry that equals the number of p's slots, which is set to one. For example, if $\texttt{maxSlot} = 5$ and the number of p's slots is 2, then the SV of p would be $[0, 0, 1, 0, 0, 0]$.

In each shuffle request, peer p sends its LV and SV, along with its view. When q receives a `ShuffleRequest` message from p, it replies with a message containing a subset of its views, SV, and LV. Peer q, then, goes through the received SV and updates its own SV entries to the average of the values for each entry in both SVs, i.e., $q.SV[i] \leftarrow (q.SV[i] + p.SV[i])/2$. Peer q also updates its LV to $(q.LV + p.LV)/2$. Likewise, peer p updates its SV and LV, when it receives `ShuffleResponse` from q. After a few exchanges, all peers and CM find the distribution of slots in their own SV, such that entry i in the SV shows the probability of peers with i slots. They also compute the swarm size locally as:

$$N_{swarm} = 1/LV \tag{4.1}$$

4.3.2 The number of infected peers estimation

The number of peers that can receive a block from either the swarm, the source or one of the AHs is bounded by the time available to the dissemination process. This time depends on a collection of system and application parameters:

- T_{delay}: No more than T_{delay} time units must pass between the generation of a block at the source and its playback at any of the peers.
- $T_{latency}$: The maximum time needed for a newly generated block to reach the *root peers*, i.e., the peers directly receive blocks from AHs or the source, is equal to $T_{latency}$. While this value may depend on whether a particular root peer is connected

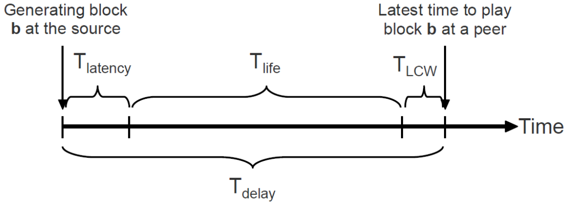


Figure 4.2: Live streaming time model.

to the source or to an AH, we consider it as an upper bound and we assume that the latency added by AHs is negligible.

- T_{lcw}: If a block is not available at a peer T_{lcw} time units before its playback time, it will be retrieved from the PH.

Therefore, a block b generated at time $t(b)$ at the source must be played at peers no later than $t(b) + T_{delay}$, otherwise the QoS contract will be violated. Moreover, the block b becomes available at a root peer at time $t(b) + T_{latency}$, and it should be available in the local buffer of any peer in the swarm by time $t(b) + T_{delay} - T_{lcw}$, otherwise the block will be downloaded from the PH (Figure 4.2). This means that the lifetime T_{life} of a block from the root peer on is equal to:

$$T_{life} = (T_{delay} - T_{latency}) - T_{lcw} \tag{4.2}$$

Whenever a root peer r receives a block b for the first time, it starts disseminating it in the swarm. Biskupski et al. in [73] show that a block disseminated by a pull mechanism through a mesh overlay follows a tree-based diffusion pattern. We define the *diffusion tree* $DT(r, b)$ rooted at a root peer r of a block b as the set of peers, such that a peer q belongs to $DT(r, b)$ if it has received b from a peer $p \in DT(r, b)$.

Learning the exact diffusion tree for all blocks is difficult, because this would imply a global knowledge of the overlay network and its dynamics, and each block may follow a different tree. Fortunately, such precise knowledge is not needed. What we would like to know is an estimate of the number of peers that can be theoretically reached through the source or the current population of AHs.

The block generation execution is divided into rounds of length T_{round}. A block uploaded at round i becomes available for upload to other peers at round $i + 1$. The maximum depth, *depth*, of any diffusion tree of a block over its T_{life} is computed as: $depth = \lfloor T_{life}/T_{round} \rfloor$. We assume that T_{round} is bigger than the average latency among the peers in the swarm. Given *depth* and the probability density function P_ω, we define the procedure `size`(P_ω, *depth*) that executes locally at CM and provides an estimate of the number of peers of a single diffusion tree (Algorithm 5). This algorithm emulates a large number of diffusion trees, based on the probability density function P_ω, and returns the smallest value obtained in this way.

Emulation of a diffusion tree is obtained by the recursive procedure `recSize`(P_ω, *depth*). In this procedure, variable n is initialized to 1, meaning that this peer belongs to the tree. If the depth of the tree is larger than 0, another round of dissemination can be completed. The number of upload-slots is drawn randomly by function `random` from the probability density function P_ω. Variable n is then increased by adding the number of peers that can be reached by recursive call to `recSize`, where the depth is decremented by 1 at each step before the next recursion.

Algorithm 5 Lower bound for the diffusion tree size.

```
procedure size $\langle P_\omega, depth \rangle$
    $min \leftarrow +\infty$
    for $i = 1$ to $k$ do
        $min \leftarrow \texttt{min}(min, recSize(P_\omega, depth))$
    end for
    return $min$
end procedure

procedure recSize $\langle P_\omega, depth \rangle$
    $n \leftarrow 1$
    if $depth > 0$ then
        $slots \leftarrow random(P_\omega)$
        for $i = 1$ to $slots$ do
            $n \leftarrow recSize(P_\omega, depth - 1)$
        end for
    end if
    return $n$
end procedure
```

At this point, the expected number of infected peers, i.e., the total peers that can receive a block directly or indirectly from AHs and the source, but not from PH, N_{exp}, is given by the total number of root peers times the estimated diffusion tree size, $N_{tree} = \texttt{size}(P_\omega, depth)$. The number of root peers is calculated by the sum of the upload slots at the source, shown by $Up(s)$, and AHs, $Up(h)$, minus the number of slots used to push blocks to the AHs themselves, as well as to the PH, which is equal to the number of AHs plus one. Formally,

$$N_{exp} = \left(Up(s) + \sum_{h \in \mathcal{AH}} Up(h) - (|\mathcal{AH}| + 1) \right) \cdot N_{tree} \tag{4.3}$$

where $\mathcal{AH}$ is the set of all AHs.

4.3.3 The management model

We define the cost C_{ah} of one AH in one round (T_{round}) as the following:

$$C_{ah} = C_{vm} + m \cdot C_{block} \tag{4.4}$$

where C_{vm} is the cost of running one AH (virtual machine) in a round, C_{block} is the cost of transferring one block from an AH to a peer, and m in the number of blocks that one AH uploads per round. Since we utilize all the available upload slots of an AH, we can assume that $m = Up(h)$. Similarly, the cost C_{ph} of pulling blocks from PH per round is:

$$C_{ph} = C_{storage} + r \cdot (C_{block} + C_{req}) \tag{4.5}$$

where $C_{storage}$ is the storage cost, C_{req} is the cost of retrieving (GET) one block from PH and r is the number of blocks retrieved from PH per round. C_{block} of PH is the same as in AH. Moreover, since we store only a few minutes of the live stream in the storage, $C_{storage}$ is negligible.

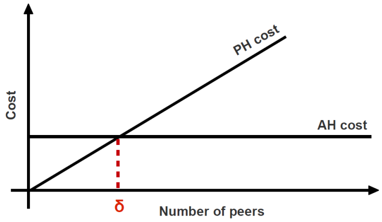


Figure 4.3: Calculating the number of peers that is economically reasonable to serve with PH utilization instead to run an additional AH.

Figure 4.3 shows how C_{ah} and C_{ph} (depicted in Formulas 4.4 and 4.5) changes in one round (T_{round}), when the number of peers increases. We observe that C_{ph} increases linearly with the number of peers (number of requests), while C_{ah} is constant and independent of the number of peers in the swarm. Therefore, if we find the intersection of the cost functions, i.e., the point δ in Figure 4.3, we will know when is economically reasonable to add a new AH, instead of putting more load on PH.

$$\delta \approx \frac{C_{vm} + m \cdot C_{block}}{C_{block} + C_{req}} \tag{4.6}$$

CM considers the following thresholds and regulation behavior:

- $N_{swarm} > N_{exp} + \delta$: This means that the number of peers in the swarm is larger than the maximum size that can be served with a given configuration, thus, more AHs should be added to the system.
- $N_{swarm} < N_{exp} + \delta - Up(h) \cdot N_{tree}$: Current configuration is able to serve more peers than the current network size, thus, extra AHs can be removed. $Up(h) \cdot N_{tree}$ shows the number of peers served by one AH.
- $N_{exp} + \delta - Up(h) \cdot N_{tree} \leq N_{swarm} \leq N_{exp} + \delta$: In this interval the system has adequate resource and no change in the configuration is required.

CM periodically checks the above conditions, and takes the necessary actions, if any. In order to prevent temporary fluctuation, it adds/removes only single AH in each step.

4.3.4 Discussion

T_{lcw} is a system parameter that has an important impact on the quality of the received media at end users, as well as on the total cost. Finding an appropriate value for T_{lcw} is challenging. With a too small T_{lcw} peers may fail to fetch blocks from PH in time for playback, while a too large T_{lcw} increases the number requests to PH, thus, increases the cost. Therefore, the question is how to choose a value for T_{lcw} to achieve (i) the best QoS with a (ii) minimum cost.

Impact of T_{lcw} on the QoS. As we mentioned in Section 4.3.2, each peer buffers a number of blocks ahead of its playback time, to guarantee a given level of QoS. The number of buffered blocks corresponds to a time interval of length T_{lcw}. The length of T_{lcw} should be chosen big enough, such that if a block is not received through other peers, there is enough time to send a request to PH and retrieve the missing block from it in time for playback.

The required time for fetching a block from the PH depends on the round trip time (T_{rtt}) between a peer and the PH, and thus it is not the same for all peers. Therefore, each peer measures T_{rtt} locally, which consists of the latency to send a request to the PH, plus the latency to receive a block at the peer's buffer. A peer should send a request for a missing block to PH no later than T_{rtt} time units before the playback time, otherwise, the retrieved block is useless. Therefore, each peer sets the minimum value of T_{lcw} to T_{rtt}.

While T_{lcw} is a local value at each peer, T_{life}, which is used by the CM to calculate the number of infected peers, depends on the T_{lcw} value (Equation 4.2). Therefore, the CM should be aware of the average T_{lcw} among peers. To provide this information to the CM, all peers, including the CM, participate in an aggregation protocol to get the average of T_{lcw} among all peers. Algorithm 4 shows that in each shuffle, a peer sends its local T_{lcw} to other peers, and upon receiving a reply it updates its T_{lcw} to the average of its own T_{lcw} and the received one.

Impact of T_{lcw} on the cost. Equation 4.5 shows that the cost of PH increases linearly with the number of requests in each round. On the other hand, increasing T_{lcw} increases the PH cost in a round, as peers send more requests to PH. To have a more precise definition of PH cost in Equation 4.5, we replace r, which is the number of received requests at PH, with $\delta \times l$, where l is the normalized value of the system T_{lcw} at CM by the average T_{lcw} (achieved in the aggregation protocol), i.e., $l = \frac{T_{lcw}}{\text{avg}T_{lcw}}$, and δ is the number of peers sending requests to PH in a round. Therefore, Equation 4.6 can be rewritten as follows:

$$\delta \approx \frac{C_{vm} + m \cdot C_{block}}{l \times (C_{block} + C_{req})} \tag{4.7}$$

CM uses the average T_{lcw} to tune T_{lcw} of the system. If CM finds out that changing T_{lcw} can decrease the cost, without violating the QoS, it floods the new value of T_{lcw} to the peers. To do that, it sends the new T_{lcw} to the directly connected peers, and each peer forwards it to all its neighbors, expect the one that it receives the message from. In the flooding path, the peers with smaller T_{lcw} than the received one update their local T_{lcw}. However, if T_{lcw} at a peer is bigger than the received value, it does not change it, as its current T_{lcw} is the minimum required time to get a block from PH. Note that in Equation 4.6, we assumed the local CM T_{lcw} equals the aggregated average of T_{lcw}, thus, $l = 1$.

Figure 4.3 shows the relation between T_{lcw} and PH cost. The higher T_{lcw} is, the steeper the PH cost line is. This means that the cost of PH increases faster with the bigger T_{lcw}, since PH receives more requests in a shorter time. Moreover, smaller values for T_{lcw} push the PH cost line toward the x-axis. For example, if T_{lcw} is zero, i.e., peers never use PH, the PH cost is zero and the line overlaps the x-axis. However, we cannot set T_{lcw} to zero, since we use PH as a backup of the blocks to guarantee the promised QoS.

Increasing the T_{lcw} not only increases the PH cost, but also increases the total system cost. We see in Section 4.3.3, that CM uses two parameters to manage the AHs, (i) the value of δ, and (ii) the number of infected peers. As Equation 4.7 shows, the higher T_{lcw} is, the smaller δ is. On the other hand, according to Equation 4.2, increasing T_{lcw} decreases T_{life}, and consequently, decreases the number of infected peers. Hence, increasing T_{lcw}, decreases both δ, and number of infected peers, and as a result CM adds more AHs to the system, according to the management model in Section 4.3.3, which increases the total cost.

To summarize, we can say that the best value for the system T_{lcw} is the aggregated average T_{lcw}, where $l = 1$. Decreasing the system T_{lcw} below the average T_{lcw}, i.e., $l < 1$, decreases the QoS at peers, as they may fail to fetch blocks from PH before their playback time. On the other hand, although increasing T_{lcw} may increase QoS, it also increases the cost ($l > 1$). Hence, CM never broadcasts new value of T_{lcw} to the system.

4.4 Gossip-based distribution estimation

Monitoring the components of distributed systems is necessary if they are to become self-managing. For example, CLIVE needs an estimate of upload-slot distribution across all peers to compute the number of peers that can be served in a live video streaming service. To the best of our knowledge, Adam2 [93] and Equi-Depth [94] are the only available gossip-based solutions for the distribution estimation problem. In this section, we present a practical gossip-based distribution estimation protocol that has an order of magnitude less overhead than Adam2 [93] and Equi-Depth [94], while obtaining a comparable accuracy.

We consider a network consisting of a collection of nodes that communicate through message exchanges. Each node is uniquely identified by a logical ID. The network is highly dynamic and subject to churn, i.e., new nodes may join at any time, and existing nodes may voluntarily leave or crash. We use $\mathcal{N}(t)$ to denote the population of the network at time t. Byzantine behavior is not considered in this work. We assume that each node in the network has a single local attribute *attr* that represents a local property, e.g., CPU load or disk space. Let $\mathcal{V}$ be the set of all possible values for *attr*, and let $n(v,t)$ be the number of nodes whose attribute is equal to $v \in \mathcal{V}$ at time t. The *global frequency*, *freq*(v,t), of value v at time t is defined as the fraction of nodes with value v at that time:

$$\mathit{freq}(v,t) = \frac{n(v,t)}{\sum_{w \in \mathcal{V}} n(w,t)} \tag{4.8}$$

Our goal is to provide each node with an estimate of *freq*(v,t), for each value v in $\mathcal{V}$, in a completely decentralized way. Algorithm 6 shows our solution. Similar to Algorithm 4, this algorithm is based on the gossip paradigm, and because of similar event handlers, we do not repeat them here, and only explain how it estimates a distribution of a value. In each message exchange, a node sends its attribute value, in addition to its partial view. Nodes store the number of received values in *count*, which is a map that counts the number of times a given value in $\mathcal{V}$ has been received during each of the rounds executed so far. The map *count* is indexed by values in $\mathcal{V}$ and by round number, so that *count*$[v,r]$ counts the number of received messages containing v during round r. We assume that $\mathcal{V}$ is static and known in advance.

At the beginning of round r, *count*$[v,r]$ is initialized to zero for all values $v \in \mathcal{V}$ and the local attribute value is inserted in the `ShuffleRequest` message. Whenever an attribute value v is received in round r, *count*$[v,r]$ is incremented by one. To estimate the global frequency, we consider a small time window into the past (*history*), given by the last α complete rounds. α is a system parameter that is characterized by a trade-off between the accuracy (the larger α, the better) and up-to-datedness (the smaller α, the better) of our estimation. We count in variable *count*$_\alpha[v,r]$ the total number of times that a node has received value v during such period of time:

$$count_{\alpha}[v,r] = \sum_{j=1}^{\alpha} count[v, r-j] \tag{4.9}$$

Our estimate of global frequency of v at round r over the previous α rounds can thus be computed locally at p as the ratio between the number of received messages by p with value v to the total number of messages received by p:

$$est[v,r] = \frac{count_{\alpha}[v,r]}{\sum_{w \in \mathcal{V}} count_{\alpha}[w,r]} \tag{4.10}$$

If there is no bias between the average gossip round-time of all nodes and in the message loss between them, $est[v,r]$ can be considered a good approximation of $freq(v, t(r))$, where $t(r)$ is the approximate time when round r has started:

$$est[v,r] \approx freq(v, t(r)) \tag{4.11}$$

Note that Algorithm 6 has been designed just to illustrate the main characteristics of the algorithm, and many important optimizations are missing. For example, storing the number of messages received more than α rounds ago is not necessary, and the current value of $count_{\alpha}$ can be obtained by the previous value by adding the counters of the current round r and removing those of round $r - \alpha$.

Improvement. The more values a node receives in a round, the faster it converges to the correct estimate. In the explained model, (*baseline-gossip* solution), nodes attach only their single local value to each message exchange. However, as we see in Algorithm 6, in each message exchange, together with this value, a small number of node descriptors are sent in `subView` as well. In the *enhanced-gossip* solution, a node descriptor is a triple (q, t, v) composed by a node ID q, a timestamp t and an attribute value v. In this way, a larger number of values are disseminated around and can be used to obtain a more accurate estimate of the distribution in less time. However, we should notice that this improvement is achieved at the cost of a slight increase in the traffic overhead.

4.5 Experiments

In this section, we evaluate the performance of CLIVE using Kompics [76], a framework for building P2P protocols that provides a discrete event simulator for testing the protocols using different bandwidth, latency and churn scenarios.

4.5.1 Experimental setup

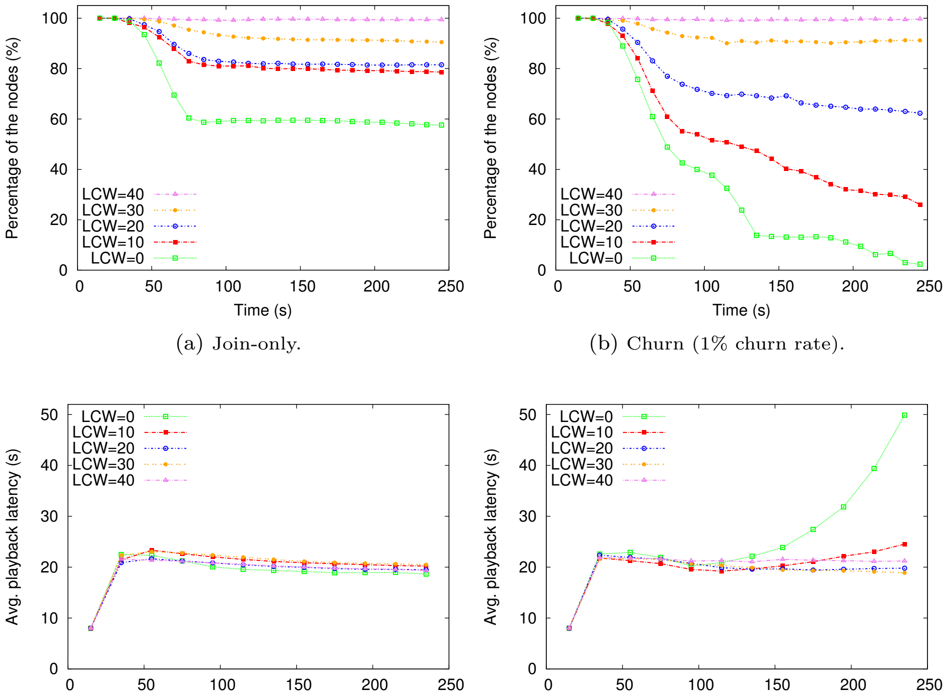
In our experimental setup, we set the streaming rate to 500kbps, which is divided into blocks of 20kb; each block, thus, corresponds to 0.04s of video stream. Peers start playing the media after buffering it for 15 seconds, and T_{delay} equals 25 seconds. We set the bandwidth of an upload-slot and download-slot to 100kbps. Without loss of generality, we assume all peers have enough download bandwidth to receive the stream with the correct rate. In these experiments, all peers have 8 download-slots, and we consider three classes of upload slot distributions: (i) *homogeneous*, where all peers have 8 upload-slots, (ii)

Algorithm 6 Shuffling and estimation algorithm.

```
// Run by each node $p$ in each gossiping round.
procedure Round $\langle\rangle$
    // $\triangleright$ Distribution estimation
    $round \leftarrow round + 1$
    $ComputeEstimate(round)$
    for all $v \in \mathcal{V}$ do
        $count[v, round] \leftarrow 0$
    end for
    // $\triangleright$ Partial view shuffling
    $q \leftarrow this.view.selectOldest()$
    $this.view.remove(q)$
    $pSubView \leftarrow this.view.subset()$
    $pSubView.add(this)$
    Send $ShuffleRequest(pSubView, this.attr)$ to $q$
end procedure

// Handling the shuffle request.
upon event $\langle$SHUFFLEREQUEST $\mid pSubView, pv\rangle$ from $p$
    $count[pv, round] \leftarrow count[pv, round] + 1$
    $qSubView \leftarrow this.view.subset()$
    Send $ShuffleRequest(qSubView, this.attr)$ to $p$
    $UpdateView(this.view, qSubView, pSubView)$
end event

// Handling the shuffle response.
upon event $\langle$SHUFFLERESPONSE $\mid qSubView, qv\rangle$ from $q$
    $count[v, round] \leftarrow count[v, round] + 1$
    $UpdateView(this.view, pSubView, qSubView)$
end event

// Updating the view.
procedure UpdateView $\langle view, sentView, receivedView\rangle$
    for all $node_i$ in $receivedView$ do
        if $view.contains(node_i)$ then
            $view.updateAge(node_i)$
        else if $view$ has free space then
            $view.add(node_i)$
        else
            $node_j \leftarrow sentView.poll()$
            $view.remove(node_j)$
            $view.add(node_i)$
        end if
    end for
end procedure

// Calculating the distribution of the received values.
procedure ComputeEstimate $\langle r\rangle$
    $tot \leftarrow 0$
    for all $v \in \mathcal{V}$ do
        $count_\alpha[v, r] \leftarrow 0$
        for $j \leftarrow 1$ to $\alpha$ do
            $count_\alpha[v, r] \leftarrow count_\alpha[v, r] + count[v, r - j]$
        end for
        $tot \leftarrow tot + count_\alpha[v, r]$
    end for
    for all $v \in \mathcal{V}$ do
        $est[v, r] \leftarrow count_\alpha[v, r]/tot$
    end for
end procedure
```

Table 4.1: Slot distribution in freerider overlay.

Number of slots	Percentage of peers
0	49.3%
1	18.7%
2	8.4%
3-19	5.2%
20	6.8%
Unknown	11.6%



(a) Join-only.

(b) Churn (1% churn rate).

(c) Join-only.

(d) Churn (1% churn rate).

Figure 4.4: The percentage of the peers receiving 99% playback continuity, and the average playback latency with different values of T_{lcw} (measured in number of blocks).

heterogeneous, where the number of upload-slots in peers is picked uniformly at random from 4 to 13, and (iii) *real trace*, based on a study of large scale streaming systems, shown in Table 4.1 [85]. In this model, around 50% of the peers do not contribute in the data distribution. The media source is a single node that pushes blocks to 10 other peers. We assume PH has infinite upload bandwidth, and each AH can push blocks to 20 other peers. Latencies between peers are modeled using a latency map based on the King data-set [68].

In our experiments, we used two failure scenarios: *join-only* and *churn*: (i) in the join-only scenario, 1000 peers join the system following a Poisson distribution with an average inter-arrival time of 10 milliseconds, and after joining the system they will remain till the end of the simulation, and (ii) in the churn scenario, approximately 0.01%, 0.1% and 1% of the peers leave the system per second and rejoin immediately as newly initialized peers [95]. However, unless stated otherwise, we did the experiments with 1% churn rate to show how the system performs in presence of high dynamism.

4.5.2 System performance evaluation

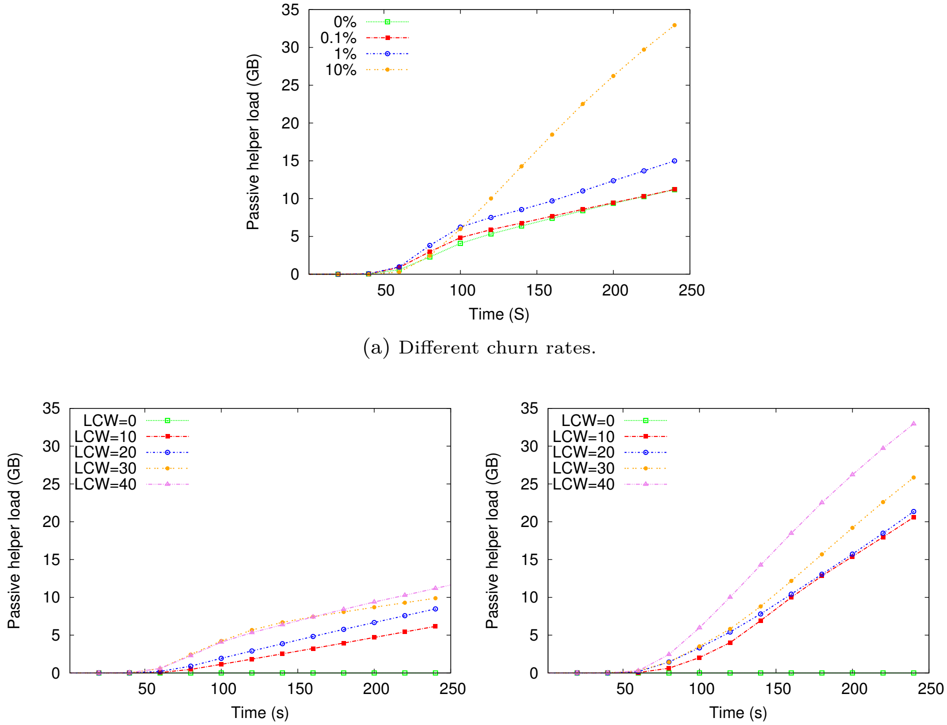
In the first experiment, we evaluate the system behavior with different values for T_{lcw}, measured in number of blocks. In this experiment, we measure playback continuity and playback latency, which combined together reflect the QoS experienced by the overlay peers. Playback continuity shows the percentage of blocks received on time by peers, and playback latency represents the difference, in seconds, between the playback point of a peer and the source.

For a cleaner observation of the effect of T_{lcw}, we use the homogeneous slot distribution in this experiment. Figures 4.4a and 4.4b show the fraction of peers that received 99% of the blocks before their timeout with different T_{lcw} in the join-only and churn scenarios (1% churn rate). We changed T_{lcw} between 0 to 40 blocks, where zero means peers never use PH, and 40 means that a peer retrieves up to block $b + 40$ from PH, if the peer is currently playing block b. As we see, the bigger T_{lcw} is, the more peers receive blocks in time. Although for any value of $T_{lcw} > 0$ peers try to retrieve the missing blocks from PH, the network latency may not allow to obtain the missing block in time. As Figures 4.4a and 4.4b show, all the peers retrieve 99% of the blocks on time when $T_{lcw} = 40$. Given that each block corresponds to 0.04 seconds, $T_{lcw} = 40$ implies 1.6 seconds. The average playback latency of peers is shown in Figures 4.4c and 4.4d. In the join-only scenario, playback latency does not depend on T_{lcw}, while in the churn scenario we can see a sharp increase when T_{lcw} is small.

We also measured PH load or the amount of fetched blocks from PH with different T_{lcw} values and churn rates. Figures 4.5b and 4.5c show the cumulative load of PH in the join-only and churn scenarios, respectively. As we see in these figures, by increasing T_{lcw}, more requests are sent to PH, thus, increasing its load. Figure 4.5a depicts the cumulative PH load over time for four different churn rates and T_{lcw} equals 40 blocks. As the figure shows, there is no big change in PH load under low churn scenarios (0.01% and 0.1%), which are deemed realistic in deployed P2P systems [95]. However, it sharply increases in the presence of higher churn rates (1%), because peers lose their neighbors more often, thus, they cannot pull blocks from the swarm in time, and consequently they have to fetch them from PH.

4.5.3 Economic cost evaluation

In this experiment, we measure the effect of adding/removing AHs on total cost. Note, in these experiments we set T_{lcw} to 40 blocks, therefore, regardless of the number of AHs, all the peers receive 99% of the blocks before their playback time. In fact, AHs only affect the total cost of the service. In Section 4.3.2, we showed how CM estimates the required number of AHs. Figure 4.6 depicts how the number of AHs changes over time. In the



(a) Different churn rates.

(b) Join-only.

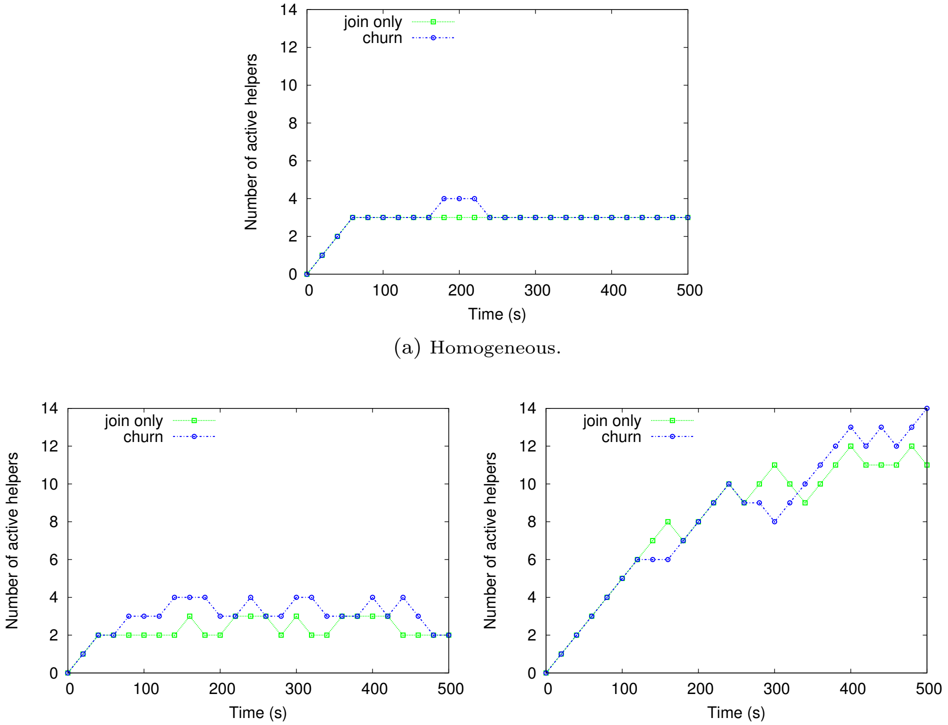
(c) Churn (1% churn rate).

Figure 4.5: The cumulative PH load with different values of T_{lcw} and churn rates.

join-only scenario and the homogeneous slot distribution (Figure 4.6a), the CM estimates the exact value of the peers that receive the blocks on time using the existing resources in the system, and consequently the exact number of required AHs. Hence, as it is shown, the number of AHs will be fixed during the simulation time. However, in the heterogeneous and real trace slot distributions (Figures 4.6b and 4.6c), CM estimation changes over time, and based on this, it adds and removes AHs. In the churn scenario (1% churn rate), CM estimation also changes over the time, thus, the number of AHs fluctuates.

Relatively, we see how PH load changes in different scenarios in the baseline and enhanced models (Figure 4.7). Figure 4.6a shows that three AHs are added to the system in the join-only scenario and the homogeneous slot distribution. On the other hand, we see in Figure 4.7a, in the join-only scenario, with the help of these three AHs (the enhanced model), the load of PH goes down nearly to zero. It implies that three AHs in the system are enough to minimize PH load, while preserving the promised level of QoS. Hence, adding more than three AHs in this setting does not have any benefit and only increases the total cost. Moreover, we can see in the join-only scenario, if there is no AH in the system (the baseline model), PH load is much higher than the enhanced model, e.g., around $90mb$, $40mb$, and $130mb$ per second in the homogeneous, heterogeneous, and real trace, respectively. The same difference appears in the churn scenario.

Figure 4.8 shows the cumulative total cost over the time in different scenarios and slot



(a) Homogeneous.

(b) Heterogeneous.

(c) Real trace.

Figure 4.6: Number of AHs in different settings and scenarios.

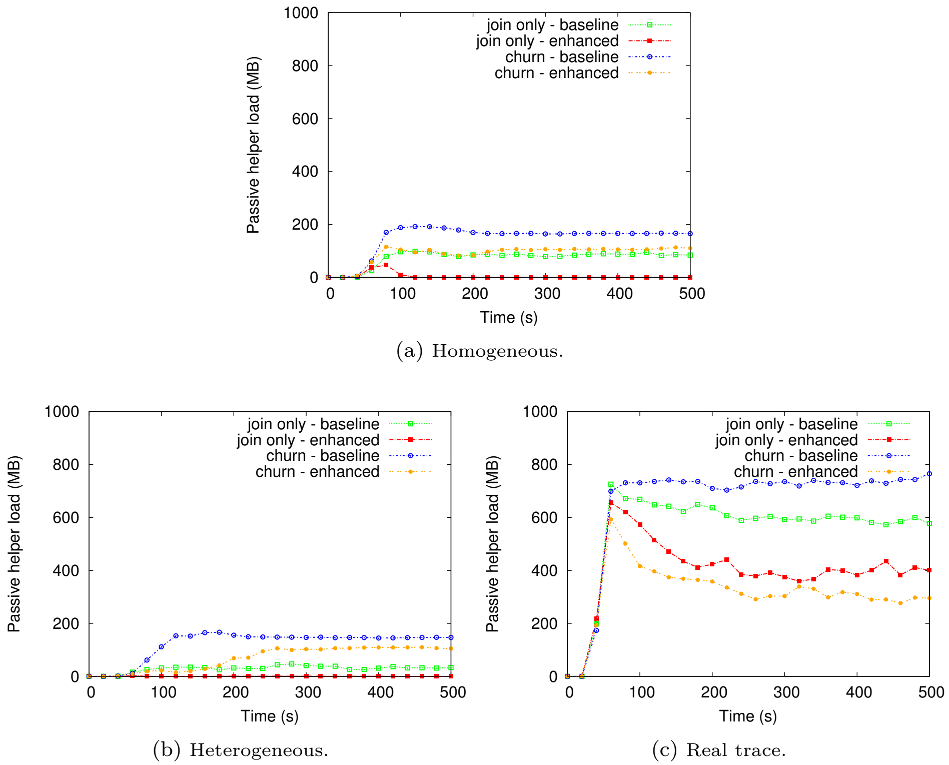
distributions. In this measurement, we use Amazon S3 as PH and Amazon EC2 as AHs. According to the price list of Amazon [89, 90], the data transfer price of S3 is 0.12$ per GB, for up to 10 TB in a month. The cost of GET requests are 0.01$ per 10000 requests. Similarly, the cost of data transfer in EC2 is 0.12$ per GB, for up to 10 TB in a month, but since the AHs actively push blocks, there is no GET requests cost. The cost of a large instance of EC2 is 0.34$ per hour.

Considering the block size of $20kb$ $(0.02mb)$ in our settings, we can measure the cost of PH in Amazon S3 per round (second) according to the Formula 4.5:

$$\begin{aligned} C_{ph} &\approx r \cdot (C_{block} + C_{req}) \\ &\approx \frac{r \times 0.02 \times 0.12}{1000} + \frac{r \times 0.01}{10000} \end{aligned} \tag{4.12}$$

where r is the the number of received requests by PH in one round (second). The cost of storage is negligible. Given that each AH pushes blocks to 20 peers with the rate of $500kbps$ $(0.5mbps)$, then the cost of running one AHs in Amazon EC2 per second according to Formula 4.4 is:

$$\begin{aligned} C_{ah} &= C_{vm} + m \cdot C_{block} \\ &= \frac{0.34}{3600} + \frac{20 \times 0.5 \times 0.12}{1000} \end{aligned} \tag{4.13}$$



(a) Homogeneous.

(b) Heterogeneous.

(c) Real trace.

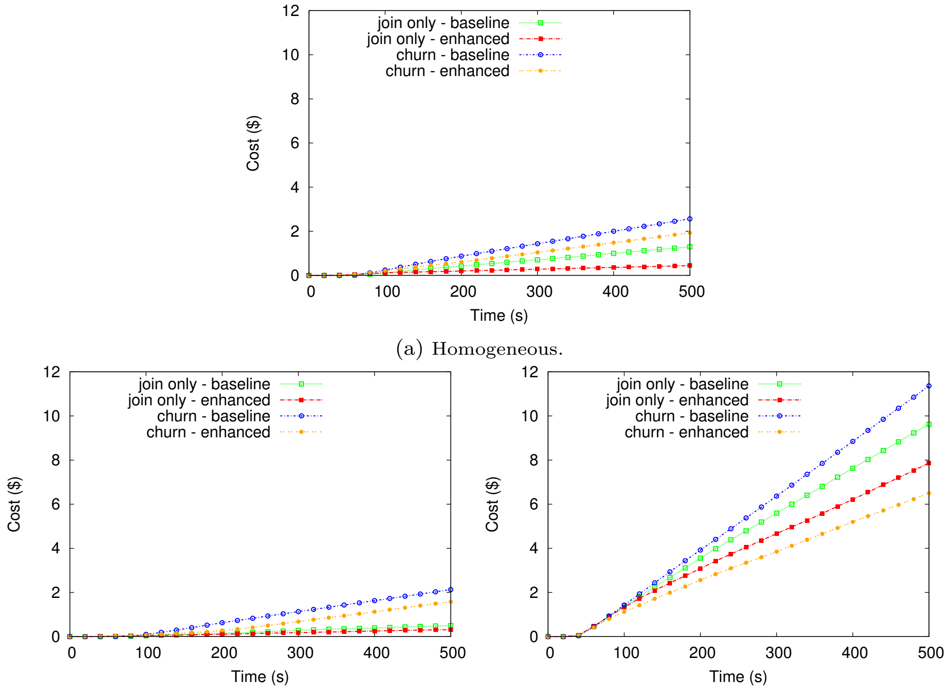
Figure 4.7: PH load in different scenarios with dynamic changes of the number of AHs.

As Figure 4.8 shows, it is clear that adding AHs to the system reduces the total cost, while keeping the QoS as promised. For example, in the high churn scenario (1% churn rate) and the real trace slot distribution the total cost of system after 600 seconds is 24$ in the absence of AHs (baseline model), while it is close to 13$ if AHs are added (enhanced model), which saves around 45% of the cost.

4.5.4 Accuracy evaluation

In this experiment, we evaluate the accuracy of our estimations in form of evaluating the accuracy of upload slot distribution, and the accuracy of estimating the number of infected peers.

Upload slot distribution estimation. Here, we evaluate the estimation of upload slots distribution in the system. We adopt the Kolmogorov-Smirnov (KS) distance [96], to define the upper bound on the approximation error of any peer in the system. The KS distance is given by the maximum difference between the actual slot distribution, ω, and the estimated slot distribution, $E(\omega)$. We compute $E(\omega)$ based on P_ω for different number of slots. In addition to the maximum error, which is determined by a single point (slot) difference between ω and $E(\omega)$, we also measure the average error at each peer as

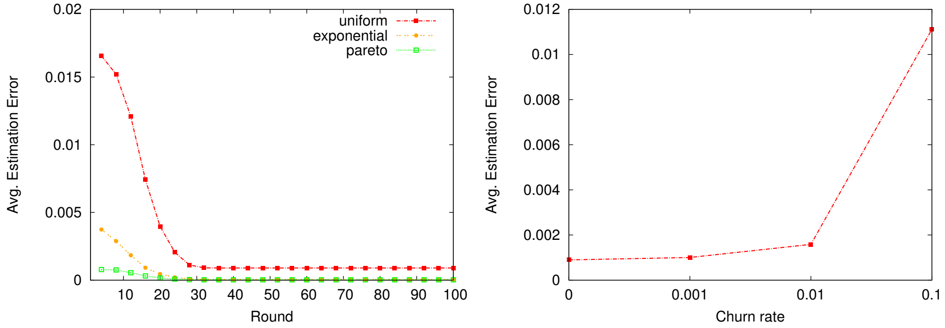


(a) Homogeneous.

(b) Heterogeneous.

(c) Real trace.

Figure 4.8: The cumulative total cost for different setting and scenarios.



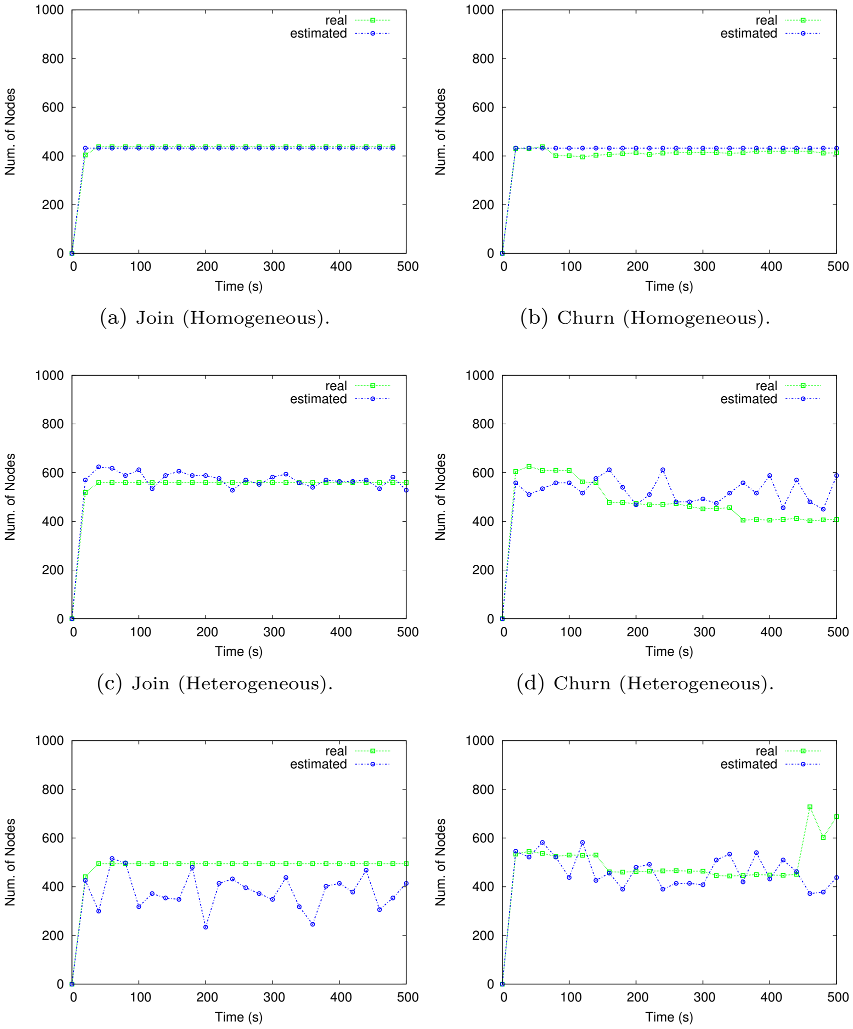
(a) Avg. error in different slot distributions.

(b) Avg. error in different churn rates.

Figure 4.9: Avg. estimation error.

the average error contributed by all points (slots) in ω and $E(\omega)$. The total average error is then computed as the average of these local average errors.

We consider three slot distributions in this experiment: (i) the uniform distribution, (ii) the exponential distribution ($\lambda = 1.5$), and (iii) the Pareto distribution ($k = 5, x_m = 1$). Figure 4.9a shows the average error in three slot distributions, and Figure 4.9b shows how the accuracy of the estimation changes in different churn rates.



(a) Join (Homogeneous).

(b) Churn (Homogeneous).

(c) Join (Heterogeneous).

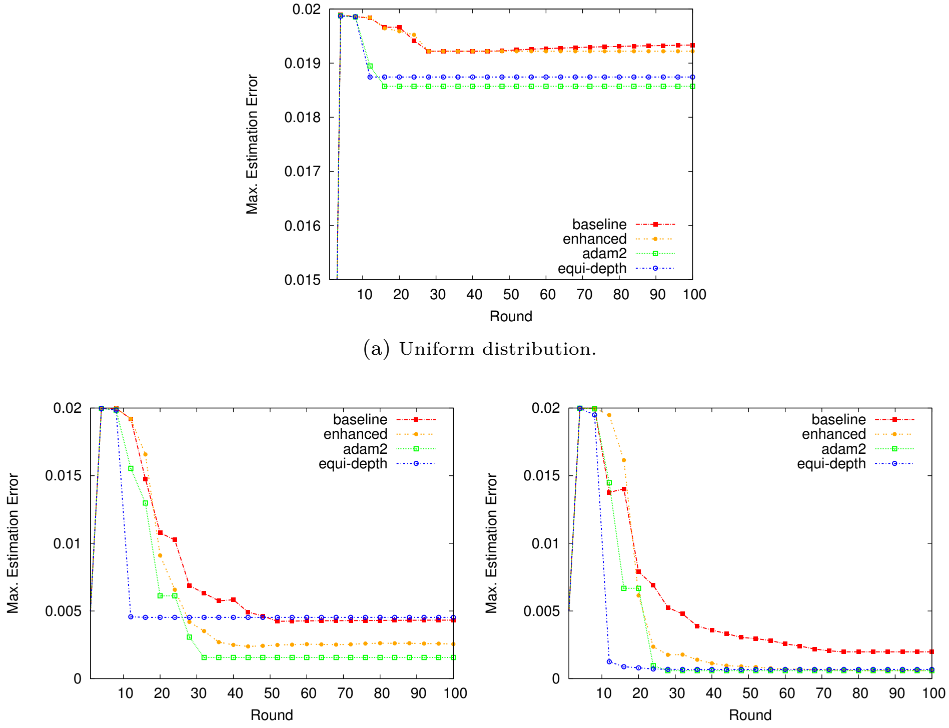
(d) Churn (Heterogeneous).

(e) Join (Real trace).

(f) Churn (Real trace).

Figure 4.10: The real number of infected peers vs. the estimated ones.

Number of infected peers estimation. Here, we evaluate the accuracy of estimating the number of infected peers. Figure 4.10 shows the real number of infected peers and the estimated ones in three upload-slot distributions and in the join and churn scenarios. As shown in the homogeneous and heterogeneous slot distributions, our estimation of the number of infected peers closely fits the real number of such peers. However, in the real trace slot distribution, it may happen that a peer without upload-slot connects directly to the source and prevents other peers to join the system, or on the other hand, a very high upload bandwidth peer joins close to the source and serves many other peers. That



(a) Uniform distribution.

(b) Exponential distribution.

(c) Pareto distribution.

Figure 4.11: Maximum estimation error with different distributions.

is why we see more difference between the real and estimated number of infected peers in the real trace slot distribution.

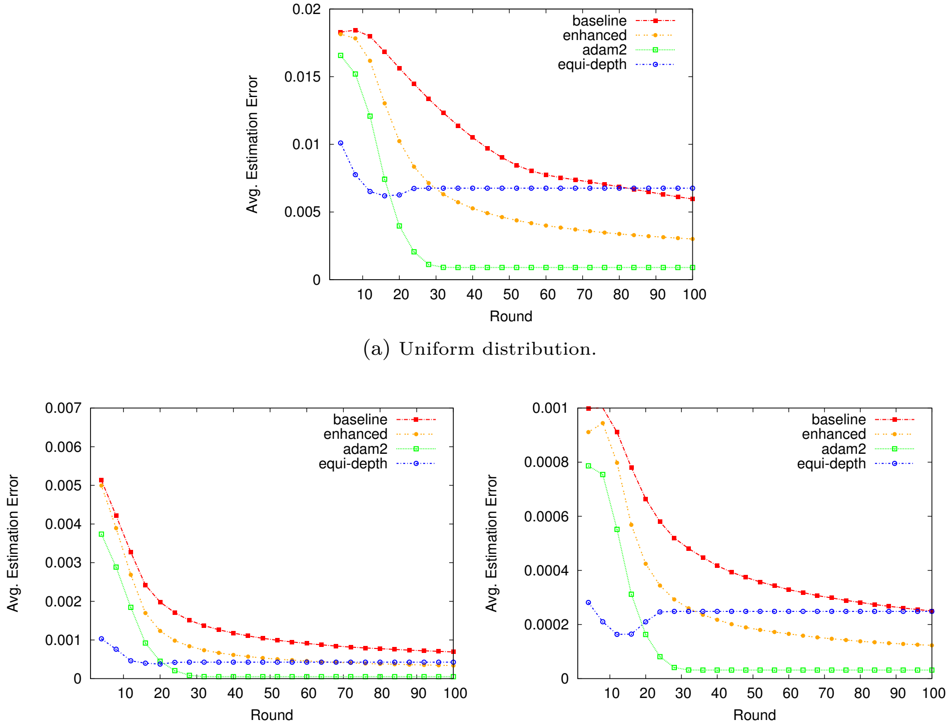
4.5.5 Distribution estimation evaluation

In this section, we evaluate the accuracy of distribution estimation of the baseline-gossip and enhanced-gossip solutions, and compare them with the existing gossip-based solutions Equi-Depth [94] and Adam2 [93].

Here, again we use the KS distance [96] as the maximum difference between an estimated distribution and the original distribution. For each node p and for all values $v \in \mathcal{V}$, we measure the *maximum error* at round r and at node p as the distance between $freq(v, t(r))$ and the estimate $est_p[v, r]$, where $t(r)$ is the approximate time when round r has started:

$$maxErr_p(r) = \max_{v \in \mathcal{V}} |freq(v, t(r)) - est_p[v, r]| \tag{4.14}$$

We measure, then, the *maximum error* at round r as the maximum error over all



(a) Uniform distribution.

(b) Exponential distribution.

(c) Pareto distribution.

Figure 4.12: Average estimation error with different distributions.

nodes:

$$maxErr(r) = \max_{p \in \mathcal{N}(r)} maxErr_p(r) \tag{4.15}$$

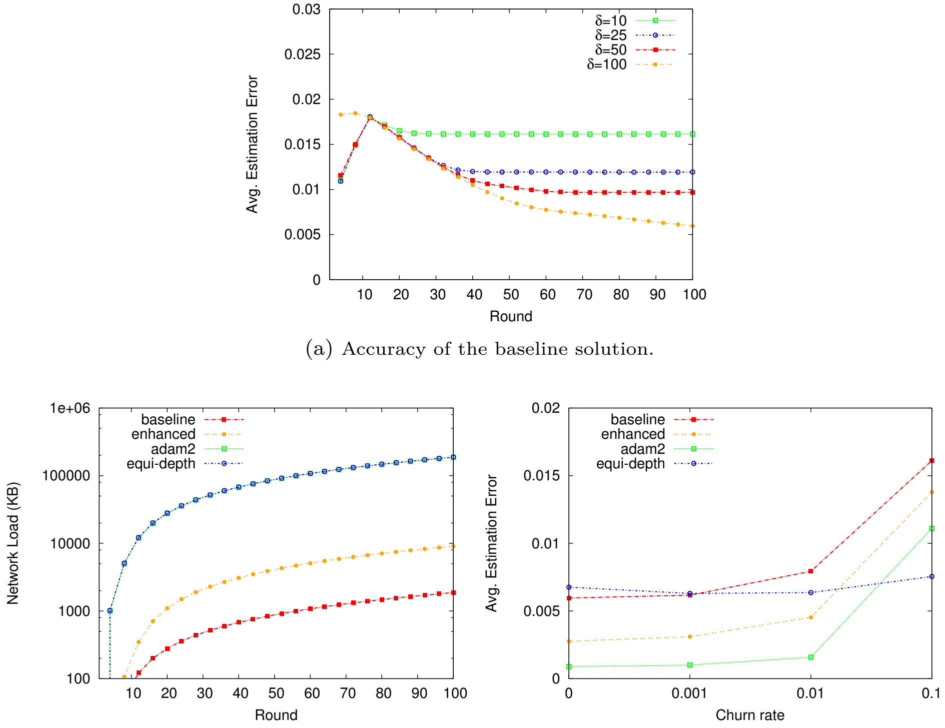
Since the maximum error is determined by a single point difference between $freq(v, t(r))$ and $est_p[v, r]$, it is sensitive to noise, thus, we also measure the average error at each node:

$$avgErr_p(r) = \frac{1}{|\mathcal{V}|} \sum_{v \in \mathcal{V}} |freq(v, t(r)) - est_p[v, r]| \tag{4.16}$$

Our total average error is then calculated as the average of these local average errors:

$$avgErr(r) = \frac{1}{|\mathcal{N}(r)|} \sum_{p \in \mathcal{N}(r)} avgErr_p(r) \tag{4.17}$$

In our experiments, 10,000 nodes participate in the distribution estimation. The nodes join the system following a Poisson distribution with an inter-arrival time of one millisecond. In the experimental setup, for all four protocols, i.e., the baseline-gossip model, the enhanced-gossip model, Equi-Depth and Adam2, the size of partial view is $c = 10$, and the size of the subset views sent in each view exchange is 5. The gossiping round period



(a) Accuracy of the baseline solution.

(b) Traffic overhead.

(c) Average estimation error.

Figure 4.13: The accuracy, traffic overhead and average estimation error of the systems.

for view exchange is set to one second. Unless stated otherwise, we set the history size α equal to 100 in both models. We consider three value distributions in the experiments: the uniform distribution, the exponential distribution ($\lambda = 1.5$), and the Pareto distribution ($k = 5, x_m = 1$). We assume that the size of $\mathcal{V}$ is equal to 100.

Error. We compare the error and the convergence time of the four solutions in the three test distributions in Figures 4.11 and 4.12. As we see Adam2 converges faster than the other solutions with smaller average error. However, Figure 4.11 shows that maximum error of the enhanced-gossip model and Adam2 are very close and better than Equi-Depth and the baseline-gossip. Additionally, we see in Figure 4.12 that the accuracy of the baseline-gossip model increases over time and after a number of rounds its estimation converges to the estimation of Equi-Depth.

The local history size. The local history size α has an important effect on the accuracy of the estimated distribution of the values. If α equals the system life time, the estimated distribution is approximately equivalent to the real distribution. However, in reality nodes need to bound the size of their history. Figure 4.13a shows the average error of the baseline-gossip model for different values of α. In this experiment, the values are distributed uniformly among the nodes. As we see, the bigger α is, the more accurate the results are.

The traffic overhead. Figure 4.13b shows the overhead traffic of the four protocols. In this figure the Y-axis shows the cumulative traffic of 10,000 nodes in logarithmic scale. Given that in the baseline-gossip model nodes only send their own value, the generated traffic is much smaller than the enhanced-gossip model and the two other solutions. In the enhanced-gossip model, the nodes add their values to their descriptors. Therefore, the overhead increases proportionally to the exchanged view size. However, as we see in Figure 4.13b, the enhanced-gossip model traffic overhead is much smaller than Equi-Depth and Adam2, which send the whole vector of values.

Churn. Finally, we compare the average error of the four systems in different churn scenarios. In this experiment, we assume three churn rates, such that approximately 0.1%, 1% and 10% of nodes leave the system per second and rejoin immediately as newly initialized nodes [95]. Figure 4.13c shows that by increasing the churn rate the average error also increases. The figure shows that the enhanced-gossip model and Adam2 compare to Equi-Depth have lower average error in 0.1% and 1% churn rates, however, Equi-Depth shows a better performance in high churn scenarios.

Chapter 5

NAT-Aware Peer Sampling

Peer sampling services (PSS) have been widely used in large scale distributed applications, such as information dissemination [56], aggregation [57], and overlay topology management [7, 58]. A PSS periodically provides a node with a uniform random small sample of live nodes in the system, called *partial view*, where the sample size is typically much smaller than the system size.

PSS' can be implemented using gossip protocols [11, 63] or random walks [61], although random walks are only suitable for static networks with low levels of churn [64]. Gossip-based PSS' can ensure that node descriptors are distributed uniformly at random over all partial views [11]. However, in the Internet, where a high percentage of nodes are behind NATs, these traditional gossip-based PSS' become biased. Nodes cannot establish direct connections to nodes behind NATs (*private nodes*), and private nodes become under-represented in partial views, while nodes that do support direct connectivity (*public nodes*) become over-represented in partial views [64].

Nylon [64] is the state-of-the-art system to present a distributed solution to NAT traversal that uses existing nodes in the PSS to help in NAT traversal. Nylon uses nodes that have successfully established a connection to a private node as partners who will both route messages to the private node and coordinate NAT *hole punching* algorithms [64, 97]. As node descriptors spread in the system through gossiping in Nylon, this creates routing table entries for paths that forward packets to private nodes. However, long routing paths increase both network traffic at intermediary nodes and the routing latency to private nodes. Also, routing paths become fragile when nodes frequently join and leave the system. Finally, hole punching is slow and can take up to a few seconds over the Internet [98].

We address these problems by introduction two solutions: Gozar [9] and Croupier [10]. Gozar is a gossip-based PSS that (i) provides uniform random samples in the presence of NATs, and (ii) enables direct connectivity to sampled nodes by providing a distributed NAT traversal service that requires only a single intermediary hop to connect to a private node. In Gozar, relaying and hole punching are enabled by private nodes finding public nodes who will act as both relay and rendezvous server for them. We call these public nodes the *parents* of the private nodes. For load balancing and fairness, public nodes accept only a small bounded number of private nodes. When references to private nodes are gossiped in the PSS or sampled using the PSS, they include the addresses of their parents nodes. Other nodes, then, can use these parents to communicate with private nodes.

Although relaying can resolve the mentioned problems, it introduces other complexity into PSS protocols: relaying nodes have to maintain routing tables for private nodes, and private nodes have to maintain open mappings in their NAT to relay nodes. Moreover, where the system is distributed, nodes have to discover the relay node(s) responsible for the private node they wish to communicate with. In Croupier we introduce a novel mechanism for exchanging partial views in NATed networks to build a PSS without the use of relaying.

Our main intuition in Croupier is to use two partial views in each node, one for public nodes and one for private nodes, and to have public nodes act as *croupiers*, exchanging public and private views on behalf of private nodes. View exchanges are initiated by all nodes, but only sent to public nodes (the croupiers) who shuffle the views. In order to generate a random sample from the two partial views, our protocol requires that we estimate the ratio of public to private nodes in the system. Public nodes collectively estimate the ratio of public to private nodes by sampling the recent rate of view exchange requests from public and private nodes, respectively. As all nodes send a single view exchange request per round to a random public node and the round time is equal at all nodes (subject to clock skew), we estimate the ratio of public to private nodes using a distributed averaging algorithm based on sampled request rates.

Distributed NAT traversal is embedded in Gozar, and although Croupier is a pure PSS, it can be complemented with a distributed NAT traversal middleware to provide connectivity to private nodes. To reduce the connection latency to private nodes in both systems, we propose an algorithm to minimize private-parent connection delay. We model this problem with an Integer Linear Programming (ILP) framework and present a distributed auction algorithm to solve it.

5.1 Problem description

In this section, we describe the problem of Gozar and Croupier separately, and then explain the NAT traversal problem in both systems. We model a distributed system as a network of autonomous nodes that exchange messages. There is no central point of control in the system and all nodes execute the PSS algorithm. Each node knows its own NAT type, which is either public or private (we present a distributed algorithm to detect the NAT type in Section 5.2). A public node can be communicated with using an IP address that is globally reachable from any other node, while a private node resides behind at least one NAT or firewall, and is not reachable from outside its private network unless it is the private node that initiates contact.

Gozar problem description. The problem Gozar addresses is how to design a gossip-based NAT-friendly PSS that also supports distributed NAT traversal using a system composed of both public and private nodes. The challenge with gossiping is that it assumes a node can communicate with any node selected from its partial view. To communicate with a private node, there are three existing options:

1. Relay communications to the private node using a public relay node,
2. Use a NAT hole-punching algorithm to establish a direct connection to the private node using a public rendezvous node,
3. Route the request to the private node using chains of existing open connections.

For the first two options, we assume that private nodes are assigned to different public nodes that act as relay or rendezvous servers. This leads to the problem of discovering which public nodes act as partners for the private nodes. A similar problem arises for the third option – if we are to route a request to a private node along a chain of open connections, how do we maintain routing tables with entries for all reachable private nodes. When designing a gossiping system, we have to decide on which option(s) to support for communicating with private nodes. There are several factors to consider. How much data will be sent over the connection? How long lived will the connection be? How sensitive is the system to high and variable latencies in establishing connections? How fairly should the gossiping load be distributed over public versus private nodes?

For large amounts of data traffic, the second option of hole-punching is the only really viable option, if one is to preserve fairness. However, if a system is sensitive to long connection establishment times, then hole-punching may not be suitable. If the amount of data being sent is small, and fast connection setup times are important, as in PSS', then relaying is considered an acceptable solution. Hence, Gozar uses relaying. If it is important to distribute load as fairly as possible between public and private nodes, then the third option is attractive. In Section 5.3.1, we show how Gozar PSS works and how private nodes discover public nodes, as their relay servers.

Croupier problem description. The goal of Croupier is to provide a PSS that makes samples of nodes drawn uniformly at random from the set of all nodes in the system, without using relaying or hole-punching. In Croupier, all nodes shuffle only with public nodes, and public nodes, called *croupiers*, exchange nodes views on behalf of private nodes.

In Croupier, we partition the partial view in each node into two separate bounded-size views: *public view* and *private view*. This prevents over-representation of public nodes in partial views, but it introduces the problem of how to generate a uniform random sample from the two views; we cannot just pick a random neighbor from one of either the public or private views, as we need to know the correct proportion of public to private nodes in the system when generating a sample.

That is, we need a distributed algorithm that estimates the ratio of public to private nodes in the system. This ratio may vary both between different systems and over the lifetime of a system, but when a good estimation is available locally at every node, we can use it to sample the correct proportion of nodes from either the public or private view. In Section 5.3.2, we present the Croupier PSS algorithm and explain our distributed ratio estimation solution.

Distributed NAT traversal problem. NAT traversal is implicitly built in Gozar, but not in Croupier. Instead Croupier can be augmented with a distributed NAT traversal middleware that provides connectivity to private nodes using existing public nodes in the system. To make NAT traversal possible, we need to assign one or more public nodes (*parent*) as relay nodes for each private node. Selecting good relay nodes for private nodes is challenging. Nodes may maintain routing tables to nodes that have recently been communicated with [64, 69], or using a distributed-hash table [99].

In Gozar, private nodes choose their parents randomly from the available public nodes in the system. The private nodes, then, cache the addresses of relay nodes in their descriptor. Hence, as node descriptors spread in the system through gossiping, other nodes can initiate connection to private nodes using their parent addresses in their node descriptor. In Croupier, we can also build a distributed NAT traversal on top of the

PSS, by assigning parents to each private node. A question here is that if we can reduce the private-parent delay, or in other words, how private nodes should select their parents to minimize their connection latency. In Section 5.4 we present a distributed auction algorithm to minimize connection latency to private nodes.

5.2 Distributed NAT type identification

Gozar and Croupier require that a node knows its correct NAT type as either public or private. A node's NAT type could be determined by a centralized service, such as a Session Traversal Utilities for NAT (STUN) server [65], but instead we introduce a distributed, minimal NAT type identification protocol that identifies a node as being either public or private. Our protocol can, in principle, be run at any time during system operation, but is typically run once at bootstrap time, as the vast majority of nodes stay either public or private for the duration of their session. When a node's NAT type does not change, the protocol does not need to be run for every session, as the NAT type can be cached across sessions.

The protocol is defined in Algorithm 7, and is run over UDP. Several instances of the protocol can be run in parallel against different public nodes to improve its robustness and reduce its expected completion time. It identifies a node as a public node if (i) it has a globally reachable IP address and is not behind a NAT or firewall, or (ii) if the node's NAT supports the UPnP Internet Gateway Device Protocol (that is, the node can explicitly map a local port to a port on the public interface of its UPnP-enabled NAT, where the NAT has a public IP address). If neither of these two conditions are matched, the node is a private node.

To realize these properties, the protocol executes two tests: (i) a `MatchingIpTest` compares the node-under-test's local IP address with the IP address seen by a public node, and (ii) a `ForwardTest` checks to make sure the node-under-test can receive a packet from a public node to which it has not sent a packet in the last five minutes, where five minutes is assumed higher than the NAT UDP mapping timeout. The tests are executed in parallel over a number of public nodes returned by a *bootstrap server*.

The protocol requires three network messages per run: (i) a `MatchingIpTest` is sent from the node-under-test to a public node returned by the bootstrap server, (ii) this node then inserts the public IP address from which it received the event into a `ForwardTest` event that is sent to a different public node, and (iii) the node that receives the `ForwardTest` event, then, sends that event back to the node-under-test's public IP address. The `ForwardTest` event cannot be sent to any of the public nodes returned by the bootstrap server as the node-under-test's NAT may have an entry in its NATs mapping table to that node's IP address, and the `ForwardTest` event would erroneously pass through the NAT.

If the client receives the `ForwardTest` event and its local IP address matches the IP address seen in `MatchingIpTest`, then the node's NAT type is public. If the IP addresses do not match, then the node is set to private. This case can happen if the node is behind a NAT that has an Endpoint-Independent filtering policy [98]. If the node's NAT has a more restrictive packet filtering policy or the node is behind a firewall, it will not receive the `ForwardTest` event, and its `Timeout` event handler will return that the node is private. The length of the timeout needs to be long enough to prevent false positives, but it can be adjusted upwards if a late `ForwardTest` event is received after the timeout has expired.

Algorithm 7 Distributed NAT type identification.

```
// Executed at client on joining system.
procedure NatTypeIdentificationClient ⟨⟩
    $publicNodes \leftarrow doBootstrap()$
    if $supportsUpnpIGD() = true$ then
        $this.nodeType \leftarrow public$
    else
        for all $node_i$ in $publicNodes$ do
            Send $MatchingIpTest$(publicNodes) to $node_i$
        end for
        After timeToWait Send $Timeout$(publicNodes) to $this$
    end if
end procedure

// Event handler at first public node.
upon event ⟨MatchingIpTest | $publicNodes$⟩ from $client$
    $secondPublicNode$ ←last good public node seen not in $client.publicNodes$
    Send $ForwardTest(client)$ to $secondPublicNode$
end event

// Event handler at second public node.
upon event ⟨ForwardTest | $client$⟩ from $firstPublicNode$
    Send $ForwardTest(client)$ to $client$
end event

// Event handler at client node.
upon event ⟨ForwardTest | $client$⟩ from $secondPublicNode$
    Send $CancelTimeout$ to $this$
    if $this.localIp = client.Ip$ then
        $this.nodeType \leftarrow public$
    else
        $this.nodeType \leftarrow private$
    end if
end event

// Timeout event triggered if no ForwardResp event is received in time.
upon event ⟨Timeout | $publicNodes$⟩ from $this$
    $this.nodeType \leftarrow private$
end event
```

5.3 NAT-aware peer sampling

In this section we present our works on NAT-aware peer sampling in form of two systems: (i) Gozar: NAT-aware peer sampling with one-hop relaying, and (ii) Croupier: NAT-aware peer sampling without relaying.

5.3.1 NAT-aware peer sampling with one-hop relaying

Our implementation of Gozar is based on the tail, push-pull and swapper policies for node selection, view exchange and view selection, respectively (Section 2.2.1). In Gozar, node descriptors are augmented with the node's NAT type (private or public), and each private node connects to one or more public nodes, called *parents*. Private nodes discover potential parents using the PSS, that is, private nodes select public nodes from their partial view and send *partnering* requests to them. When a private node successfully partners with a public node, it adds its parent address to its own node descriptor. As node descriptors spread in the system through gossiping, a node that subsequently selects the private node from its partial view communicates with the private node using one of its parent as a relay server.

Relaying enables faster connection establishment than hole punching, allowing for shorter periodic cycles for gossiping. Short gossiping cycles are necessary in dynamic networks, as they improve convergence time, helping keep partial views updated in a timely manner. However, for distributed applications that use a PSS, such as online gaming, video streaming, and P2P file sharing, relaying is not acceptable due to the extra load on public nodes. To support these applications, the private nodes' parents also provide a rendezvous service to enable applications that sample nodes using the PSS to connect to them using a hole punching algorithm.

Whenever a new node joins the system, it contacts the *bootstrap server* and asks for a list of nodes from the system, and also runs the modified STUN protocol or the protocol explained in Section 5.2 to determine its NAT type. If the node is public, it can immediately add the returned nodes to its partial view and start gossiping with the returned nodes. If the node is private, it needs to find a parent before it can start gossiping. It selects m public nodes from the returned nodes and sends each of them a partnering request. Public nodes only partner a bounded number of private nodes to ensure the partnering load is balanced over the public nodes. Therefore, if a public node cannot act as a parent, it returns a NACK. The private node continues sending partnering requests to public nodes until it finds a parent, upon which the private node can now start gossiping.

Each node in GOZAR maintains a partial view of the nodes in the system. A node descriptor, stored in a partial view, contains the address of the node, NAT type, and the addresses of the node's parents, which are initially empty. When a node descriptor is gossiped or sampled, other nodes learn about the node's NAT type and any parents. Later on, a node can gossip with a private node by relaying messages through the private node's parents. Private nodes proactively keep their connections to their parents open by sending *ping* messages to them periodically. Authors in [100] showed that unused NAT mapping rules remain valid for more than 120 seconds for 70% of connections. In our implementation, the private nodes send the ping messages every 50 seconds to refresh a higher percentage of mapping rules. Moreover, private nodes use the ping replies to detect the failure of their parents. If a private node detects a failed parent, it restarts the parent discovery process.

Each node p periodically executes Algorithm 8 to exchange and update its view. The method `Round` shows that in each iteration, node p first updates the age of all nodes in its view, and then chooses a node to exchange its view with. After selecting a node q, p removes that node from its view. Node p, then, selects a subset of random nodes from its view, and appends to the subset its own node descriptor (the node, its NAT type, and its parents). If the selected node q is a public node, then p sends the `ShuffleRequest` message directly to q, otherwise it sends the `ShuffleRequest` as a relay message to one of q's parents, selected uniformly at random.

Method `SelectANodeToShuffleWith` shows how a node p selects another node to exchange its view with. Node p selects the oldest node in its view (the tail policy), which is either a public node, or a private node that has at least one parent. Once node q receives the `ShuffleRequest`, it selects a random subset of node descriptors from its view and sends the subset back to the requester node p. If p is a public node, q sends the `ShuffleResponse` back directly to it, otherwise it uses one of p's parents to relay the re-

Algorithm 8 GOZAR shuffling algorithm.

```
// Run by each node $p$ in each gossiping round.
procedure Round $\langle\rangle$
    $this.view.updateAge()$
    $q \leftarrow SelectANodeToShuffleWith()$
    $this.view.remove(q)$
    $pSubView \leftarrow this.view.subset()$        ▷ a random subset from $p$'s $view$
    $pSubView.add(this, this.natType, this.parents)$
    if $q.natType$ is public then
        Send $ShuffleRequest(pSubView, this)$ to $q$
    else
        $qParent \leftarrow q.parents.random(1)$
        Send $Relay(ShuffleRequest, pSubView, q)$ to $qParent$
    end if
end procedure

// Select a node to shuffle with.
procedure SelectANodeToShuffleWith $\langle\rangle$
    for all $node_i$ in $this.view$ do
        if $node_i.natType$ = public or ($node_i.natType$ = private and $node_i.parents \neq \emptyset$) then
            $candidates \leftarrow node_i$
        end if
    end for
    $q \leftarrow candidates.selectOldest()$
    Return $q$
end procedure

// Handling the shuffle request.
upon event $\langle$SHUFFLEREQUEST $\mid pSubView, p\rangle$ from $m$        ▷ $m$ can be $p$ or $this.parent$
    $qSubView \leftarrow this.view.subset()$        ▷ a random subset from $q$'s $view$
    if $p.natType$ is public then
        Send $ShuffleResponse(qSubView, this)$ to $p$
    else
        $pParent \leftarrow p.parents.random(1)$
        Send $Relay(ShuffleResponse, qSubView, p)$ to $pParent$
    end if
    $UpdateView(this.view, qSubView, pSubView)$
end event

// Handling the shuffle response.
upon event $\langle$SHUFFLERESPONSE $\mid qSubView, q\rangle$ from $n$        ▷ $n$ can be $q$ or $this.parent$
    $UpdateView(this.view, pSubView, qSubView)$
end event

// Updating the view.
procedure UpdateView $\langle view, sentView, receivedView\rangle$
    for all $node_i$ in $receivedView$ do
        if $view.contains(node_i)$ then
            $view.updateAge(node_i)$
        else if $view$ has free entries then
            $view.add(node_i)$
        else
            $node_j \leftarrow sentView.poll()$        ▷ get and remove one entry from $sentView$
            $view.remove(node_j)$
            $view.add(node_i)$
        end if
    end for
end procedure
```

Algorithm 9 Handling the relay message.

```
upon event ⟨Relay | $msg, view, y$⟩ from $x$
    if $msg$ is $shuffleRequest$ then
        Send $ShuffleRequest(view, x)$ to $y$
    else
        Send $ShuffleResponse(view, x)$ to $y$
    end if
end event
```

Algorithm 10 NAT Traversal at node p to private node q.

```
procedure SendData ⟨$q$, $data$⟩
    if $q.natType$ is public then
        Send $data$ to $q$
    else
        $RVP \leftarrow q.parents.random(1)$
                ▷ Determine hole punching algorithm for the combination of NAT types
        $hp \leftarrow hpAlgorithm(p.natType, q.natType)$
                ▷ Start hole punching at $RVP$ using the hole punching algorithm $hp$.
        $HolePunching(hp, p, q, RVP)$
        Send $data$ to $q$
    end if
end procedure
```

sponse. Node q selects p's relaying node uniformly at random from the list of p's parents. It also updates its view. Node p updates its view when it receives a `ShuffleResponse`.

Method `UpdateView` shows how a node updates its view using the received list of node descriptors. Node p merges the node descriptors received from q with its current view by iterating through the received list, and adding the descriptors to its own view. If its view is not full, it adds the node, and if a node descriptor to be merged already exists in p's view, p updates its age (if more recent). If the view is full, p replaces one of the nodes it had sent to q with the node in received list (the swapper policy).

Algorithm 9 is triggered whenever a parent node receives a `Relay` message from another node. The node extracts the embedded message that can be a `ShuffleRequest` or `ShuffleResponse`, and forwards it to the destination private node. If a client of the PSS, node p, wants to establish a direct connection to a node q, it uses Algorithm 10 that implements the hole punching service. The algorithm shows that if q is a public node, then p sends data directly to q. Otherwise, p selects uniformly at random one of q's parents as a rendezvous node (RVP), and determines the hole punching algorithm using the combination of its own NAT type and q's NAT type RVP [98], and starts the hole punching process through the RVP. After successfully establishing a direct connection, node p sends data directly to q. See [98] for the details of `HolePunching` algorithm.

5.3.2 NAT-aware peer sampling without relaying

Croupier peer sampling algorithm is based on periodic gossip rounds, executed at roughly the same rate by all nodes (subject to clock skew), where neighboring nodes exchange local state. Our shuffling algorithm is based on the tail, push-pull and swapper policies for node selection, view exchange and view selection (Section 2.2.1). The tail policy

Algorithm 11 Croupier shuffling algorithm.

```
// Run by each node $p$ in each gossiping round.
procedure Round ⟨⟩
    $this.view_u.updateAge()$
    $this.view_v.updateAge()$
    $this.M.updateAge()$                         ▷ estimations received from public nodes
    $this.M.removeOld(\gamma)$                     ▷ remove estimations older than $\gamma$
    if $this.natType$ is public then
        $this.E \leftarrow CalcHitsRatio()$              ▷ see Algorithm 12
        $this.c_u = 0, this.c_v = 0$               ▷ initialize new estimations for current round
    end if
    $q \leftarrow this.view_u.selectOldest()$            ▷ oldest node in the public view
    $this.view_u.remove(q)$
    $pPub \leftarrow this.view_u.subset()$
    $pPri \leftarrow this.view_v.subset()$
    $pSubM \leftarrow this.M.subset()$
    if $this.natType$ is public then
        $pPub.add(this)$
    else
        $pPri.add(this)$
    end if
    Send $ShuffleRequest(pPub, pPri, pSubM, this.E)$ to $q$
end procedure

// Handling the shuffle request.
upon event ⟨ShuffleRequest | $pPub, pPri, pSubM, pE$⟩ from $p$
    if $p.natType$ is public then
        $this.c_u \leftarrow this.c_u + 1$
    else
        $this.c_v \leftarrow this.c_v + 1$
    end if
    $qPub \leftarrow this.view_u.subset()$
    $qPri \leftarrow this.view_v.subset()$
    $qSubM \leftarrow this.M.subset()$
    $UpdateView(this.view_u, qPub, pPub)$
    $UpdateView(this.view_v, qPri, pPri)$
    $this.M \leftarrow this.M \cup pSubM \cup \{pE\}$
    Send $ShuffleResponse(qPub, qPri, qSubM, this.E)$ to $p$
end event

// Handling the shuffle response.
upon event ⟨ShuffleResponse | $qPub, qPri, qSubM, qE$⟩ from $q$
    $UpdateView(this.view_u, pPub, qPub)$
    $UpdateView(this.view_v, pPri, qPri)$
    $this.M \leftarrow this.M \cup qSubM \cup \{qE\}$
end event

// Updating the public/private views.
procedure UpdateView ⟨$view, sentView, receivedView$⟩
    for all $node_i$ in $receivedView$ do
        if $view$ contains $node_i$ then
            $view.updateAge(node_i)$
        else if $view$ has free space then
            $view.add(node_i)$
        else
            $node_j \leftarrow sentView.poll()$
            $view.remove(node_j)$
            $view.add(node_i)$
        end if
    end for
end procedure
```

Algorithm 12 Calculates the hits ratio

```
procedure CalcHitsRatio ⟨⟩
    $this.C_u \leftarrow this.C_u \cup this.\{c_u\}$    ▷ keep a local history of public hits
    $this.C_v \leftarrow this.C_v \cup this.\{c_v\}$    ▷ keep a local history of private hits
    $this.C_u.removeOld(\alpha)$    ▷ remove hits older than $\alpha$ from $C_u$
    $this.C_v.removeOld(\alpha)$    ▷ remove hits older than $\alpha$ from $C_v$
    $pubCnt = 0$
    $priCnt = 0$
    for all $u$ in $this.C_u$ do
        $pubCnt \leftarrow pubCnt + u$
    end for
    for all $v$ in $this.C_v$ do
        $priCnt \leftarrow priCnt + v$
    end for
    return $\frac{pubCnt}{pubCnt+priCnt}$    ▷ calculates the local estimation
end procedure
```

involves selecting the oldest node descriptor for shuffling, while the swapper policy involves replacing the node descriptors sent to the other node with the received node descriptors.

Each node p maintains a public view, $view_u(p)$, and a private view, $view_v(p)$, both bounded in size, consisting of a set of node descriptors of public and private nodes, respectively. A node descriptor contains the node's address, its NAT type, and a timestamps storing the number of rounds since the descriptor was created. A node p (either public or private) periodically executes the procedure `Round` in Algorithm 11 to exchange and update both p's views and its ratio estimations in $E_p(\omega)$ (Equations 5.8 and 5.9). `Round` firstly updates the age of both the descriptors in p's views and the estimations in its *neighbour history*, M, which are the estimations that p received from its neighbors in previous gossip rounds. If p is a public node, then, it updates its local ratio estimation $E_p(\omega)$ (Algorithm 12). Finally, the oldest descriptor q (the tail policy) is selected from the public view, $view_u(p)$, and a `ShuffleRequest` is sent to it.

The public node q receives the `ShuffleRequest` containing the following state: (i) a random, bounded subset of the sender p's public and private views, (ii) a random, bounded subset of p's neighbour history, M_p, and (iii) an p's local estimation, $E_p(\omega)$. If p is a private node, $E_p(\omega)$ is empty. The public node q, then, does the following actions: (i) depending on whether the sender of the request is public or private, it increments the public or private shuffle counters, i.e., c_u or c_v, (ii) it updates its private and public views, and (iii) it adds the received estimations to its neighbour history, M_q.

The private and public views are updated in `UpdateView` procedure. A node q merges the received view from p with its existing view by iterating through the received list of nodes. If its view is not full, it adds the node, and if a reference to the node to be merged already exists in its view, q just refreshes the age of its reference. If the view is full, q replaces one of the nodes it sent to p with the selected node (the swapper policy). A `ShuffleResponse` is subsequently sent back to p. Similar to the request, the response includes a bounded, random subset from its public and private views and its ratio estimations. When p receives the `ShuffleResponse`, similar to the `ShuffleRequest` event handler, it updates its private and public views and its estimations.

Sampling and ratio estimation. The procedure `GenerateRandomSample` in Algorithm 13 is called to generate a uniform random sample of nodes from either a public

Algorithm 13 Sampling and ratio estimation at node p.

```
// Generates a random estimation of nodes using the ratio estimation.
procedure GenerateRandomSample ⟨⟩
    $viewChoice \leftarrow$ random real number between 0 and 1.0
    if $viewChoice < estimatePublicPrivateRatio()$ then
        return random entry from $this.view_u$
    else
        return random entry from $this.view_v$
    end if
end procedure

// Returns the estimation of the ratio of public/private nodes.
procedure EstimatePublicPrivateRatio ⟨⟩
    $cnt = 0$
    for all $m$ in $this.M$ do
        $cnt \leftarrow cnt + m$
    end for
    if $this.natType$ is public then
        $result = \frac{cnt+this.E}{this.M.size+1}$        ▷ see Algorithm 11, line 8 for $this.E$
    else
        $result = \frac{cnt}{this.M.size}$
    end if
    return $result$
end procedure
```

or private node. In the following, we assume both a static ratio of public to private nodes and a fixed number of nodes, although, as shown in our evaluation, our estimation algorithm gives good estimations for dynamic ratios. Public nodes $\mathcal{U}$ and private nodes $\mathcal{V}$ make up the set of all nodes $\mathcal{N}$ in the system: $\mathcal{N} = \mathcal{U} \cup \mathcal{V}$. The ratio ω of public to private nodes in the system is defined as:

$$\omega = \frac{|\mathcal{U}|}{|\mathcal{U}| + |\mathcal{V}|}. \tag{5.1}$$

We estimate ω using a decentralized algorithm that is based on three basic assumptions: (i) there should be no bias between the average gossip round-time of public nodes and private nodes, (ii) there should be no bias in message loss between public and private nodes, and (iii) the target of shuffle requests should be chosen uniformly at random among public nodes. Our first and second assumptions imply that the rate of shuffle requests coming from public nodes compared to private nodes is roughly the same as ω. Our third assumption is grounded on the equivalence of our node selection algorithm to Cyclon's [63], which has previously shown that nodes are selected almost uniformly at random.

The estimation of ω uses the relative number of shuffle requests received by Croupiers (public nodes) from other public nodes or private nodes, within a small time window α into the past, called the *local history*. If we assume α is equal to the system lifetime, we can define the number of shuffle requests that all Croupiers in the system receive from public nodes as $\mathbb{C}_u$, and the number of shuffle requests all Croupiers receive from private nodes as $\mathbb{C}_v$. For each Croupier i, its local public and private shuffle request counts are defined as C_{ui} and C_{vi}, respectively. That is the system-wide shuffle request counts are defined as the sum of local shuffle counts:

$$\mathbb{C}_u = \sum_{i \in \mathcal{U}} C_{ui} \text{ and } \mathbb{C}_v = \sum_{i \in \mathcal{U}} C_{vi} \tag{5.2}$$

The estimation of the ratio of public to private nodes, $E(\omega)$, can now be calculated as the ratio of the number of shuffle requests from public nodes to the number of shuffle requests from all nodes:

$$E(\omega) = \frac{\mathbb{C}_u}{\mathbb{C}_u + \mathbb{C}_v} \tag{5.3}$$

Assuming our first and second assumptions hold, over all public nodes in the system, ω is roughly equal to $E(\omega)$:

$$\omega \approx E(\omega) \tag{5.4}$$

As $E(\omega)$ is not available at any individual node, each public node i maintains its local part of the estimation E_i by updating its local counts C_{ui} and C_{vi} within the last time window α:

$$C_{ui} = \sum_{t=0}^{\alpha} c_{ui}(t) \text{ and } C_{vi} = \sum_{t=0}^{\alpha} c_{vi}(t) \tag{5.5}$$

where c_{ui} and c_{vi} are the number of received requests from public and private nodes in each shuffle round, respectively. A public node i, then, calculates the local estimation E_i as:

$$E_i = \frac{C_{ui}}{C_{ui} + C_{vi}} \tag{5.6}$$

As α approaches the system lifetime, the average of the local estimations is approximately equivalent to our global estimation:

$$E(\omega) \approx \frac{\sum\limits_{i \in \mathcal{U}} E_i}{|\mathcal{U}|} \tag{5.7}$$

Each public node i stores its own local estimation E_i, and its neighbour history M_i, which is a set of local estimations shared by other public nodes. All local estimations by public nodes should be independent of each other as shuffle requests should be uniformly distributed among public nodes. Public nodes disseminate to their neighbours (public and private) both their own local estimation E_i, as well as a subset of their local neighbor history M_i. All estimations can be shared in a simple dissemination protocol to both private and public nodes, but, for efficiency, we piggy-back these estimations on `ShuffleRequest` and `ShuffleResponse` messages (Algorithm 11, lines 21 and 36).

Estimates in neighbour histories, M, contain timestamps that are incremented at every gossip round. When two estimations for the same node are available, the older estimation is replaced by the newer estimation, and old estimations with a timestamps higher than a configurable parameter γ are removed every gossip round. For every shuffle request and shuffle response, we bound the number of estimations that are shared to a subset of M to prevent the size of messages growing for increasing system size.

Given local and neighbour estimations, a public node i estimates ω as the average of both its local estimation E_i and its neighbour history M_i:

$$E_i(\omega) = \frac{\sum\limits_{n \in M_i} E_n + E_i}{|M_i| + 1} \tag{5.8}$$

In contrast, a private node i has no local estimation E_i (as it does not receive shuffle requests), so it estimates ω as the average of its cached estimations from public nodes M_i:

$$E_i\left(\omega\right) = \frac{\sum\limits_{n \in M_i} E_n}{|M_i|} \tag{5.9}$$

Both Equations 5.8 and 5.9 are defined in the method `EstimatePublicPrivateRatio` of Algorithm 13. In Section 5.5, we show how the quality of the estimations depends on how stable the public/private ratio is, and how well tuned α and γ are to the rate of change of the ratio.

5.3.3 Discussion

Table 5.1 summarizes the similarities and differences between Gozar and Croupier. As shown, both systems use the same policies for node selection, view propagation, and view selection. On the other hand, the main difference between the two systems is the built-in NAT traversal capability in Gozar, which is missing in Croupier. However, as we present in Section 5.4, Croupier can also be complemented with NAT traversal. The two systems can be used interchangeably. However, in applications that require connectivity to private nodes, Gozar is a better choice; otherwise, Croupier is more performant, in particular, in highly dynamic systems, where the churn rate is high. Moreover, Croupier produces less traffic in the network, as compared to Gozar.

Table 5.1: Gozar vs. Croupier.

System	PSS Policies	Services	Robustness in Churn
Gozar	(tail, push-pull, swapper)	PSS + NAT traversal	weak
Croupier	(tail, push-pull, swapper)	PSS	strong

5.4 NAT traversal middleware

To provide connectivity with private nodes in Gozar and Croupier, each private node connects to one or more public nodes, as their *parents*, and parent nodes play as relay/rendezvous nodes for the assigned private nodes. To have a short connection latency to private nodes, the private nodes try to improve the connection delay to their parents over time, by switching from their existing parents to closer public nodes, if they find any.

Each private node has a number of *child-slots* that defines the number of parents that the private node is willing to have, and the number of children that a public node is able to accept is mentioned with its *parent-slots*. We show the set of all child-slots with $\mathcal{C}$, and the set of all parent-slots with $\mathcal{P}$. We say that two child-slots i and i' are *similar*, if they both belong to the same private node. For each child-slot i, the set of all child-slots similar to i is called the *similarity class* of i, and is denoted by $\mathcal{M}_{\mathcal{C}}(i)$. Likewise, two parent-slots j and j' are similar, if they belong to the same public node. $\mathcal{M}_{\mathcal{P}}(j)$ shows the similarity class of parent-slot j.

A connection between a child-slot i and a parent-slot j is shown as a pair (i, j), and it is associated with a communication delay d_{ij}. We assume the connections are symmetric, i.e., $d_{ij} = d_{ji}, \forall i \in \mathcal{C}, j \in \mathcal{P}$. We want to assign all the child-slots to different parent-slots. This problem can be represented as an *assignment problem* [70].

We define an *assignment* $\mathcal{S}$ as a set of pairs (i, j) such that:

1. For all $(i, j) \in \mathcal{S}$, $i \in \mathcal{C}$ and $j \in \mathcal{P}$.
2. For each $i \in \mathcal{C}$, there is at most one pair $(i, j) \in \mathcal{S}$.
3. For each $j \in \mathcal{P}$, there is at most one pair $(i, j) \in \mathcal{S}$.

A *complete assignment* $\mathcal{A}$ is an assignment containing $|\mathcal{C}|$ pairs, such that each $i \in \mathcal{C}$ is assigned to a different $j \in \mathcal{P}$. Our goal is to find a complete assignment $\mathcal{A}$ over all assignments $\mathcal{S}$ that minimizes the total delay. We can formulate this problem in the Integer Linear Programming (ILP) framework [5], as the following:

$$\text{minimize} \quad \sum_{i=1}^{|\mathcal{C}|} \sum_{\{j|(i,j)\in\mathcal{A}\}} d_{ij}x_{ij} \tag{5.10}$$

subject to

$$\sum_{\{j|(i,j)\in\mathcal{A}\}} x_{ij} = 1, \quad \forall i = 1, 2, \cdots, |\mathcal{C}| \tag{5.11}$$

$$\sum_{\{i|(i,j)\in\mathcal{A}\}} x_{ij} \leq 1, \quad \forall j = 1, 2, \cdots, |\mathcal{P}| \tag{5.12}$$

$$x_{ij} \in \{0, 1\}, \ \forall i = 1, 2, \cdots, |\mathcal{C}|, \forall j = 1, 2, \cdots, |\mathcal{P}| \tag{5.13}$$

$$\forall (i, j), (i', j') \in \mathcal{A}, i' \in \mathcal{M}_{\mathcal{C}}(i) \Rightarrow j' \notin \mathcal{M}_{\mathcal{P}}(j) \tag{5.14}$$

$$\forall (i, j), (i', j') \in \mathcal{A}, j' \in \mathcal{M}_{\mathcal{P}}(j) \Rightarrow i' \notin \mathcal{M}_{\mathcal{C}}(i) \tag{5.15}$$

where $x_{ij} = 1$ if a child-slot i is assigned to a parent-slot j, and $x_{ij} = 0$, otherwise. Constraint 5.11 requires that every child-slot i is assigned to one parent-slot, and Constraint 5.12 requires to ensure that each parent-slot can be assigned to at most one child-slot. Constraint 5.14 mentions that if i and i' are similar, $i' \in \mathcal{M}_{\mathcal{C}}(i)$, and if i has already assigned to a parent-slot j, then i' cannot be assigned to a parent-slot j', where j and j' are similar. The last constraint is the same as Constraint 5.14, but from the parent-slots perspective. In another word, these two last constraints say that we cannot create more than one connection between a private node and a public node.

In the rest of this section, we first briefly sketch a possible centralized solution with the auction algorithm [4, 5], and then we present a distributed model of the auction algorithm to solve this problem at large scale.

5.4.1 Centralized solution

In our problem, $|\mathcal{C}|$ child-slots compete to be assigned to some parent-slots among the set of $|\mathcal{P}|$ available parent-slots. Like ordinary auctions, the bidders, i.e., the child-slots, progressively increase their bid for the objects, i.e., parent-slots, in a competitive process.

A matching between a child-slot i and a parent-slot j is associated with a *profit* a_{ij}, and the goal of the auction is to maximize the total profit for all the matchings:

$$\sum_{i=1}^{|\mathcal{C}|}\sum_{j\in\mathcal{P}} a_{ij}x_{ij} \tag{5.16}$$

where $x_{ij} = 1$, if i is assigned to j, otherwise $x_{ij} = 0$, and a_{ij} is calculated as:

$$a_{ij} = \frac{1}{d_{ij}} \tag{5.17}$$

As Equation 5.17 mentions, by decreasing the connection delay between i and j, d_{ij}, we achieve more profit, a_{ij}. Each parent-slot j is associated with a *price* p_j, which is zero in the beginning (while it is unassigned), and is increased in auction iterations after accepting new *bids* from child-slots. A child-slot i measures the *net profit*, v_{ij}, of each parent-slot j as the following:

$$v_{ij} = a_{ij} - p_j \tag{5.18}$$

The auction algorithm proceeds in iterations, and in each iteration it creates one assignment $\mathcal{S}$, such that the net profit of each connection under the assignment $\mathcal{S}$ is maximized. In each iteration, the algorithm updates the price of all parent-slots in the assignment $\mathcal{S}$. If all the child-slots of an assignment $\mathcal{S}$ are assigned, we have a complete assignment $\mathcal{A}$, and the algorithm terminates. Otherwise, the algorithm starts the next iteration by finding parent-slots that offers maximal net profit for unassigned child-slots. Note that at the beginning of each iteration, the net profit of each connection (Equation 5.18) under the assignment $\mathcal{S}$ should be maximum.

The bidding and assignment phases of our auction algorithm are as follows:

- **Bidding phase:** In the bidding phase, each unassigned child-slot i under the assignment $\mathcal{S}$ finds the parent-slot j^* that has the highest net profit:

  $$v_{ij^*} = \max_{j\in\mathcal{P}-\mathcal{Q}} v_{ij} \tag{5.19}$$

  where $\mathcal{Q}$ denotes the set of parent-slots that has been assigned to child-slots in the similarity class of i, $M_{\mathcal{C}}(i)$. It means that child-slot i cannot connect to parent-slot j, where there has been another connection between child-slot $a \in M_{\mathcal{C}}(i)$ and parent-slot $b \in M_{\mathcal{P}}(j)$. To measure the amount of the bid, the child-slot i finds the second best parent-slot j', such that j' is not owned by the owner of j^*, i.e., $j' \notin \mathcal{M}_{\mathcal{P}}(j^*)$. The second best net profit, $w_{ij'}$, equals:

  $$w_{ij'} = \max_{j\in\mathcal{P}-\mathcal{Q}-\mathcal{M}_{\mathcal{P}}(j^*)} v_{ij} \tag{5.20}$$

  Considering $\delta_{ij^*} = v_{ij^*} - w_{ij'}$ as the difference between the highest net profit and the second one, the child-slot i raises the price of a preferred parent-slot j^* by the bidding increment δ_{ij^*}, and sends its bid, b_{ij^*}, to j^*:

  $$b_{ij^*} = p_{j^*} + \delta_{ij^*} \tag{5.21}$$

- **Assignment phase:** The parent-slot j, which receives the highest bid from i^*, removes the connection to the child-slot i' (if there was any connection to i' in the beginning of the iteration), and assigns to i^*, i.e., the connection (i^*, j) is added to the current assignment $\mathcal{S}$. The parent-slot j also updates its own price to the received bid from the child-slot i^*, i.e., $p_j = b_{i^*j}$.

As we show in Section 2.3 the auction process will terminate, if $\delta_{ij} > 0$.

5.4.2 Distributed solution

Since the auction algorithm is centralized, it does not scale to many thousands of nodes. In our distributed model, unlike the centralized implementations, we do not rely on a central server with a global knowledge of all participants, and each node, as an auction participant, has only partial information about the system.

Private nodes in this system compete to become children of public nodes that are closer to them, and parents prefer close children nodes. Children proactively switch parents when they get lower connection delay by changing their parents. Each private node q calculates the connection delay to each public node p, $(qp).delay$, locally by measuring the round trip time of a message exchange in each shuffling. As in Equation 5.17, we have $(qp).profit = \frac{1}{(qp).delay}$.

If a public node has an unused parent-slot, its *price* is zero, otherwise it is equal the lowest profit it gains over its already connected children. For example, if public node p has three parent-slots and three children with delays 2ms, 3ms and 4ms, the price of p is $\frac{1}{4} = 0.25$. Private nodes constantly try to increase their connection profits over all their parent connections by competing for connections to the public nodes with a lower delay.

Private nodes place bids for parent-slots at the public nodes in their public views with the highest net profit, e.g., closest nodes to them. Public nodes increase their price by receiving new bids, and the more expensive a node is, the lower net profit it has. Thus, a parent node, which had a high net profit in one iteration, turns out to be a low profit node after receiving a number of bids. Hence, the seeking nodes will try to bid for other nodes with a higher net profit. This implies that in an overlay with $|\mathcal{C}|$ child-slots, if there is no churn in the system, eventually $|\mathcal{C}|$ distinct parent-slots receive at least one bid, and consequently the algorithm terminates by assigning all the child-slots to parent-slots. However, in a dynamic network, where nodes continuously join and leave the system, our algorithm keeps running and optimizes the connections in the overlay.

Since the price of a public node with unassigned parent-slot is zero, the first bid for a parent-slot will always win. It enables children to immediately connect to available parent-slots. However, when all of a parent's parent-slots are assigned, a bidding private node with lower delay than the existing connection will win the parent-slot and the parent will replace its child with the lower delay node. A child that has lost the parent-slot has to discover new nodes and bid for their parent-slots.

To establish the parent-child connections, a private node q periodically checks through `FindParent` (Algorithm 14), if it has a node p in its public view that (i) has a higher profit (lower delay) than its existing parents, and (ii) the profit of connecting to p is more than p's price. For example, assume the delay between node q and its worst parent is 5ms, and q finds two nodes a and b in its public view that have lower delays, for example 4ms, which means that $(qa).profit = (qb).profit = \frac{1}{4} = 0.25$. Assume the price of a is 0.2,

Algorithm 14 NAT traversal handlers

```
//Find better parents at private node $q$.
procedure FindParent $\langle\rangle$
    $z \leftarrow$ the lowest profit parent among $this.parents$
    if $this.childSlots$ has free entries then
        $(qz).profit \leftarrow 0$
    end if
    for all $p$ in $this.publicView$ do
        if $(qp).profit > (qz).profit$ and $(qp).profit > p.price$ then
            Send $ParentRequest(qp.profit)$ to $p$
        end if
    end for
end procedure

// Handling the parent request from private node $q$ at public node $p$.
upon event $\langle$ParentRequest $\mid (qp).profit\rangle$ from $q$
    if $this.parentSlots$ has free entries then
        assign a $parentSlot$ to $q$
        Send $ParentResponse(AssignAccepted)$ to $q$
    else
        if $(qp).profit > p.price$ then
            $z \leftarrow$ the lowest profit child among $this.children$
            assign a $parentSlot$ to $q$
            Send $Release$ to $z$
            Send $ParentResponse(AssignAccepted)$ to $q$
        else
            Send $ParentResponse(AssignRejected)$ to $q$
        end if
    end if
end event

// Handling the parent response from public node $p$ at private node $q$.
upon event $\langle$ParentResponse $\mid msg\rangle$ from $p$
    if $msg$ is AssignAccepted then
        if $q.childSlot$ has free entries then
            $q.parents.add(p)$                                ▷ add $p$ to the parent list
        else
            $z \leftarrow$ the lowest net profit parent             ▷ the worst parent
            if $((qp).profit - p.price) > ((qz).profit - z.price)$ then
                $q.parents.remove(z)$
                $q.parents.add(p)$
                Send $RemoveMeFromYourChildren$ to $z$
            else
                Send $RemoveMeFromYourChildren$ to $p$
            end if
        end if
    end if
end event
```

while the price of b is 0.5, thus, q sends its request only to a, since $(qa).profit > a.price$ and $(qb).profit < b.price$.

If q finds such a node, it sends a parenting request to it. When a public node p receives a parenting request from node q, accepts it if has free parent-slots, otherwise, if $(qp).profit$ is greater than the price of p, p abandons its child that has the lowest profit (highest delay), and accepts q as a new child. The disconnected node has to find a new parent. If p's price is greater than or equal to $(qp).profit$, p declines the request.

When a child node receives the acceptance message from a public node, it assigns one of its child-slots to a parent-slot of the parent. However, since a private node may send more connection requests than its number of child-slots, it might receive more acceptance messages than it needs. In this case, if the child has a free child-slot, it accepts the parent, otherwise, it checks all its assigned parents and finds the one with the lowest profit or the worst parent. If the profit of the connection to the worst parent is lower than the new parent, the child node releases the connection to the worst parent and accepts the new one, otherwise it ignores the received message.

5.5 Experiments

In this section, we evaluate the accuracy of our public-private estimation algorithm in simulation and compare the performance of the Croupier and Gozar with Nylon [64], the state-of-the-art performing NAT-friendly gossip-based PSS we found in the literature. We use also Cyclon [63] as a baseline for comparison, where Cyclon experiments are executed using only public nodes. Moreover, we show how our distributed auction algorithm improves the connection latency to private nodes and compare it to random assignment.

5.5.1 Experimental setup

We implemented Croupier, Gozar, Nylon, and Cyclon on the Kompics platform [76]. Kompics provides a framework for building P2P protocols and a discrete event simulator for simulating them using different bandwidth, latency and churn models. Our implementations of Cyclon and Nylon are based on the system descriptions in [63] and [64], respectively. For a cleaner comparison with Nylon, all protocols use the same tail and swapper policies for node selection and view merging, respectively.

In our experimental setup, for all four systems, the size of a node's partial view is 10 entries, and the size of subset of the partial view sent in each view exchange is 5. The gossiping round period for view exchange is set to one second. Latencies between nodes are modeled on Internet latencies, using a latency map based on the King data-set [68]. Unless stated otherwise, we use a public-private ratio of 0.2, similar to that seen in existing P2P systems [24, 100]. All experiments results are averaged over 5 runs. The evaluation metrics for new nodes that join the system are not included until they have executed 2 rounds, giving them enough time to initialize their estimates.

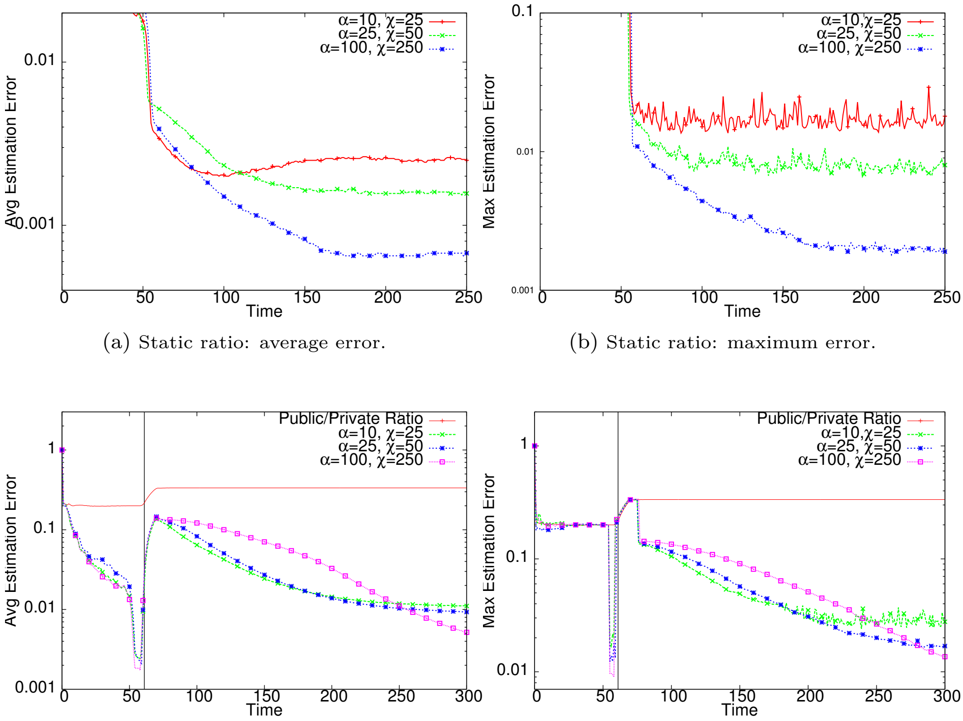
5.5.2 Estimation algorithm evaluation

We measure the accuracy of our ratio estimation protocol using two error metrics: the *maximum approximation error* and the *average approximation error*. Firstly, we define the upper bound on the approximation error of any nodes in the system using the Kolmogorov-Smirnov [96] metric. At each node i, the distance between the real ratio ω and the estimated ratio $E(\omega_i)$ is:

$$Err\,(i) = \|\omega \;- E\,(\omega_i)\,\| \tag{5.22}$$

We measure the maximum error as the maximum error over the set of all nodes, $\mathcal{N}$, in the system:

$$Err_{max} = \arg\max_{i \in \mathcal{N}} Err\,(i) \tag{5.23}$$



(a) Static ratio: average error.

(b) Static ratio: maximum error.

(c) Dynamic ratio: average error.

(d) Dynamic ratio: maximum error.

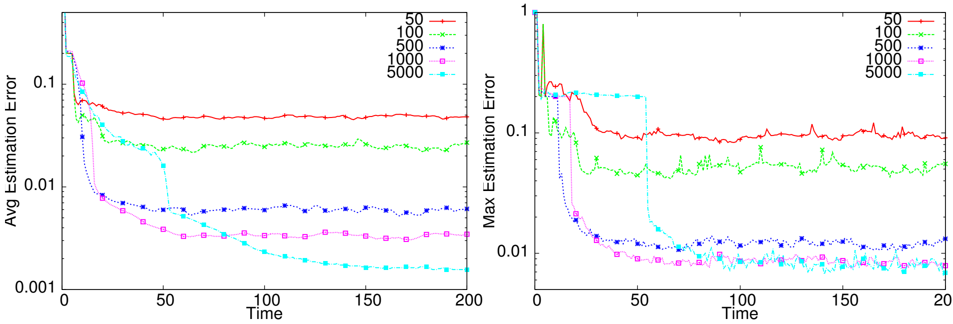
Figure 5.1: Convergence to a static/dynamic public/private ratio for different values of α and γ.

We also measure the average error over all the nodes, which is:

$$Err_{avg} = \frac{\sum\limits_{i \in \mathcal{N}} Err\,(i)}{|\mathcal{N}|} \tag{5.24}$$

History window. In this experiment, we evaluate the accuracy of our public-private ratio estimation using both a stable ratio and a dynamic ratio (where the ratio of public to private nodes changes over time). Both experiments have 1000 public nodes and 4000 private nodes join the system following a Poisson distribution with an inter-arrival time of 50 and 12.5 milliseconds, respectively. We measure the average error and maximum error while varying the size of the local history (α) and the neighbour history (γ).

Our experiments use three pairs of history window sizes: (i) *small*: $\alpha = 10$ and $\gamma = 25$, (ii) *medium*: $\alpha = 25$ and $\gamma = 50$, and (iii) *large*: $\alpha = 100$ and $\gamma = 250$. For the stable ratio, in Figures 5.1a and 5.1b, we can see clearly that larger values of α and γ have a slower convergence rate, but more accurate estimations. All 5000 nodes have joined the system by time $t = 51$, and it takes roughly 100 rounds longer for the largest history windows ($\alpha = 100$, $\gamma = 250$) to converge on good estimates compared to the smallest history



(a) Average errors.

(b) Maximum errors.

Figure 5.2: The effect of system size on the estimation algorithm for a stable ratio of 0.2 (α=25, and γ=50).

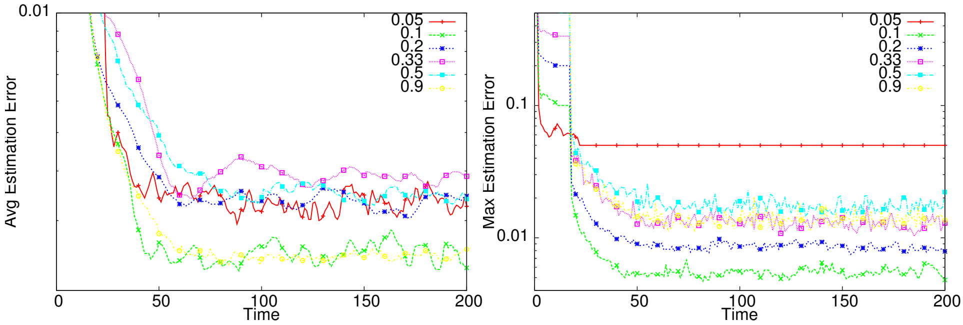
windows ($\alpha = 10$, $\gamma = 25$). The largest history window run converges to an average error of 0.07% with a maximum error of 0.2%, while the smallest window converges to an average error of 0.25% with a maximum error of 1.8%.

In Figures 5.1c and 5.1d, we observe the convergence rate and estimation accuracy for a public-private ratio that grows slowly in size. We use the same scenario of joining 1000 public nodes and 4000 private nodes over the first 51 rounds, then waited 7 rounds, and then added a new public node every 42ms. The actual ratio is 0.3 until time $t = 58$, then the ratio rises at a constant rate to $t = 72$ to reach 0.33, whereupon the ratio remains at 0.33 until the end of the experiment run.

We can see here that for a dynamic public-private ratio the largest history windows take a lot longer to converge on the new ratio, while the smallest history windows converge quicker, but eventually with less accurate estimations when the ratio stabilizes again. From $t = 58$ to $t = 180$, the smallest window has the lowest average error, while from $t = 180$ to $t = 260$ the medium-sized window has the lowest average error, then after $t = 260$, the largest window converges closer to the real ratio. For a ratio that changes frequently and by a large amount, we would need window sizes closer to our smaller window sizes, but for more stable ratios medium or large-sized windows would have lower average error and lower maximum errors. Unless stated otherwise, further experiments use the medium history window sizes, $\alpha = 25$ and $\gamma = 50$, as, for a real system, it would provide a reasonable balance of good estimations and adaptability to a dynamic ratio.

Network size. In this experiment, we vary the number of nodes in the system to see its effect on the estimation accuracy. We measure systems with 50, 100, 500, 1000, and 5000 nodes. In these experiments, public and private nodes public nodes join the system following a Poisson distribution with an inter-arrival time of 50 and 12.5 milliseconds, respectively.

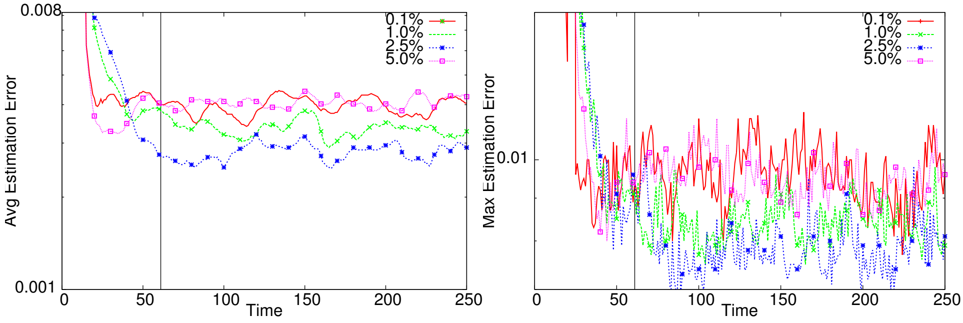
In Figure 5.2, we can see that there is an increase in estimation accuracy with increasing system size. For systems with 5000 nodes, average estimation error is only 0.2%, while for systems with only 100 nodes it rises to 2.5%, rising again to 5% for systems with only 50 nodes. Similarly, the maximum estimation error rises from 0.7% for 5000 nodes to 5.5% for 100 nodes, and to 9% for 50 nodes. In general, we can say that estimation accu-



(a) Average errors.

(b) Maximum errors.

Figure 5.3: Estimation accuracy for different ratios of public to private nodes.



(a) Average error.

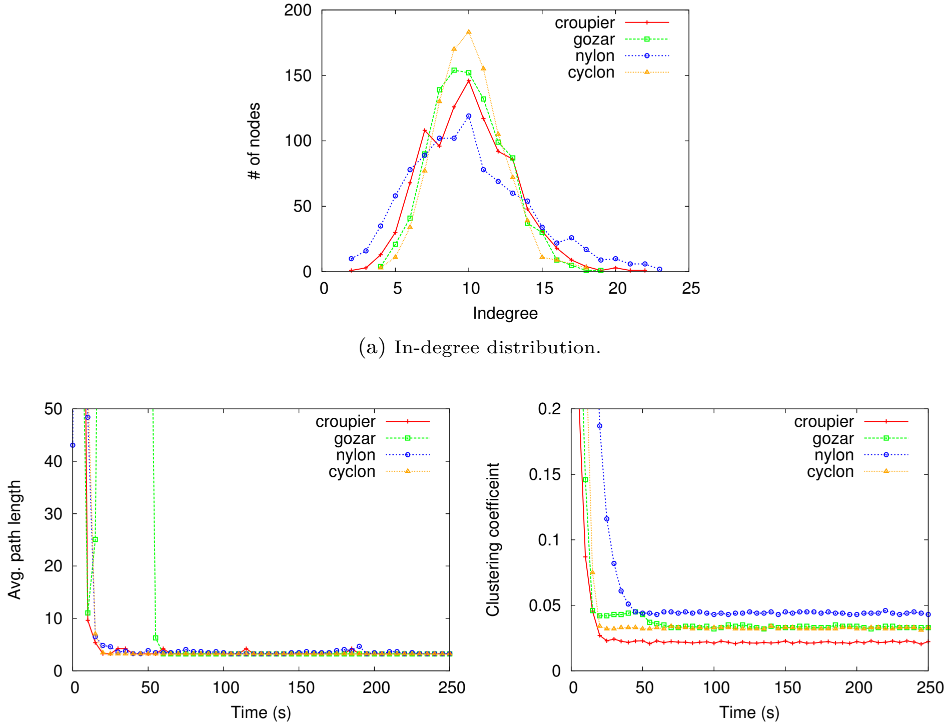
(b) Maximum error.

Figure 5.4: Effect of churn on the estimation algorithm for a stable ratio (α=25, and γ=50). Churn started at time t = 61.

racy improves rapidly up to systems with several hundred nodes, and then only becomes gradually better thereafter. For example, the change in estimation accuracy from 1000 to 5000 nodes is negligible - an improvement in average estimation error of only 0.15% and no difference in maximum estimation error.

Different ratios. Different P2P systems will have different ratios of public to private nodes, so here we investigate the accuracy of estimations for different stable ratios of public to private nodes, with experiments of 1000 nodes. We measure the average and maximum estimation errors for ratios of 5%, 10%, 20%, 33%, 50%, 80%. We concentrate our measurements more on systems with smaller relative numbers of public nodes, as this is commonly the case in real-world systems.

As we can see in Figure 5.3, there is no significant difference in the average estimation error for all ratios. We do notice, however, for only 5% public nodes that the maximum error becomes significantly higher (5%) and constant. This is the result of an outlier private node that happens not to receive enough different estimates from public to improve its



(a) In-degree distribution.

(b) Avg. path length.

(c) Clustering co-efficient.

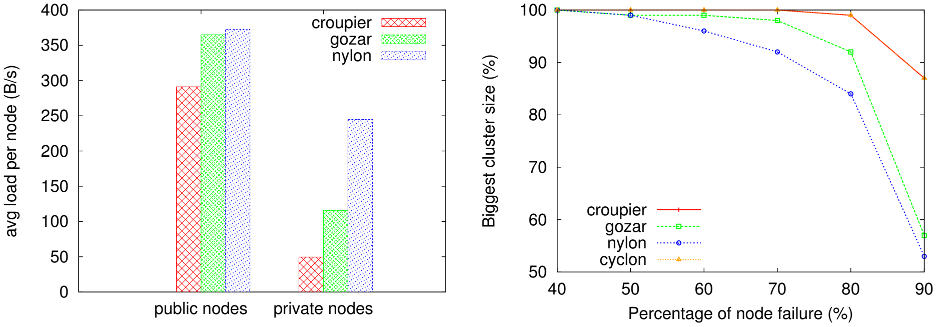
Figure 5.5: Randomness properties.

local estimation. So, for systems with fewer than 5% public nodes, we can expect that a few private nodes may have poor ratio estimates.

Churn. Node membership in large-scale distributed systems is typically subject to continuous change, in a process called churn. We model churn by replacing a fixed fraction of randomly selected public and private nodes with new nodes at each gossiping round, but keeping the ratio of public to private nodes stable. The churn rate is set to a level common for P2P systems [95]: assuming a gossip round-time of one second and a mean session duration of 15 minutes, approximately 0.1% of nodes leave the system per second and rejoin immediately as newly initialized nodes. Figure 5.4 shows the average error and maximum error, respectively, for ratio estimation under churn. As can be seen, there is no significant effect of churn of up to 5% on the estimation algorithm. This rate of churn is 50 times higher than rates measured in [95].

5.5.3 Peer sampling evaluation

In this subsection, we evaluate the performance of the PSS, which builds on the estimation protocol for its correct functioning.



(a) Protocol overhead relative to Cyclon.

(b) Connectivity after catastrophic failure.

Figure 5.6: Protocol overhead.

PSS randomness. Here, we compare the randomness of the PSS' of Croupier with Gozar and Nylon. Cyclon is used as a baseline for true randomness. In the first experiment we measure the in-degree distribution over the nodes in the all four systems. Figure 5.5a shows the in-degree distribution of nodes after 250 rounds (the out-degree of all nodes is 10). In a uniformly random system, we expect that the in-degree is distributed uniformly among all nodes. Cyclon shows this behaviour as the node in-degree is almost distributed uniformly among nodes [63]. We can see the same distribution in Croupier, as well as, in Gozar and Nylon - their in-degree distributions are very close to Cyclon.

In Figure 5.5b, we compare the average path length of the three systems, with Cyclon as a baseline. The path length for two nodes is measured as the minimum number of hops between two nodes, and the average path length is the average of all path lengths between all nodes in the system. Figure 5.5b also shows the average path length for the system in different rounds. Here, we can see the average path length of Croupier, Gozar and Nylon track Cyclon very closely. As we can see, in the first few rounds, the path length of Gozar is high, as this is the time that nodes need to find their partners used for relaying.

Finally, we compare the clustering coefficient of the systems. A node's clustering coefficient shows at what level the neighbours of a node are also neighbours of each other. For a complete graph, it is 1, and for a tree, where there is the no connection between any two neighbours of a node, it is 0. We calculate the average clustering coefficient as the average across all nodes in the system. Figure 5.5c shows the evolution of the clustering coefficient of the constructed overlay by each system. We can see that Croupier has smaller clustering coefficient than Gozar, Nylon and Cyclon. Our understanding of why Croupier has a smaller clustering coefficient is as follows. Since a private node in Croupier exchanges its view only with a public node, two private nodes never have a chance to exchange their neighbour list directly. Therefore, the probability that two private node establish a connection with each other's neighbours decreases. Since in our experiments 80% of nodes are private nodes, the average clustering coefficient in the overlay also decreases.

Protocol overhead. An important objective for any PSS is to minimize communication costs and to bound the extra overhead on public nodes (and achieve fairness). The network traffic exchanged by a node in Croupier is proportional to the rate of gossiping, as message sizes are bounded. Every node, both public and private, send one message per round. Private nodes receive one message per round (the response to the message they sent). On average, every public node receives one message from a public node per round, one response to a message they sent per round, and n messages from private nodes per round (where n is the ratio of private nodes to public nodes).

In this experiment, we set the local history α to 25, and the neighbour history length γ to 100. As in the other experiments, we bounded the number of estimations piggybacked on shuffle requests to 10. Each estimation required 5 bytes: two bytes for the node identifier, one byte each for the public and private counts, and one for the timestamps. The steady-state overhead is shown in Figure 5.6a. As we can see in Figure 5.6a, the public node overhead in Croupier is less than that of Gozar and Nylon. Interestingly, the overhead of private nodes, which are 80% of the nodes, is less than half compared to Gozar, and less than one fourth compared to Nylon. As such, we conclude that the overhead on public nodes is not excessive, and our goal of fairness to public nodes has been achieved.

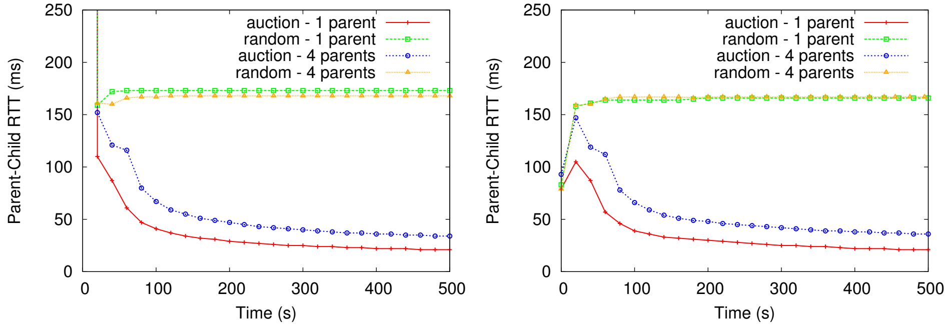
Connectivity. We finally evaluate the behaviour of Croupier and Gozar if high numbers of nodes leave the system or crash at a single instant in time. We measure the size of biggest cluster after a catastrophic failure. Figure 5.6b shows the size of biggest cluster for Croupier, Gozar and Nylon for varying percentages of private nodes, when varying numbers of nodes fail. We can see that Croupier is more resilient to node failure than both Gozar and Nylon. For example, in the case of 80% private nodes, when 90% of the nodes fail, the biggest cluster still covers more than 85% of the nodes, while it covers 57% and 53% of nodes in Gozar and Nylon, respectively.

5.5.4 NAT traversal evaluation

Here, we compare the performance of our auction algorithm in establishing parental connection between public and private nodes with random assignment. There are 5000 nodes in the system, such that 20% of nodes are public. We have done this experiment in two scenarios: (i) join only scenario, where the nodes join the system with an inter-arrival time of 10 milliseconds, and (ii) the churn scenario, where 2000 public nodes join and leave with an inter-arrival time of 10 milliseconds from time $t = 150$, and afterwards no more changes happen in the system. Each public node can accept up to 25 private nodes as children, and for private nodes we consider two cases, where they connect to one parent, and four parents.

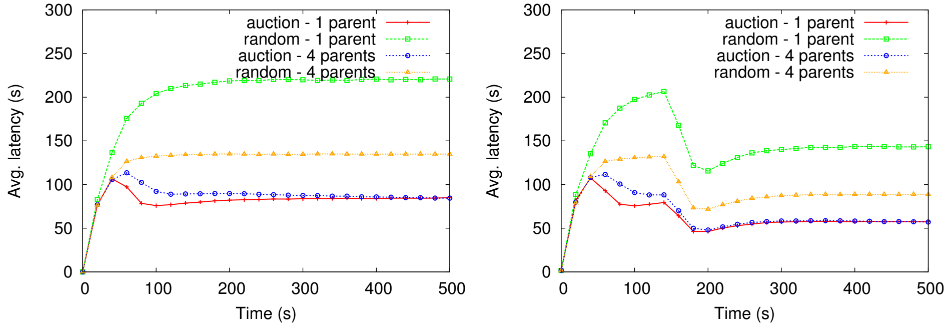
In this experiment, we first evaluate the parent-child round trip time (RTT) over time. Figure 5.7 shows that how RTT between public and private nodes decreases, when we use the auction algorithm in the system, while when the private nodes choose their parents at random, no improvement happens in RTT. As we see, when private nodes have only one parent, their RTT is less than when they have four parents. What we show as RTT for private nodes with four parents is the average of RTT to all four parents. That is why it is a bit higher compare to the case when private nodes have only one parent.

We see the total connection latency to private nodes in Figure 5.8. The connection latency is the time to make a connection to all the private nodes in the system. To measure it, each node finds the average time to establish connection to all the nodes in its private view, and then we sum up all these averages. We do not count unreachable node in this value. We see that after finding a stable parental relation ($t = 125$ in join, and $t = 200$



(a) Join. (b) Churn.

Figure 5.7: The private-parent RTT latency in the auction algorithm and random assignment.



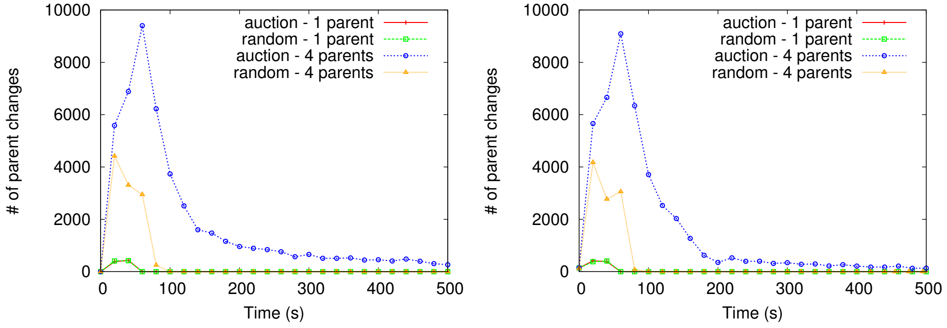
(a) Join. (b) Churn.

Figure 5.8: The connection latency to private nodes in the auction algorithm and random assignment.

in churn), the latency becomes constant. In the churn scenario we see that the latency decreases during the churn period. That is why in this period the number of unreachable private nodes increases (Figure 5.10b), and therefore smaller number of nodes are used in our measurement. Again, here we see that the latency in the auction model is much less than the random assignment.

We see in Figure 5.9a, there is a high number of parent switching happens in the beginning, but while private nodes find appropriate parents, the system converges to a more stable situation and the number of changes drops to zero. In the churn scenario (Figure 5.9b) we see almost the same behaviour, because in this scenario, after a while public nodes stop joining/leaving and therefore private nodes can find their parents properly. As we expect, the number of parent switching in random assignment is much less than the auction model, since private nodes do not change their parents. Since we count all the parent assignments (not only switching), the value seen in Figures 5.9a and 5.9b are greater than zero in the random assignment.

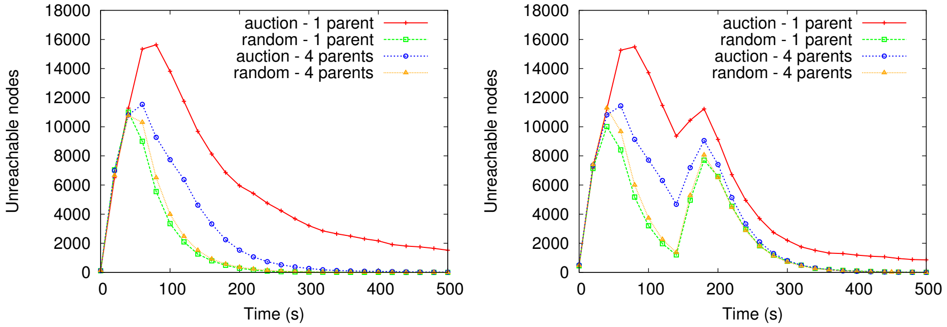
Finally, Figure 5.10 shows the number of unreachable private nodes. Unreachable



(a) Join.

(b) Churn.

Figure 5.9: The number of parent switching in the auction algorithm and random assignment.



(a) Join.

(b) Churn.

Figure 5.10: The total number of unreachable private nodes in the auction algorithm and random assignment.

private nodes are those private nodes that other nodes cannot establish connection to them using the parent address in their node descriptors. The reason behind it is that a node p may have an old node descriptor of a private node q and q'parents in its private view, while q has updated its parents. Hence, p fails to make a connection using q's old descriptor. In both join and churn scenarios, we see the number of unsuccessful connections decreases over time. However, Figure 5.10b shows that during the churn period, the number of unreachable nodes increases, because the known parents for a number of private nodes may have failed. Here, we see that the number of unreachable private nodes in the random assignment is less than the auction algorithm. That is because in the auction model, private nodes change their parents more often, and it increases the probability that nodes use their old parents to make connection to them.

Chapter 6

Conclusions

In this thesis, we focused on four problems in P2P live streaming systems: (i) designing and implementing a P2P live streaming system that guarantees the quality of service (QoS) in the presence of dynamism in the network, (ii) resolving the free-riding problem and incentivizing nodes to participate in media distribution, (iii) guaranteeing QoS at end users in case of bottlenecks in the available resources in the overlay, and (iv) overcoming the NAT problem in the Internet. We answered to these questions by presenting a number of algorithms and systems:

Sepidar and Glive. Within our streaming systems, we have proposed a distributed market model to construct a content distribution overlay, such that (i) nodes with increasing upload bandwidth are located closer to the media source, and (ii) nodes with similar upload bandwidth become neighbours. We use this model to build a multiple-tree overlay in Sepidar, as well as a mesh overlay in Glive. In the former solutions the data blocks are pushed through the trees, while in the latter, nodes pull data from their neighbours in the mesh.

We assume each node can have a number upload connections and a number of download connections. To be able to distribute data blocks to all the nodes, the download connections of nodes should be assigned to other nodes' upload connections. We model this problem as an assignment problem. There exist centralized solutions for this problem, e.g., the auction algorithm, which are not feasible in large and dynamic networks with real-time constraints. An alternative decentralized implementation of the auction algorithm is based on sampling from a random overlay, but it has a slow convergence time. Therefore, we address the problem by using the gossip-generated Gradient overlay to provide nodes with a partial view of other nodes that have a similar upload bandwidth or slightly higher.

We evaluate Sepidar and Glive in simulation, and compare their performance with the state-of-the-art NewCoolstreaming. We show that our solutions provide better playback continuity and lower playback latency than that of NewCoolstreaming in different scenarios. In addition, we compare Sepidar with Glive to highlight the differences of the multiple-tree and the mesh overlays. We observe that the mesh-based overlay outperforms the multiple-tree overlay in all the scenarios. Moreover, we compare the convergence time of our systems, Sepidar and Glive, when the node samples are given by the Gradient overlay rather than a random network. The experiment results show that the overlays converge faster when our market model works on top of the Gradient overlay. Finally, we evaluate SEPIDAR and GLIVE performance in different free-rider settings, and examine the effectiveness of our mechanism for addressing the free-riding problem.

Clive. The main contribution of CLIVE is a P2P live streaming system that integrates cloud resources (helpers) whenever the nodes resources are not sufficient to guarantee a predefined QoS. Two types of helpers are used in CLIVE, (i) active helper (AH), which is an autonomous virtual machine, e.g., Amazon EC2, that participates in the streaming protocol, and (ii) passive helper (PH), which is a storage service, e.g., Amazon S3, that provides content on demand. CLIVE estimates the available capacity in the system through a gossip-based aggregation protocol and provisions the required resources from the cloud to guarantee a given level of QoS at low cost.

We implemented a prototype CLIVE system based on Amazon's services like EC2, and S3. In such approach, rented cloud resources (helpers) are added on demand to the overlay, to increase the amount of total available bandwidth and the probability of receiving the video on time. Hence, the problem to be solved becomes minimizing the economical cost, provided that a set of constraints on QoS is satisfied. To demonstrate the feasibility of CLIVE, we performed extensive simulations and evaluate our system using large scale experiments under dynamic realistic settings. We show that we can save up to 45% of the cost by choosing the right number of AHs compared to only using a PH to guarantee the predefined QoS.

Croupier and Gozar. The main contribution of CROUPIER and GOZAR is a gossip-based peer sampling service (PSS) provides each node with a small list of live nodes in a system. In the Internet, however, most of existing gossiping protocols break down, as nodes cannot establish direct connections to nodes behind NATs. Moreover, existing NAT traversal algorithms for establishing connectivity to private nodes rely on third party servers running at well-known, public IP addresses.

GOZAR is a NAT-friendly gossip-based peer sampling service that also provides a distributed NAT traversal service to clients of the PSS. Public nodes are leveraged to provide both the relaying and hole punching services. Relaying is only used for gossiping to private nodes, and is preferred to hole punching or routing through existing open connections (as done in Nylon), as relaying has lower connection latency, enabling a faster gossiping cycle, and the messages relayed are small, thus, adding only low overhead to public nodes. Relaying and hole punching services provided by public nodes are enabled by every private node partnering with a small number of (redundant) public nodes and keeping a connection open to them. We extended node descriptors for private nodes to include the addresses of their partners, so when a node wishes to send a message to a private node (through relaying) or establish a direct connection with the private node through hole punching, it sends a relay or connection message to one (or more) of the private node's partners.

Our second contribution in this problem area is CROUPIER, a NAT-friendly gossip-based peer sampling service that is built without relaying. Public nodes act as Croupiers, shuffling views amongst one another as well as on behalf on private nodes. Our main insight was to partition a node's view into two parts: a public view and a private view. This decision, however, necessitated that we could identify a node as being either public or private, and that nodes have a local estimation of the ratio of public to private nodes in the system. To solve these problems, we presented a minimal, distributed algorithm for the identification of a node's NAT type, as well a protocol to estimate the public/private ratio that piggybacks on existing CROUPIER shuffle messages.

Finally, we presented a NAT traversal middleware on top of CROUPIER PSS that enable nodes to establish connections to private nodes. We also explained how our distributed auction algorithm can reduce the connection latency to private nodes in our NAT traversal middleware. We showed in simulation that GOZAR and CROUPIER preserve the randomness properties of a gossip-based peer sampling service. We showed the robustness of both systems when a large fraction of nodes reside behind NATs and also in catastrophic failure scenarios. Moreover, we showed that CROUPIER's overhead is less than GOZAR and both are an improvement on existing NAT-aware PSS'. We also showed that the extra overhead incurred by public nodes in CROUPIER is acceptable for the most applications.

Future work

In the current implementation of our streaming overlays, we consider upload bandwidth of nodes as the only influencing parameter in the overlay construction. We believe this model can be extended to include other important node characteristics, such as node uptime, load, reputation, and locality. Furthermore, in our streaming systems, we did not address the problem of nodes colluding to receive the video stream for free. It would be interesting to solve the free-rider problem, where a group of nodes collude to receive data without helping in distributing it.

Another future research direction is to integrate our existing streaming applications with GOZAR and CROUPIER and deploy them in the open Internet. Moreover, we are currently implementing a prototype of CLIVE system based on Amazon's services like EC2, and S3. It would also be interesting to have a real implementation of CLIVE in the Internet, and evaluate how the AH/PH combination affects the total cost.

Bibliography

[1] A. Oreskovic, “Youtube video views hit 4 billion per day.” http://www.reuters.com/article/2012/01/23/us-google-youtube-idUSTRE80M0TS20120123, [Online; accessed 20-Nov-2012].

[2] A. Payberah, J. Dowling, F. Rahimian, and H. Haridi, “Sepidar: Incentivized market-based p2p live-streaming on the gradient overlay network,” in *Proc. of ISM’10*, pp. 1–8, IEEE, 2010.

[3] A. Payberah, J. Dowling, and S. Haridi, “Glive: The gradient overlay as a market maker for mesh-based p2p live streaming,” in *Proc. of ISPDC’11*, pp. 153–162, IEEE, 2011.

[4] D. Bertsekas, “The auction algorithm: a distributed relaxation method for the assignment problem,” *Annals of Operations Research*, vol. 14, no. 1, pp. 105–123, 1988.

[5] D. Bertsekas, *Network optimization: continuous and discrete models.* Athena Scientific Belmont, 1998.

[6] A. Payberah, J. Dowling, F. Rahimian, and S. Haridi, “Distributed optimization of p2p live streaming overlays,” *Springer Computing, Special Issue on Extreme Distributed Systems: From Large Scale to Complexity*, vol. 94, no. 8, pp. 621–647, 2012.

[7] J. Sacha, B. Biskupski, D. Dahlem, R. Cunningham, R. Meier, J. Dowling, and M. Haahr, “Decentralizing a service-oriented architecture,” *Peer-to-Peer Networking and Applications (PPNA)*, vol. 3, no. 4, pp. 323–350, 2010.

[8] J. Sacha, J. Dowling, R. Cunningham, and R. Meier, “Discovery of stable peers in a self-organizing peer-to-peer gradient topology.,” in *Proc. of DAIS’06*, pp. 70–83, Springer, 2006.

[9] A. Payberah, J. Dowling, and S. Haridi, “Gozar: Nat-friendly peer sampling with one-hop distributed nat traversal,” in *Proc. of DAIS’11*, pp. 1–14, Springer, 2011.

[10] J. Dowling and A. Payberah, “Shuffling with a croupier: Nat-aware peer-sampling,” in *Proc. of ICDCS’12*, pp. 102–111, IEEE, 2012.

[11] M. Jelasity, S. Voulgaris, R. Guerraoui, A. Kermarrec, and M. van Steen, “Gossip-based peer sampling,” *ACM Transaction on Computer System (TOCS)*, vol. 25, no. 3, 2007.

[12] W. Yiu, X. Jin, and S. Chan, "Challenges and approaches in large-scale p2p media streaming," *MultiMedia*, vol. 14, no. 2, pp. 50–59, 2007.

[13] V. Padmanabhan, H. Wang, P. Chou, and K. Sripanidkulchai, "Distributing streaming media content using cooperative networking," in *Proc. of NOSSDAV'02*, pp. 177–186, ACM, 2002.

[14] Y. Guo, K. Suh, J. Kurose, and D. Towsley, "Directstream: A directory-based peer-to-peer video streaming service," *Computer Communications*, vol. 31, no. 3, pp. 520–536, 2008.

[15] S. Banerjee, B. Bhattacharjee, and C. Kommareddy, "Scalable application layer multicast," in *Proc. of SIGCOMM'02*, pp. 205–217, ACM, 2002.

[16] D. Tran, K. Hua, and T. Do, "Zigzag: An efficient peer-to-peer scheme for media streaming," in *Proc. of INFOCOM'03*, pp. 1283–1292, IEEE, 2003.

[17] A. Gong, G. Ding, Q. Dai, and C. Lin, "Bulktree: An overlay network architecture for live media streaming," *Journal of Zhejiang University-Science A*, vol. 7, pp. 125–130, 2006.

[18] M. Ripeanu, "Peer-to-peer architecture case study: Gnutella network," in *Proc. of P2P'01*, pp. 99–100, IEEE, 2001.

[19] X. Jiang, Y. Dong, D. Xu, and B. Bhargava, "Gnustream: a p2p media streaming system prototype," in *Proc. of ICME'03*, pp. II–325, IEEE, 2003.

[20] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 4, pp. 149–160, 2001.

[21] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *Proc. of Middleware'01*, pp. 329–350, Springer, 2001.

[22] M. Castro, P. Druschel, A. Kermarrec, A. Nandi, A. Rowstron, and A. Singh, "Splitstream: high-bandwidth multicast in cooperative environments," in *ACM SIGOPS Operating Systems Review*, pp. 298–313, ACM, 2003.

[23] T. Locher, R. Meier, S. Schmid, and R. Wattenhofer, "Push-to-pull peer-to-peer live streaming," *Distributed Computing*, pp. 388–402, 2007.

[24] B. Li, S. Xie, Y. Qu, G. Keung, C. Lin, J. Liu, and X. Zhang, "Inside the new coolstreaming: Principles, measurements and performance implications," in *Proc. of INFOCOM'08*, pp. 1031–1039, IEEE, 2008.

[25] X. Zhang, J. Liu, B. Li, and Y. Yum, "Coolstreaming/donet: a data-driven overlay network for peer-to-peer live media streaming," in *Proc. of INFOCOM'05*, pp. 2102–2111, IEEE, 2005.

[26] F. Pianese, D. Perino, J. Keller, and E. Biersack, "Pulse: an adaptive, incentive-based, unstruct@inproceedingsured p2p live streaming system," *IEEE Transaction on Multimedia*, vol. 9, no. 8, pp. 1645–1660, 2007.

[27] A. Payberah, J. Dowling, F. Rahimian, and S. Haridi, "gradientv: Market-based p2p live media streaming on the gradient overlay," in *Proc. of DAIS'10*, pp. 212–225, Springer, 2010.

[28] R. Fortuna, E. Leonardi, M. Mellia, M. Meo, and S. Traverso, "Qoe in pull based p2p-tv systems: Overlay topology design tradeoffs," in *Proc. of P2P'10*, pp. 1–10, IEEE, 2010.

[29] K. Park, S. Pack, and T. Kwon, "Climber: An incentive-based resilient peer-to-peer system for live streaming services," in *Proc. of IPTPS'08*, 2008.

[30] S. Jarvis, G. Tan, D. Spooner, and G. Nudd, "Constructing reliable and efficient overlays for p2p live media streaming," *International Journal of Simulation and Process Modelling (IJSPM)*, vol. 7, no. 2, pp. 54–62, 2006.

[31] J. Mol, D. Epema, and H. Sips, "The orchard algorithm: Building multicast trees for p2p video multicasting without free-riding," *IEEE Transaction on Multimedia*, vol. 9, no. 8, pp. 1593–1604, 2007.

[32] V. Venkataraman, K. Yoshida, and P. Francis, "Chunkyspread: Heterogeneous unstructured tree-based peer-to-peer multicast," in *Proc. of ICNP'06*, pp. 2–11, IEEE, 2006.

[33] N. Magharei, R. Rejaie, and Y. Guo, "Mesh or multiple-tree: A comparative study of live p2p streaming approaches," in *Proc. of INFOCOM'07*, pp. 1424–1432, IEEE, 2007.

[34] D. Frey, R. Guerraoui, A. Kermarrec, and M. Monod, "Boosting gossip for live streaming," in *Proc. of P2P'10*, pp. 1–10, IEEE, 2010.

[35] V. Pai, K. Kumar, K. Tamilmani, V. Sambamurthy, and A. Mohr, "Chainsaw: Eliminating trees from overlay multicast," *Peer-to-peer systems IV*, pp. 127–140, 2005.

[36] S. Asaduzzaman, Y. Qiao, and G. Bochmann, "Cliquestream: an efficient and fault-resilient live streaming network on a clustered peer-to-peer overlay," in *Proc. of P2P'08*, pp. 269–278, IEEE, 2008.

[37] F. Wang, Y. Xiong, and J. Liu, "mtreebone: A collaborative tree-mesh overlay network for multicast video streaming," *IEEE Transaction on Parallel and Distributed Systems (TPDS)*, vol. 21, no. 3, pp. 379–392, 2010.

[38] N. Magharei and R. Rejaie, "Prime: Peer-to-peer receiver-driven mesh-based streaming," *IEEE/ACM Transaction on Networking*, vol. 17, no. 4, pp. 1052–1065, 2009.

[39] B. Chang, Y. Shi, and N. Zhang, "Hymonet: a peer-to-peer hybrid multicast overlay network for efficient live media streaming," in *Proc. of AINA'06*, vol. 1, pp. 6–pp, IEEE, 2006.

[40] M. Hefeeda, A. Habib, D. Xu, B. Bhargava, and B. Botev, "Collectcast: A peer-to-peer service for media streaming," *Multimedia Systems*, vol. 11, no. 1, pp. 68–81, 2005.

[41] M. Hefeeda, A. Habib, B. Botev, D. Xu, and B. Bhargava, "Promise: peer-to-peer media streaming using collectcast," in *Proc. of ICMR'03*, pp. 45–54, ACM, 2003.

[42] D. Kostić, A. Rodriguez, J. Albrecht, and A. Vahdat, "Bullet: High bandwidth data dissemination using an overlay mesh," in *Proc. of SOSP'03*, pp. 282–297, ACM, 2003.

[43] M. Zhang, Y. Tang, L. Zhao, J. Luo, and S. Yang, "Gridmedia: A multi-sender based peer-to-peer multicast system for video streaming," in *Proc. of ICME'05*, pp. 614–617, IEEE, 2005.

[44] A. Vlavianos, M. Iliofotou, and M. Faloutsos, "Bitos: Enhancing bittorrent for supporting streaming applications," in *Proc. of INFOCOM'06*, pp. 1–6, IEEE, 2006.

[45] J. Liang and K. Nahrstedt, "Dagstream: Locality aware and failure resilient peer-to-peer streaming," in *Proc. of Electronic Imaging*, pp. 60710L–60710L, International Society for Optics and Photonics, 2006.

[46] X. Su and S. Dhaliwal, "Incentive mechanisms in p2p media streaming systems," *Internet Computing*, vol. 14, no. 5, pp. 74–81, 2010.

[47] M. Karakaya, I. Korpeoglu, and O. Ulusoy, "Free riding in peer-to-peer networks," *Internet Computing*, vol. 13, no. 2, pp. 92–98, 2009.

[48] G. Tan and S. Jarvis, "A payment-based incentive and service differentiation scheme for peer-to-peer streaming broadcast," *IEEE Transaction on Parallel and Distributed Systems (TPDS)*, vol. 19, no. 7, pp. 940–953, 2008.

[49] C. Buragohain, D. Agrawal, and S. Suri, "A game theoretic framework for incentives in p2p systems," in *Proc. of P2P'03*, pp. 48–56, IEEE, 2003.

[50] R. Ma, S. Lee, J. Lui, and D. Yau, "Incentive and service differentiation in p2p networks: a game theoretic approach," *IEEE/ACM Transaction on Networking (TON)*, vol. 14, no. 5, pp. 978–991, 2006.

[51] B. Cohen, "Incentives build robustness in bittorrent," in *Proc. of Economics of Peer-to-Peer systems*, vol. 6, pp. 68–72, 2003.

[52] M. Meulpolder, J. Pouwelse, D. Epema, and H. Sips, "Bartercast: A practical approach to prevent lazy freeriding in p2p networks," in *Proc. of IPDPS'09*, pp. 1–8, IEEE, 2009.

[53] S. Kamvar, M. Schlosser, and H. Garcia-Molina, "The eigentrust algorithm for reputation management in p2p networks," in *Proc. of WWW'03*, pp. 640–651, ACM, 2003.

[54] J. Mol, J. Pouwelse, M. Meulpolder, D. Epema, and H. Sips, "Give-to-get: free-riding resilient video-on-demand in p2p systems," in *Electronic Imaging*, pp. 681804–681804, International Society for Optics and Photonics, 2008.

[55] H. Li, A. Clement, E. Wong, J. Napper, I. Roy, L. Alvisi, and M. Dahlin, "Bar gossip," in *Proc. of OSDI'06*, pp. 191–204, USENIX Association, 2006.

[56] P. Eugster, R. Guerraoui, S. Handurukande, P. Kouznetsov, and A. Kermarrec, "Lightweight probabilistic broadcast," *ACM Transaction on Computer Systems (TOCS)*, vol. 21, no. 4, pp. 341–374, 2003.

[57] M. Jelasity, A. Montresor, and O. Babaoglu, "Gossip-based aggregation in large dynamic networks," *ACM Transaction on Computer Systems (TOCS)*, vol. 23, no. 3, pp. 219–252, 2005.

[58] M. Jelasity, A. Montresor, and O. Babaoglu, "T-man: Gossip-based fast overlay topology construction," *Computer Networks*, vol. 53, no. 13, pp. 2321–2339, 2009.

[59] R. Baldoni, M. Platania, L. Querzoni, and S. Scipioni, "Practical uniform peer sampling under churn," in *Proc. of ISPDC'10*, pp. 93–100, IEEE, 2010.

[60] E. Bortnikov, M. Gurevich, I. Keidar, G. Kliot, and A. Shraer, "Brahms: Byzantine resilient random membership sampling," *Computer Networks*, vol. 53, no. 13, pp. 2340–2359, 2009.

[61] A. Ganesh, A. Kermarrec, and L. Massoulié, "Peer-to-peer membership management for gossip-based protocols," *IEEE Transactions on Computers (TC)*, vol. 52, no. 2, pp. 139–149, 2003.

[62] M. Jelasity and A. Montresor, "Epidemic-style proactive aggregation in large overlay networks," in *Proc. of ICDCS'04*, pp. 102–109, IEEE, 2004.

[63] S. Voulgaris, D. Gavidia, and M. Van Steen, "Cyclon: Inexpensive membership management for unstructured p2p overlays," *Journal of Network and Systems Management (JNSM)*, vol. 13, no. 2, pp. 197–217, 2005.

[64] A. Kermarrec, A. Pace, V. Quema, and V. Schiavoni, "Nat-resilient gossip peer sampling," in *Proc. of ICDCS'09*, pp. 360–367, IEEE, 2009.

[65] J. Rosenberg, R. Mahy, P. Matthews, and D. Wing, "Session traversal utilities for nat (stun)," 2008.

[66] R. Mahy, P. Matthews, and J. Rosenberg, "Traversal using relays around nat (turn): Relay extensions to session traversal utilities for nat (stun)," 2010.

[67] N. Drost, E. Ogston, R. van Nieuwpoort, and H. Bal, "Arrg: real-world gossiping," in *Proc. of HPDC'07*, pp. 147–158, ACM, 2007.

[68] K. Gummadi, S. Saroiu, and S. Gribble, "King: Estimating latency between arbitrary internet end hosts," in *Proc. of IMC'02*, pp. 5–18, ACM, 2002.

[69] J. Leitão, R. van Renesse, and L. Rodrigues, "Balancing gossip exchanges in networks with firewalls," in *Proc. of IPTPS'10*, pp. 7–7, USENIX, 2010.

[70] C. Vasconcelos and B. Rosenhahn, "Bipartite graph matching computation on gpu," in *Proc. of EMMCVPR'09*, pp. 42–55, Springer, 2009.

[71] A. Datta, I. Stoica, and M. Franklin, "Lagover: latency gradated overlays," in *Proc. of ICDCS'07*, pp. 13–13, IEEE, 2007.

[72] H. W. Kuhn, "The hungarian method for the assignment problem," *Naval Research Logistic Quarterly,*, vol. 2, no. 1-2, pp. 83–97, 1955.

[73] B. Biskupski, M. Schiely, P. Felber, and R. Meier, "Tree-based analysis of mesh overlays for peer-to-peer streaming," in *Proc. of DAIS'08*, pp. 126–139, Springer, 2008.

[74] N. Carlsson and D. Eager, “Peer-assisted on-demand streaming of stored media using bittorrent-like protocols,” in *Proc. of NETWORKING’07*, pp. 570–581, Springer, 2007.

[75] B. Zhao, J. Lui, and D. Chiu, “Exploring the optimal chunk selection policy for data-driven p2p streaming systems,” in *Proc. of P2P’09*, pp. 271–280, IEEE, 2009.

[76] C. Arad, J. Dowling, and S. Haridi, “Developing, simulating, and deploying peer-to-peer systems using the kompics component model,” in *Proc. of COMSWARE’09*, p. 16, ACM, 2009.

[77] C. Arad, J. Dowling, and S. Haridi, “Message-passing concurrency for scalable, stateful, reconfigurable middleware,” *Proc. of Middleware’12*, pp. 208–228, 2012.

[78] Y. Lu, B. Fallica, F. Kuipers, R. Kooij, and P. Mieghem, “Assessing the quality of experience of sopcast,” *International Journal of Internet Protocol Technology*, vol. 4, no. 1, pp. 11–23, 2009.

[79] R. Kumar and K. Ross, “Peer-assisted file distribution: The minimum distribution time,” in *Proc. of HOTWEB’06*, pp. 1–11, IEEE, 2006.

[80] R. Sweha, V. Ishakian, and A. Bestavros, “Angels in the cloud: A peer-assisted bulk-synchronous content distribution service,” in *Proc. of CLOUD’11*, pp. 97–104, IEEE, 2011.

[81] R. Sweha, V. Ishakian, and A. Bestavros, “Angelcast: cloud-based peer-assisted live streaming using optimized multi-tree construction,” in *Proc. of MMsys’12*, pp. 191–202, ACM, 2012.

[82] A. Montresor and L. Abeni, “Cloudy weather for p2p, with a chance of gossip,” in *Proc. of P2P’11*, pp. 250–259, IEEE, 2011.

[83] Y. Wu, C. Wu, B. Li, X. Qiu, and F. Lau, “Cloudmedia: When cloud on demand meets video on demand,” in *Proc. of ICDCS’11*, pp. 268–277, IEEE, 2011.

[84] X. Jin and Y. Kwok, “Cloud assisted p2p media streaming for bandwidth constrained mobile subscribers,” in *Proc. of ICPADS’10*, pp. 800–805, IEEE, 2010.

[85] K. Sripanidkulchai, A. Ganjam, B. Maggs, and H. Zhang, “The feasibility of supporting large-scale live streaming applications with dynamic application end-points,” in *Proc. of SIGCOMM’04*, pp. 107–120, ACM, 2004.

[86] K. Sripanidkulchai, B. Maggs, and H. Zhang, “An analysis of live streaming workloads on the internet,” in *Proc. of IMC’04*, pp. 41–54, ACM, 2004.

[87] A. Payberah, H. Kavalionak, V. Kumaresan, A. Montresor, and S. Haridi, “Clive: Cloud-assisted p2p live streaming,” in *Proc. of P2P’12*, pp. 79–90, IEEE, 2012.

[88] A. Montresor and A. Ghodsi, “Towards robust peer counting,” in *Proc. of P2P’09*, pp. 143–146, IEEE, 2009.

[89] “Amazon elastic compute cloud (Amazon EC2).” http://aws.amazon.com/ec2/, [Online; accessed 20-Nov-2012].

[90] “Amazon simple storage service (Amazon S3).” http://aws.amazon.com/s3/, [Online; accessed 20-Nov-2012].

[91] S. Goel and R. Buyya, "Data replication strategies in wide area distributed systems," tech. rep., 2006.

[92] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, *et al.*, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, 2010.

[93] J. Sacha, J. Napper, C. Stratan, and G. Pierre, "Adam2: Reliable distribution estimation in decentralised environments," in *Proc. of ICDCS'10*, pp. 697–707, IEEE, 2010.

[94] M. Haridasan and R. van Renesse, "Gossip-based distribution estimation in peer-to-peer networks," in *Proc. of IPTPS'08*, USENIX, 2008.

[95] D. Stutzbach and R. Rejaie, "Understanding churn in peer-to-peer networks," in *Proc. of IMC'06*, pp. 189–202, ACM, 2006.

[96] G. Schay, *Introduction to probability with statistical applications*. Birkhäuser, 2007.

[97] B. Ford, P. Srisuresh, and D. Kegel, "Peer-to-peer communication across network address translators," in *Annual Technical Conference*, USENIX, 2005.

[98] R. Roverso, S. El-Ansary, and S. Haridi, "Natcracker: Nat combinations matter," in *Proc. of ICCCN'09*, pp. 1–7, IEEE, 2009.

[99] S. Niazi and J. Dowling, "Usurp: Distributed nat traversal for overlay networks.," in *Proc. of DAIS'11*, pp. 29–42, Springer, 2011.

[100] L. D'Acunto, J. Pouwelse, and H. Sips, "A measurement of nat and firewall characteristics in peer-to-peer systems," in *Proc. of ASCI'09*, pp. 1–5, Advanced School for Computing and Imaging (ASCI), 2009.

Swedish Institute of Computer Science

SICS Dissertation Series

1. Bogumil Hausman, Pruning and Speculative Work in OR-Parallel PROLOG, 1990.
2. Mats Carlsson, Design and Implementation of an OR-Parallel Prolog Engine, 1990.
3. Nabiel A. Elshiewy, Robust Coordinated Reactive Computing in SANDRA, 1990.
4. Dan Sahlin, An Automatic Partial Evaluator for Full Prolog, 1991.
5. Hans A. Hansson, Time and Probability in Formal Design of Distributed Systems, 1991.
6. Peter Sjödin, From LOTOS Specifications to Distributed Implementations, 1991.
7. Roland Karlsson, A High Performance OR-parallel Prolog System, 1992.
8. Erik Hagersten, Toward Scalable Cache Only Memory Architectures, 1992.
9. Lars-Henrik Eriksson, Finitary Partial Inductive Definitions and General Logic, 1993.
10. Mats Björkman, Architectures for High Performance Communication, 1993.
11. Stephen Pink, Measurement, Implementation, and Optimization of Internet Protocols, 1993.
12. Martin Aronsson, GCLA. The Design, Use, and Implementation of a Program Development System, 1993.
13. Christer Samuelsson, Fast Natural-Language Parsing Using Explanation-Based Learning, 1994.
14. Sverker Jansson, AKL - - A Multiparadigm Programming Language, 1994.
15. Fredrik Orava, On the Formal Analysis of Telecommunication Protocols, 1994.
16. Torbjörn Keisu, Tree Constraints, 1994.
17. Olof Hagsand, Computer and Communication Support for Interactive Distributed Applications, 1995.
18. Björn Carlsson, Compiling and Executing Finite Domain Constraints, 1995.
19. Per Kreuger, Computational Issues in Calculi of Partial Inductive Definitions, 1995.

20. Annika Waern, Recognising Human Plans: Issues for Plan Recognition in Human-Computer Interaction, 1996.
21. Björn Gambäck, Processing Swedish Sentences: A Unification-Based Grammar and Some Applications, 1997.
22. Klas Orsvärn, Knowledge Modelling with Libraries of Task Decomposition Methods, 1996.
23. Kia Höök, A Glass Box Approach to Adaptive Hypermedia, 1996.
24. Bengt Ahlgren, Improving Computer Communication Performance by Reducing Memory Bandwidth Consumption, 1997.
25. Johan Montelius, Exploiting Fine-grain Parallelism in Concurrent Constraint Languages, 1997.
26. Jussi Karlgren, Stylistic experiments in information retrieval, 2000.
27. Ashley Saulsbury, Attacking Latency Bottlenecks in Distributed Shared Memory Systems, 1999.
28. Kristian Simsarian, Toward Human Robot Collaboration, 2000.
29. Lars-åke Fredlund, A Framework for Reasoning about Erlang Code, 2001.
30. Thiemo Voigt, Architectures for Service Differentiation in Overloaded Internet Servers, 2002.
31. Fredrik Espinoza, Individual Service Provisioning, 2003.
32. Lars Rasmusson, Network capacity sharing with QoS as a financial derivative pricing problem: algorithms and network design, 2002.
33. Martin Svensson, Defining, Designing and Evaluating Social Navigation, 2003.
34. Joe Armstrong, Making reliable distributed systems in the presence of software errors, 2003.
35. Emmanuel Frécon, DIVE on the Internet, 2004.
36. Rickard Cöster, Algorithms and Representations for Personalised Information Access, 2005.
37. Per Brand, The Design Philosophy of Distributed Programming Systems: the Mozart Experience, 2005.
38. Sameh El-Ansary, Designs and Analyses in Structured Peer-to-Peer Systems, 2005.
39. Erik Klintskog, Generic Distribution Support for Programming Systems, 2005.
40. Markus Bylund, A Design Rationale for Pervasive Computing - User Experience, Contextual Change, and Technical Requirements, 2005.
41. Åsa Rudström, Co-Construction of hybrid spaces, 2005.
42. Babak Sadighi Firozabadi, Decentralised Privilege Management for Access Control, 2005.
43. Marie Sjölinder, Age-related Cognitive Decline and Navigation in Electronic Environments, 2006.

44. Magnus Sahlgren, The Word-Space Model: Using Distributional Analysis to Represent Syntagmatic and Paradigmatic Relations between Words in High-dimensional Vector Spaces, 2006.
45. Ali Ghodsi, Distributed k-ary System: Algorithms for Distributed Hash Tables, 2006.
46. Stina Nylander, Design and Implementation of Multi-Device Services, 2007
47. Adam Dunkels, Programming Memory-Constrained Networked Embedded Systems, 2007
48. Jarmo Laaksolahti, Plot, Spectacle, and Experience: Contributions to the Design and Evaluation of Interactive Storytelling, 2008
49. Daniel Gillblad, On Practical Machine Learning and Data Analysis, 2008
50. Fredrik Olsson, Bootstrapping Named Entity Annotation by Means of Active Machine Learning: a Method for Creating Corpora, 2008
51. Ian Marsh, Quality Aspects of Internet Telephony, 2009
52. Markus Bohlin, A Study of Combinatorial Optimization Problems in Industrial Computer Systems, 2009
53. Petra Sundström, Designing Affective Loop Experiences, 2010
54. Anders Gunnar, Aspects of Proactive Traffic Engineering in IP Networks, 2011
55. Preben Hansen, Task-based Information Seeking and Retrieval in the Patent Domain: Process and Relationships, 2011
56. Fredrik Österlind, Improving low-power wireless protocols with timing-accurate simulation, 2011
57. Ahmad Al-Shishtawy, Self-Management for Large-Scale Distributed Systems, 2012
58. Henrik Abrahamsson, Network overload avoidance by traffic engineering and content caching, 2012
59. Mattias Rost, Mobility is the Message: Experiment with Mobile Media Sharing, 2013