**ROYAL INSTITUTE
OF TECHNOLOGY**

Distributed Optimization of P2P Media Delivery Overlays

AMIR H. PAYBERAH

Licentiate Thesis
Stockholm, Sweden 2011

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av teknologie licentiatesexamen i datalogi Fredag den 3 Juni 2011 klockan 10.00 i sal D i Forum IT-Universitetet, Kungl Tekniskahögskolan, Isajordsgatan 39, Kista.

Abstract

Media streaming over the Internet is becoming increasingly popular. Currently, most media is delivered using global content-delivery networks, providing a scalable and robust client-server model. However, content delivery infrastructures are expensive. One approach to reduce the cost of media delivery is to use *peer-to-peer (P2P) overlay networks*, where nodes share responsibility for delivering the media to one another.

The main challenges in P2P media streaming using overlay networks include: (i) nodes should receive the stream with respect to certain timing constraints, (ii) the overlay should adapt to the changes in the network, e.g., varying bandwidth capacity and join/failure of nodes, (iii) nodes should be intentivized to contribute and share their resources, and (iv) nodes should be able to establish connectivity to the other nodes behind NATs. In this work, we meet these requirements by presenting P2P solutions for live media streaming, as well as proposing a distributed NAT traversal solution.

First of all, we introduce a distributed market model to construct an approximately minimal height multiple-tree streaming overlay for content delivery, in *gradienTv*. In this system, we assume all the nodes are cooperative and execute the protocol. However, in reality, there may exist some opportunistic nodes, *free-riders*, that take advantage of the system, without contributing to content distribution. To overcome this problem, we extend our market model in *Sepidar* to be effective in deterring free-riders. However, gradienTv and Sepidar are tree-based solutions, which are fragile in high churn and failure scenarios. We present a solution to this problem in *GLive* that provides a more robust overlay by replacing the tree structure with a mesh. We show in simulation, that the mesh-based overlay outperforms the multiple-tree overlay. Moreover, we compare the performance of all our systems with the state-of-the-art NewCoolstreaming, and observe that they provide better playback continuity and lower playback latency than that of NewCoolstreaming under a variety of experimental scenarios.

Although our distributed market model can be run against a random sample of nodes, we improve its convergence time by executing it against a sample of nodes taken from the *Gradient overlay*. The Gradient overlay organizes nodes in a topology using a local utility value at each node, such that nodes are ordered in descending utility values away from a core of the highest utility nodes. The evaluations show that the streaming overlays converge faster when our market model works on top of the Gradient overlay.

We use a gossip-based peer sampling service in our streaming systems to provide each node with a small list of live nodes. However, in the Internet, where a high percentage of nodes are behind NATs, existing gossiping protocols break down. To solve this problem, we present *Gozar*, a NAT-friendly gossip-based peer sampling service that: (i) provides uniform random samples in the presence of NATs, and (ii) enables direct connectivity to sampled nodes using a fully distributed NAT traversal service. We compare Gozar with the state-of-the-art NAT-friendly gossip-based peer sampling service, Nylon, and show that only Gozar supports one-hop NAT traversal, and its overhead is roughly half of Nylon's.

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

Acknowledgements

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 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 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 project. I would also like to thank Dr. Ali Ghodsi, for acquaining me with peer-to-peer overlays and guided me in the first year of PhD, as well as during my Master studies.

I am thankful to Professor Vladimir Vlassov for his valuable feedbacks on this thesis. I am also grateful to Sverker Janson for giving me the chance to work as a member of CSL group at SICS. I acknowledge the help and support by Dr. 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 Tallat Mahmood Shafaat, Ahmad Al-Shishtawy and Roberto Roverso, 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

Part I

Thesis Overview

Chapter 1

Introduction

Media streaming over the Internet is getting more popular everyday. The conventional solution for such applications 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 support by Internet Service Providers. The approach, used in this thesis, is *Application Level Multicast (ALM)*, which uses *overlay networks* to distribute large-scale media streams to a large number of clients. 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 to the other nodes in a network. In this model, nodes who 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.

Media streaming using P2P overlays is a challenging problem. To have a smooth media playback, data blocks should be received with respect to certain timing constraints. Otherwise, either the quality of the playback is reduced or its continuity is disrupted. Moreover, in live streaming, it is expected that at any moment, clients receive points of the media that are close in time, ideally, 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 can see the goal happening. Satisfying these timing requirements is more challenging in a dynamic network, where nodes join/leave/fail continuously and concurrently, and the network capacity changes over time. Yet another challenge for P2P overlays in the Internet is the presence of Network Address Translation gateways (NATs). Nodes that reside behind NATs, *private nodes*, do not support direct connectivity

---

[1]http://www.thetechherald.com/article.php/200942/4604/YouTube-s-daily-hit-rate-more-than-a-billion

by default. Furthermore, the nodes should be incentivized to contribute and share their resources in a P2P overlay. Otherwise, the opportunistic nodes, called *free-riders*, can take advantage of a system, without contributing to content distribution.

Many different solutions are already proposed for P2P media streaming, but few of them are able to satisfy all the above mentioned requirements. We believe this is partly because some of these requirements are conflicting. For example, in order to provide a constant high quality stream, users should store the media in their buffer for a while before they start to play; which will result in a high playback latency and start up delay.

1.1   Contribution

In this dissertation, we present our P2P live media streaming solution in the form of three systems: *gradienTv* [1], *Sepidar* [2], and *GLive* [3]. In gradienTv and 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 all these streaming overlays, the nodes with higher available upload bandwidth that can serve relatively more nodes are positioned closer to the media source. This structure reduces the average number of hops from nodes to the media source; reducing both the probability of a streaming disruptions and playback latency at nodes. Nodes are also incentivized to provide more upload bandwidth, as nodes that contribute more upload bandwidth have relatively higher playback continuity and lower latency than the nodes further to the media source.

To construct our streaming overlays, firstly in gradienTv, we present a distributed market model inspired by the auction algorithm [4]. Our distributed market model differs from the centralized implementations of the auction algorithm, in that we do not rely on a central server with a global knowledge of all participants. In our model, each node, as an auction participant, has only partial information about the system. Nodes continuously exchange their information, in order to acquire more knowledge about other participating nodes in the system. There are different options for how communication between nodes could be implemented. For example, a naive solution could use flooding, but it is costly in terms of bandwidth consumption, and therefore is not scalable. On the other hand, the communication could be based on random walks or sampling from a random overlay, but we show in the papers [2, 3] that random sampling has slow convergence time. To enable faster convergence of the streaming overlay, our distributed market model acquires knowledge of the system by sampling nodes using the gossip-generated *Gradient overlay* network [5, 6]. The Gradient overlay facilitates the discovery of neighbours with similar upload bandwidth.

The Gradient overlay is a class of P2P overlays that arrange 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. In our implementation, we use

upload bandwidth as the utility value, however, the model can easily be extended to include other characteristics such as node uptime, load and reputation.

The free-riding problem, as one of the problems in P2P streaming systems, is not considered in gradienTv. We address 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 use a gossip-based peer sampling service (PSS) as a building block of our systems. A PSS periodically provides a node with 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 private nodes, traditional gossip-based PSS breaks down. To overcome this problem, we present *Gozar*, a NAT-friendly gossip-based PSS that uses existing *public nodes* in the system (nodes not behind NATs) to help in NAT traversal.

Our contributions in this thesis include:

- a distributed market model to construct P2P streaming overlays, firstly as a tree-based overlay, Sepidar and gradienTv, and secondly as a mesh-based overlay, GLive. We also show how the Gradient overlay can improve the convergence time of the distributed market model in comparison with a random network,

- two solutions to overcome the free-riding problem in a tree-based (Sepidar) and a mesh-based (GLive) overlay,

- Gozar, a gossip-based peer sampling service that provides uniform random samples in the presence of NATs, and enables direct connectivity to the sampled nodes using a fully distributed NAT traversal service.

1.2 Assumptions

We assume a network of nodes that communicate through message passing. New nodes may join the network at any time to watch the video. Existing nodes may leave the system either voluntarily or by crashing.

Nodes are not assumed to be cooperative; nodes may execute protocols that attempt to download data blocks without forwarding it to other nodes. We do not, however, address the problem of nodes colluding to receive the video stream.

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 the P2P media streaming and introduce a frame-

work for classifying and comparing different P2P streaming solutions. More-
over, we go through the basic concepts of the peer sampling services and
introduce the Gradient overlay. Furthermore, we show the effects of NATs on
the behaviour of P2P applications, and explore the existing NAT traversal
solutions.

- In chapter 3, we present our distributed market model to construct tree-
  based and mesh-based P2P streaming overlays. Moreover, we show how we
  use the Gradient overlay to improve the convergence time of our systems.
  Additionally, we present our free-rider detector mechanism, and finally explain
  our NAT-friendly gossip-based peer sampling service.

- In chapter 4, we show our future research directions, and we conclude the
  work in this chapter.

- In chapter 5, 6, 7 and 8, we present our related papers covered in this disser-
  tation.

Chapter 2

Background

In this chapter we explore the necessary background for the thesis. First of all, we review the main concepts of P2P media streaming systems, e.g., how to construct and maintain streaming overlays. 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.

2.1 P2P media streaming

Each P2P media streaming solution should provide answer to the following two main questions:

1. What overlay topology is built for content distribution?

2. How to construct and maintain the overlay?

Following, we study a number of answers to these questions.

2.1.1 Content distribution overlay topology

The first question a P2P streaming application needs to answer is that what overlay topology should be constructed for content distribution. In general, three main topologies are used for this purpose:

- Tree-based topologies

- Mesh-based topologies

- Hybrid topologies

Single tree structure is one of the earliest overlay for this purpose [7]. In this model, a tree overlay is constructed on top of all the nodes in a system, and each node *pushes* data it receives to a number of other nodes. A node that forwards data is called a *parent* node, and a node that receives it, is a *child* node.

Fast data distribution among nodes is the main advantage of this model. However, this structure is very fragile to node failures. If a node fails, all other nodes that are located in the subtree rooted at the failed node, do not receive the content any more, meanwhile they rejoin the overlay. Moreover, the load distribution among nodes is not fair. The interior nodes carry the contents and the leaf nodes do not contribute in data dissemination, while the number of leaf nodes increases much faster than the number of interior nodes. Furthermore, the interior nodes may not have enough upload bandwidth to transfer contents with the required rate. Therefore, those nodes become bottleneck, and the nodes in their subtree may not receive the data on time. Nevertheless, many P2P streaming systems have used the single tree structure, e.g., Climber [8], ZigZag [9], NICE [10], and [11].

To overcome the single tree overlay problems, SplitStream [12] introduced the *multiple-tree* structure. In multiple-tree overlays, the media stream is split into a number of sub-streams or *stripes*, and each stripe is delivered to nodes through a separate tree. Therefore, unlike the single tree structure, a child node may receive the data from multiple parents that each one forwards the contents of one stripe. This helps to have a more resilient overlay in the presence of failures, because, if a child node loses one of its parents, it still can get the other stripes from other parents. In addition, a node can play different roles in different trees, e.g., a leaf node in one tree can be an interior node in another tree, so the load is distributed more fairly among the nodes. However, the complexity of the multiple-tree structure and the time to construct the trees are the problems of this topology. Moreover, although, a node can receive the blocks of each stripe independently, it loses the contents of one stripe, if the providing parent of that stripe fails. Therefore, meanwhile the child node, whose father failed, finds an appropriate parent for that stripe, the node misses the content of that stripe. Sepidar [2], gradienTv [1], Orchard [13], ChunkySpread [14], and CoopNet [15] are the solutions in this class.

Rajaee et al. show in [16] that mesh overlays possess a higher performance over the tree-based approaches. In mesh-based overlays, unlike tree-based structures that data is pushed through trees, nodes *pull* contents from their neighbours in a mesh. Each node periodically sends its content availability information or *buffer map* to its neighbours. The other nodes, then, use this information to schedule and request data from their neighbours. Since the neighbours of nodes are updated periodically, it is highly resilient to nodes failure. However, it is subject to unpredictable latencies due to the frequent exchange of notifications and requests [7]. GLive [3], Gossip++ [17], DONet/Coolstreaming [18], Chainsaw [19], PULSE [20] and [21] are the systems that use the mesh structure for data dissemination.

An alternative solution for data dissemination is the *hybrid* overlay that uses the benefits of tree-based approaches with the advantages of mesh-based approaches. Example systems include CliqueStream [22], mTreebone [23], NewCoolStreaming

[24], Prime [25], and [26].

2.1.2 Constructing/maintaining the overlay

The second fundamental problem is how to construct and maintain the content distribution overlay, or in other words, how nodes discover the other supplying nodes. The main solutions in literatures for this question are:

- Centralized method

- Hierarchical method

- Controlled flooding method

- DHT-based method

- Gossip-based method

The *centralized method* is a solution used mostly in initial 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 [15] and DirectStream [27] are two sample systems that use the cenrtal method. Since the central server has a global view to the overlay network, it can handle node joins and leaves 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 the *hierarchical method*. This approach is used in several systems, such as Nice [10], ZigZag [9], and Bulk Tree [28]. For example, in Nice and ZigZag, 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 *controlled flooding*, which is originally proposed by Gnutella [29]. GnuStream [30] 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 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 [31] and Pastry [32]. SplitStream [12] and [26] 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 [18], PULSE [20] and [21] use a gossip-generated random overlay network to search for the supplying nodes. We use the gossip-generated Gradient overlay [5] for node discovery in gradineTv [1], Sepidar [2], and GLive [3]. 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 protocol is scalable and failure-tolerant, but because of the randomness property of the neighbour selection, sometimes the appropriate providers are not found in a reasonable time.

2.2 Peer sampling service

Peer sampling services (PSS) have been widely used in large scale distributed applications, such as information dissemination [33], aggregation [34], and overlay topology management [6, 35]. Gossiping algorithms are the most common approach to implementing a PSS [36–40]. In gossip-based PSS', protocol execution at each node is divided into periodic cycles. In each cycle, every node selects a node from its partial view to exchange a subset of its partial view with the selected node. Both nodes subsequently update their partial views using the received node descriptors.

Implementations vary based on a number of different policies [41]:

1. *Node selection*: determines how a node selects another node to exchange information with. It can be either 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 the other 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 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 [40, 41]. 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, and the clustering coefficient of a node is the number of links between the neighbors of the node divided by all possible links.

2.3 The Gradient overlay

The Gradient overlay is a class of P2P overlays that arrange 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 [5, 6].

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.

2.4 The NAT problem

In [42], Kermarrec et al. evaluated the impact of NATs on traditional gossip-based PSS'. They showed that the network becomes partitioned when the number of nodes behind NAT, called *private nodes*, exceeds a certain threshold. In principle, existing PSS' could be adapted to work over NATs. This can be done by having all nodes, run a protocol to identify their NAT type, such as STUN [43]. Then, nodes identified as private keep open a connection to a third party rendezvous server. When a node wishes to gossip with a private node, it can request a connection to the private node via the rendezvous server. The rendezvous server then executes a NAT traversal protocol to establish a direct connection between the two nodes.

A model of NAT behaviour is necessary to enable private nodes to identify what type of NAT they reside behind. When a node attempts to establish a direct connection to a private node, the NAT types of both nodes are then used to determine what NAT traversal algorithm should be used to traverse any intermediary NATs.

When determining a node's NAT type, the main observable behaviour of a NAT is that it maps an IP address/port pair at a private node to a public port on a public interface of the NAT. IP packets sent from the address/port at the private node to a destination outside the NAT are translated by the NAT replacing the packet's private IP address and port number with the public IP and mapped port on the NAT. NAT behaviour is classified according to (i) the port mappings created, (ii) how and when the NAT generates new mapping rules and updates existing rules, and (iii) the type of filtering the NAT performs on packets sent to a mapped port on the NAT. Other aspects of NATs that we do not model, but have less impact on the success of NAT traversal, are multi-level NATs and whether the NAT has multiple public interfaces.

The earliest model of NAT behaviour was STUN that grouped NATs into four groups: full cone, restricted cone, partial cone and symmetric [44]. However, this model is quite crude, and its NAT traversal solutions ignore the fact that when two nodes both reside behind NATs, it is the combination of NAT types that determines the NAT traversal algorithm that should be used. In [45], a richer classification of NAT types is presented, that decomposes a NAT's behaviour into three main policies: *port mapping*, *port assignment* and *port filtering*. We adopt this model:

- *Port mapping*: This policy decides when to create a new mapping (NAT rule) from a private port to a public port. That is, it decides for each packet from a private node IP address/port pair whether to allocate a new public port on the NAT or reuse an existing one. Three different port mapping policies have been found in existing NATs [46]:

  1. *Endpoint-Independent (EI)*: The NAT reuses the same mapping rule for address/port pairs from the same private node. That is, all source addresses on packets sent from the private node are mapped to the same public port on the NAT, regardless of the packet's destination IP address and port.

  2. *Host-Dependent (HD)*: The NAT will reuse the same mapping rule for all address/port pairs from the same private node when the packets are destined for the same IP address. That is, for a given destination IP address (and regardless of the destination port), all source addresses on packets sent from the private node are mapped to the same public port on the NAT.

  3. *Port-Dependent (PD)*: The NAT will reuse the same mapping rule for all address/port pairs from the same private node when the packets are destined for the same IP address and port number. That is, for a given

destination IP address and port, all source addresses on packets sent from the private node are mapped to the same public port on the NAT.

The mapping policies can be ordered in terms of increasing level of difficulty for NAT traversal as $EI < HD < PD$.

- *Port assignment*: This policy decides which port should be assigned whenever a new mapping rule is created, that is a new public port is mapped to a private address/port. Three different port assignment policies have been found in existing NATs [46]:

  1. *Port-Preservation (PP)*: The NAT maps the port number at the private node to the same port number on the public interface of the NAT. This may cause a conflict if two private nodes behind the same NAT request the same port. In the case of a port mapping conflict, an alternative port assignment policy is used to assign a new port - typically either port-contiguity or random.

  2. *Port-Contiguity (PC)*: The NAT maintains an internal variable storing the most recently assigned port number. When a new mapping rule is created, the new mapping's port on the NAT is some small constant number higher than the the most recently assigned port number. In other words, for two consecutive ports mapped on the NAT, u and v, it binds $u = v + \Delta$, for some $\Delta = 1, 2, \cdots$.

  3. *Random (RD)*: The NAT maps a random public port for each new mapping rule created.

  The assignment policies can be ordered in terms of increasing level of difficulty for NAT traversal as $PP < PC < RD$.

- *Port filtering*: The port filtering policy decides whether incoming packets to a public port mapped on the NAT are forwarded to the mapped port on the private node. Three different port filtering policies have been found in existing NATs [46]:

  1. *Endpoint-Independent (EI)*: The NAT forwards all packets to the private node, regardless of the external node's IP address and port.

  2. *Host-Dependent (HD)*: The NAT filters all incoming traffic on the public port, except those packets that come from an external node with an IP address X that has previously received at least one packet from this public port.

  3. *Port-Dependent (PD)*: The NAT filters all incoming traffic on the public port, except those packets that come from an external node with IP address X and port P that has previously received at least one packet from this public port.

The filtering policies can be ordered in terms of increasing level of difficulty for NAT traversal as $EI < HD < PD$.

In addition to these three policies, it is useful to determine the length of time for which NAT mappings remain valid without packets being sent over the mapped port. A protocol for determining all three policies and the NAT mapping timeout is described in [46].

In this model of NAT behaviour, there are, in total, 27 different possible NAT types, and there are $\frac{27 \times 28}{2} = 378$ different possible NAT combinations for any two private nodes [46].

NAT traversal techniques

There are two general techniques that are used to communicate with private nodes: *hole punching* and *relaying*. 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 the two nodes and when slow connection setup times are not a problem. Relaying is preferable when the connection setup time should be short (less than one second) and small amounts of data will be sent over the connection.

- *Hole punching*: enables two nodes to establish a direct connection over intermediary NATs with the help of a third party rendezvous server [47, 48]. Connection reversal is the simplest form of hole punching, which is when a public node attempts to connect to a private node, it contacts the rendezvous server, that, in turn, requests the private node to establish a connection with the public node. Hole punching, however, more commonly refers to how mapping rules are created on NATs for a connection that is not yet established, but soon will be. Simple hole punching (SHP) [46] is a NAT traversal algorithm, where both nodes reside behind NATs and both nodes attempt to send packets to mapped ports on their respective NATs with the goal of creating NAT mappings on both sides to allow traffic to flow directly between the two nodes. SHP is feasible when (i) the filtering policy is EI or (ii) the mapping policy is EI or (iii) the mapping policy is stronger than EI and the filtering policy is weaker than PD [46]. Port prediction using contiguity (PRC) that uses port scanning is another NAT traversal algorithm that can be used when the port assignment policy is PC. Similarly, when the port assignment policy is PP, prediction using port preservation (PRP) can be used [46].

- *Relaying*: Relaying can be used either where hole punching techniques do not succeed or where hole punching takes too long to complete. In relaying, a third party relay server that has a public IP address keeps an open connection with the private node, and other nodes communicate with the private node

by sending messages to the relay node. The relay node forwards the messages to the private node and the responses to the source node. TURN [49] is a protocol for relaying messages.

Chapter 3

Thesis contribution

In this chapter, we present a summary of the thesis contribution. First, we list the publications that were produced during this work. Then, we explain our solution to construct a streaming overlay in form of the multiple-tree and the mesh. Later, we optimize our solution by sampling nodes from the Gradient overlay rather than a random network. Finally, we present our solution to solve the nodes connectivity problem in the Internet, where a high percentage of the nodes are behind NATs.

3.1   List of publications

- Amir H. Payberah, Jim Dowling, Seif Haridi, *Gozar: NAT-friendly Peer Sampling with One-Hop Distributed NAT Traversal*, in the 11th IFIP international conference on Distributed Applications and Interoperable Systems (DAIS'11), Reykjavik, Iceland, June 2011.

- Amir H. Payberah, Jim Dowling, and Seif Haridi, *GLive: The gradient overlay as a market maker for mesh-based p2p live streaming*, in the 10th IEEE International Symposium on Parallel and Distributed Computing (ISPDC'11), Cluj-Napoca, Romania, July 2011.

- Amir H. Payberah, Jim Dowling, Fatemeh Rahimian, and Seif Haridi, *Sepidar: Incentivized market-based p2p live-streaming on the gradient overlay network*, in the IEEE International Symposium on Multimedia (ISM'10), vol. 0, pp. 1–8, 2010.

- Amir H. Payberah, Jim Dowling, Fatemeh Rahimian, and Seif Haridi, *gradienTv: Market- based P2P Live Media Streaming on the Gradient Overlay*, in Lecture Notes in Computer Science (DAIS'10), pp. 212–225, Springer Berlin, Heidelberg, Jan 2010.

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

- Fatemeh Rahimian, Sarunas Girdzijauskas, Amir H. Payberah, Seif Haridi, *Vitis: A Gossip-based Hybrid Overlay for Internet-scale Publish/Subscribe*, in the 25th IEEE International Parallel & Distributed Processing Symposium (IPDPS'11), USA, May 2011.

3.2   Tree-based approach

In this section, we explain our tree-based systems: gradienTv [1] and Sepidar [2]. We show how we can model the overlay construction as an assignment problem [50], and then we present our distributed market model to solve this problem. The details of the tree-based solutions, which are published in two papers, are covered in chapters 5 and 6.

3.2.1   Problem description

We assume the media stream is split into a number of sub-streams or *stripes*, and each stripe is divided into *blocks* of equal size without any coding. Sub-streams allow more nodes to contribute bandwidth and enable more robust systems through redundancy [12]. Every block has a sequence number to represent its playback order in the stream. Nodes can retrieve any stripe independently from any other node that can supply it. The number of stripes that nodes are willing and able to forward and to download at the same time is defined as its number of *upload slots* and *download slots*, respectively.

The problem we address here is how to deliver a video stream from a source as multiple stripes over multiple overlay trees that each form the structure of a *Degree-Height-Balanced (DHB) tree*. A DHB tree is a *height-balanced* tree, where the height of the two child subtrees from any node differ by at most one. A DHB tree is also a *degree-balanced* tree, where nodes lower in the tree will have less than or an equal number of upload slots compared to nodes higher in the tree. That is, the out-degrees of nodes at depth l are less than or equal to the out-degrees of nodes at depth $l-1$. The root node is at depth zero.

This problem can be represented as the *assignment problem* [50]. Every node n contains an equal number of download slots, and a variable number of upload slots. The set of all download and upload slots are denoted by D and U, respectively. In order to forward the stream to all nodes, every download slot needs to be assigned to an upload slot, and download slots at a node must download different stripes. We define an assignment or a *mapping m_{ij}*, for a stripe S from a parent node i to a child node j, as a pair containing one upload slot at i and one download slot at j:

$$m_{ij} = (u_i, d_j) : u \in U, d \in D, i, j \in N, i \neq j, \tag{3.1}$$

where N is the set of all nodes, and with the constraint that the slots are not located at the same node. A *cost function* is defined for a mapping m_{ij} as the distance from the parent node to the source for that stripe in terms of the number of hops, that is,

$$c(m_{ij}) : m_{ij} \rightarrow number\,of\,hops\,from\,i\,to\,root. \tag{3.2}$$

We define a *complete assignment A* as a set of mappings, where each download slot is assigned to a different upload slot, that is, every download slot in D is a member of a mapping in A. For the system as a whole, we define the *Resource Index (RI)* as the ratio of the number of upload slots to the number of download slots, $RI = \frac{|U|}{|D|}$. To have a complete assignment, the RI of the system must be greater than one, that is, there must be at least as many upload slots as download slots. The total cost of a complete assignment is calculated as follows:

$$c(A) = \sum_{m \in A} c\,(m) \tag{3.3}$$

The goal of our system is to minimize the cost function in equation 3.3. Here, we show that by building the DHB tree, we minimize the total cost function.

Theorem 1. *If T is a DHB tree, then the cost function in equation 3.3 is minimized.*

Proof. See the appendix 3.A. □

For live streaming, we have real-time constraints in solving this assignment problem. Good solutions should allow child nodes to be assigned a parent as quickly as possible, to enable quick viewing of the stream. Centralized solutions to this problem are possible for small system sizes. For example, if all nodes send their number of upload slots to a central server, the server can use any number of algorithms that solve linear sum assignments, such as the auction algorithm [4], the Hungarian method [51], or more recent high-performance parallel algorithms [50]. We briefly sketch a possible solution with the auction algorithm.

The auction algorithm can be used to solve the assignment problem by n download slots competing for m upload slots through iteratively increasing in their prices in competitive bidding, where $RI = \frac{m}{n} \geq 1$. Each matching between a download slot i and an upload slot j is associated with a benefit a_{ij}, and the goal of the auction is to assign download to upload slots such that the total benefit for all matchings is maximized: $\sum_{i=1}^{n} a_{ij}$.

However, download slots have a certain amount of currency with which to find a matching of maximum benefit to it. Download slots search for upload slots they can afford that have the highest net value, that is, upload slots whose benefit minus their current price is highest. The algorithm then consists of two iterative phases: a bidding phase and an assignment phase. Download slots first bid for upload

slots of highest net value, and then upload slots assign the download slots with the highest bids. These two phases iterate, and prices for upload slots increase until all download slots have been assigned an upload slot.

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 [50], breaking our real-time constraints. In the next subsection, we present a distributed market model, inspired by the auction algorithm as an approximate solution to this problem.

3.2.2   Constructing the multiple-tree overlay

Our market model is based on minimizing costs (instead of maximizing benefits) through nodes iteratively bidding for upload slots. We use the following three properties, calculated at each node, to approximately build the minimum delay 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 establishing 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. In addition, the price of a node that has a free-riding child is zero.

3. *Cost*: the cost of an upload slot at a node for a particular stripe is the distance from that node to the root (the media server) for that stripe, see equation 3.2. Since the media stream consists of several stripes, nodes may have different costs for different stripes. The lower the depth a node has for a stripe (the lower its cost), the more desirable a parent it is for that stripe. Nodes constantly try to reduce their costs over all their parent connections by bidding for connections to lower depth nodes.

Nodes in these system compete to become *children* of nodes that are closer to the 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. The parent then *pushes* to the child subsequent blocks in the stripe, as long as it remains the child's parent. Children can proactively switch parents when the market-modeled benefit of switching is greater than the cost of switching.

Our market model could be best described as an approximate auction algorithm, where there is *no reserve price*. For each stripe, child nodes place bids of their entire money for upload slots at the parent nodes with lowest cost (depth). Although the money is not used up, it can be reused for other bids for other connections. A

parent node sets a price of zero for an upload slot when at least one of its upload slots is unassigned. Thus, the first bid for an upload slot will always win (no reserve price), 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. 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 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. The restartable auction is only implemented in Sepidar, while gradienTv does not resolve the free-rider problem. In the following subsection we show how a parent node detects its free-rider children in Sepidar.

3.2.3   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 *strong completeness* property. By strong completeness property, we mean that, if a *non-freerider* node does not have free upload slots, it eventually detects all its free-riding children. Nodes identify free-riders through transitive auditing using their children's children. The readers are kindly referred to see chapter 6, for more details of this procedure.

After detecting a node as a free-rider, the parent node p, decreases its own price (p's price) to zero and as a *punishment* considers the free-rider node q as its child with the lowest money. On the next bid from another node, p replaces the free-rider node with the new node. Therefore, if a node claims it has more upload bandwidth 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   Mesh-based approach

In GLive [3], we use our market model to construct a mesh overlay for content delivery. In the following subsections, we present the problem and explain the differences between the mesh-based and the tree-based approaches. The results of this work is published as a paper [3], which is available in chapter 7.

3.3.1   Problem description

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 B blocks of equal size without any coding. Every block $b_i \in B$ has a sequence number to represent its playback order in the stream. Nodes can pull any 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* to partners and accept a bounded number of *upload connections* from partners over which blocks are downloaded and uploaded, respectively. We define a node q as the *parent* of a *child* p, if an upload connection of q is bounded to a download connection of p. Unlike the tree-based approach that assigns upload slots to download slots of nodes for each stripe, here, we need to find the mapping of upload connections to download connections to distribute each block among all the nodes.

Similar to the problem description in subsection 3.2.1, we define the set of all download and upload connections as D and U, respectively. In order to receive a block, a node requires one of its download connection to be assigned to an upload connection over which the block will be copied. We define an assignment or a *mapping m_{ijk}*, from a node i to a node j for block b_k, as a triplet containing one upload connection at i and one download connection at j for block b_k:

$$m_{ijk} = (u_i, d_j, b_k) : u \in U, d \in D, b \in B, i, j \in N, i \neq j \qquad (3.4)$$

where N is the set of all nodes, b_k is block k from the set of all blocks B, and the connection from i to j is between two different nodes. We keep the definition of the cost function of each mapping, the complete assignment and the total cost of a complete assignment as it is in subsection 3.2.1.

The goal of our system is to minimize the cost function in equation 3.3 for every block $b \in B$, such that a shortest path tree is constructed over the set of available connections for every block. If the set of nodes, connections, and the upload bandwidth of all nodes is static for all blocks B, then we can solve the same assignment problem $|B|$ times. However, P2P systems, typically have churn (nodes join and fail) and available bandwidth at nodes changes over time, so we have to solve a slightly different assignment problem every time a node join, exits or a node's bandwidth changes.

In the next subsection, we present a modified version of the distributed auction algorithm introduced in subsection 3.2.2 to construct a mesh overlay.

3.3.2   Constructing the mesh overlay

To build a mesh overlay, we keep the definition of the price and the cost as it is in subsection 3.2.2. We redefine the money as the total number of blocks uploaded to children during the last 10 seconds.

Each node periodically sends its money, cost and price to all its *partners*, which are its neighbours in the mesh. For each of its download connections, a child node p sends a bid request to nodes that: (i) have lower cost than one of the existing parents assigned to download connections in p, and (ii) the price of a connection is less than p's money.

A parent node who receives a bid request accepts it, if: (i) it has a free upload connection (its cost is zero), or (ii) it has assigned an upload connection to another node with a lower amount of money. If the 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. When a child node receives the acceptance message from another node, it assigns one of its download connections to the upload connection of the parent. Since a node may send more connection requests than its has download connections, it might receive more acceptance messages than it needs. In this case, if all its download connections are already assigned, it checks the cost of all its assigned parents and finds the one with the highest cost. If the cost of that parent is higher than the new received acceptance message, it releases the connection to that parent and accepts the new one, otherwise it ignores the received message.

Although there is no guarantee that the parent will forward all blocks over its connection to a child, parents who forward a relatively lower number of blocks will be removed as children of their parents. Nodes that claim that they have forwarded more blocks than they actually have forwarded are removed as children, and, an auction is restarted for the removed child's connection. Nodes are incentivized to increase the upper bound on the number of their upload connections, as it will help increase their upload rate and, hence, their attractiveness as children for parents closer to the root.

3.3.3 Handling free-riders

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, and these scores are periodically sent to the parents of their parents. The details of the scoring mechanism is covered in chapter 7.

When a node with no free upload connection receives a connection request, it sorts its children based on their latest scores. If an existing child has a score less than a threshold s, then the 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 whose score is less than s, then the lowest score is selected. If all children have a score higher than s, then the parent accepts the connection if the connecting node has offers more money than the lowest money of its existing children. When the parent accepts such a connection, it then abandons (removes the connection to) the child with the lowest money. The abandoned child then has to search for and bid for a new connection to a new parent.

3.4   The Gradient overlay as a market-maker

One difference between our market model with the auction algorithm is that our market model is decentralized; nodes have only a partial (changing) view of a small number of nodes in the system with whom they can bid for upload slots. 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 value. The 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 chapters 6 and 7.

It is important that nodes' partial views enable them to find good matching parents quickly. We use the Gradient overlay [5, 6] 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. As such, this algorithm gives us an approximate solution to the assignment problem.

The details of the constructing the Gradient overlay is presented in chapter 5.

3.5   Handling the NAT problem

As mentioned in section 2.4, when a high percentage of nodes are behind NATs, it is impossible to create direct connection between those nodes, and it breaks down the existing gossip-based PSS. In Gozar, we address this problem by designing a gossip-based NAT-friendly PSS that supports distributed NAT traversal using a system composed of both public and private nodes.

The challenge with gossiping PSS 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 NAT traversal is the only really viable option, if one is to preserve fairness. However, if a system is sensitive to long connection establishment times, then NAT traversal is a problem, which affects both options 2 and 3. If the amount of data being sent is small, and fast connection setup times are important, then relaying is considered an acceptable solution. If it is important to distribute load as fairly as possible between public and private nodes, then option 3 is attractive. In existing systems, Skype supports both options 1 and 2, and can considered to have a solution to the fairness problem that, by virtue of its widespread adoption, can be considered acceptable to their user community [52].

Gozar is a NAT-friendly gossip-based peer sampling protocol with support for distributed NAT traversal. Our implementation of Gozar is based on the tail, push-pull and swapper policies for node selection, view exchange and view selection (see section 2.2). In Gozar, node descriptors are augmented with the node's NAT type (private or public) and the mapping, assignment and filtering policies determined for the NAT [46]. A STUN-like protocol is run on a bootstrap server when a node joins the system to determine its NAT type and policies. We consider running STUN once at bootstrap time acceptable, as, although some corporate NAT devices can change their NAT policies dynamically, the vast majority of consumer NAT devices have a fixed NAT type and fixed policies.

In Gozar, each private node connects to one or more public nodes, called *partners*. Private nodes discover potential partners 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 partner 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 partners 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' partners also provide a rendezvous service to enable applications that sample nodes using the PSS to connect to them using a hole punching algorithm (if hole punching is possible).

Table 3.1: Number of nodes of subtree T_a and T_b in different levels

Depth	T_a	T_b	Comments
$l+0$	$N_0 = 1$	$M_0 = 0$	$m_0 = MD(T, l+0)$
$l+1$	$N_1 = k$	$M_1 = 1$	$m_1 = MD(T, l+1)$
$l+2$	$N_2 = \displaystyle\sum_{i=1}^{N_1} k_i$	$M_2 = M_1 \times m_0$	$m_2 = MD(T, l+2)$
$l+3$	$N_3 = \displaystyle\sum_{i=1}^{N_2} k_i$	$M_3 = M_2 \times m_1$	$m_3 = MD(T, l+3)$
$\ldots$	$\ldots$	$\ldots$	$\ldots$
$l+h-1$	$N_{h-1} = \displaystyle\sum_{i=1}^{N_{h-2}} k_i$	$M_{h-1} = M_{h-2} \times m_{h-3}$	$m_{h-1} = MD(T, l+h-1)$
$l+h$	$N_h = r$	$M_h = M_{h-1} \times m_{h-2}$	$r = 0$, because we assume $H(T_a) = h$

The result of this work is published as a paper [53], which is available in chapter 8.

3.A   A DHB tree minimizes the cost function

In this appendix we prove the theorem 1 in subsection 3.2.1.

Firstly, we define the following functions:

- $H(T)$: returns the height of the tree T.

- $D(a)$: returns the depth of the node a in a tree.

- $S(T)$: returns the number of nodes in the tree T.

- $MD(T, l)$: returns the lowest out-degree of the nodes in T at depth l. In table 3.1, we represent $MD(T, l+i)$ as m_i.

Lemma 1. *In a DHB tree T, if there exist two subtrees T_a and T_b, such that depth of T_a's root is less than depth of T_b's root, then $S(T_a) \geq S(T_b)$.*

Proof. Assume a and b are the roots of subtrees T_a and T_b, such that $D(a) < D(b)$. First, let us assume a and b are placed at two consecutive depths, e.g., $D(a) = l$ and $D(b) = l + 1$. If $H(T) = t$, then by the height-balanced property of T, the

depth of its leaves are t and/or $t - 1$. We can measure the height of T_a and T_b as follows:

$$H(T_a) = \begin{cases} t - l & T_a\text{'s leaves are at depth t in } T \\ (t-1) - l & T_a\text{'s leaves are at depth t-1 in } T \end{cases}$$

$$H(T_b) = \begin{cases} t - (l+1) & T_b\text{'s leaves are at depth t in } T \\ (t-1) - (l+1) & T_b\text{'s leaves are at depth t-1 in } T \end{cases}$$

The minimum difference between the height of T_a and T_b is when the leaves of T_a are at depth $t - 1$ in T, and leaves of T_b are at depth t in T. In this situation both have the same height $h = t - l - 1$. In the rest of the proof we assume that $H(T_a) = H(T_b) = h$.

Table 3.1 shows the number of nodes in T_a and T_b at different depths. N_i shows the number of nodes in T_a at depth $l + i$ and M_i shows the maximum number of nodes in T_b at depth $l + i$. Using table 3.1, the number of nodes for each subtree is calculated by summing up the values in its corresponding column:

$$S(T_a) = N_0 + N_1 + N_2 + \cdots + N_{h-1} + N_h$$

$$S(T_b) = M_0 + M_1 + M_2 + \cdots + M_{h-1} + M_h$$

We know that $M_0 = 0$ and $N_h = 0$. Following the degree-balanced property of T, T_a and T_b, we have:

$$M_i \leq N_{i-1}; \forall i \in \{1, ..., h\}$$

Thus, $S(T_b) \leq S(T_a)$.

If $D(b) - D(a) > 1$, then we can find a node c, which is a descendant of node a at depth $D(b) - 1$. We already proved that $S(T_b) \leq S(T_c)$, therefore $S(T_b) \leq S(T_a)$. □

Theorem 1. *If T is a DHB tree, then the cost function in equation 3.3 is minimum.*

Proof. Assume to the contrary that T is a DHB tree, but the cost function is not minimized. That is, there exists different assingment (implemented using a tree rebalancing operation) that can be used to reduce the total cost of T by equation 3.3.

The tree rebalancing operation we consider here involves swapping the position of two nodes (and their subtrees) in the tree. Assume we select two nodes a and b as the root of the two subtrees T_a and T_b, such that $D(a) < D(b)$ and $D(b) - D(a) = d$. In the light of the lemma 1, we have $S(T_b) \leq S(T_a)$. Our rebalancing operation swaps the positions of T_a and T_b. It moves a and its sub-tree nodes lower in the tree, increasing its depth (and the depth of nodes in its sub-tree) by d. By equation 3.2, we increase the cost of all mappings in T_a by $S(T_a)d$. The same rebalancing operation moves b and its sub-tree nodes higher in the tree by the same depth d, decreasing the mapping costs of the moved nodes by $S(T_b)d$. As $S(T_a)d \geq S(T_b)d$, and by equation 3.3, after swapping, the cost of all moved mappings is either higher

or the same, so it does not decrease the total cost for the tree, thus, contradicting our earlier assumption.

Now, assume $D(a) = D(b)$, so $d = D(a) - D(b) = 0$. Here, by swapping the positions of T_a and T_b , we do not move up or down those nodes and their subtrees $(S(T_a)d = S(T_b)d = 0)$, therefore the total cost for the tree is not decreased. Again, this contradicts our initial assumption.

The only other operation, besides swapping, for assigning different mappings to rebuild T is to reposition a node in T. Given the height-balanced property of T, the only available positions in T are located at T's leaves. Assume if $H(T) = t$ and b is the root of subtree T_b, such that $D(b) = d$, then cutting T_b and adding it at leaves of T increases the mapping costs of b and its descendants by $S(T_b)(t - d)$. Since $t \geq d$, after rebalancing, the cost of all moved mappings is either higher or the same as before. $\square$

Chapter 4

Conclusions

In this project, we focused on two topics: (i) designing and implementing a distributed market model to construct P2P streaming overlays, in form of three systems: gradienTv, Sepidar, and GLive, and (ii) presenting a gossip-based NAT-friendly peer sampling service, called Gozar.

4.1 Sepidar, gradienTv 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 gradienTv and 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. Sepidar differs from gradienTv in that it handles the free-riding problem.

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 gradienTv, 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 GLive and Sepidar performance in different free-rider settings, and examine the effectiveness of our mechanism for addressing the free-riding problem.

4.2   Gozar

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. In this work, we present Gozar, a gossip-based peer sampling service that: (i) provides uniform random samples in the presence of NATs, and (ii) enables direct connectivity to sampled nodes using a fully distributed NAT traversal service, where connection messages require only a single hop to connect to private nodes.

We show in simulation that Gozar preserves the randomness properties of a gossip-based peer sampling service. We show the robustness of Gozar when a large fraction of nodes reside behind NATs and also in catastrophic failure scenarios. For example, if 80% of nodes are behind NATs, and 80% of the nodes fail, more than 92% of the remaining nodes stay connected. In addition, we compare Gozar with existing NAT-friendly gossip-based peer sampling services, Nylon and ARRG. We show that Gozar is the only system that supports one-hop NAT traversal, and its overhead is roughly half of Nylon's.

4.3   Future work

In the current implementation of our systems, we consider the upload bandwidth of the 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. In addition, in our streaming systems, we did not address the problem of nodes colluding to receive the video stream. As future work, it would be interesting to solve the free-rider problem, where a group of nodes cooperate with each other to cheat and receive data without helping in distributing it.

As another direction of our future work, we will integrate our existing streaming applications with Gozar and evaluate their behaviour in the open Internet.

Part II

Research Papers

Chapter 5

gradienTv: Market-based P2P live media streaming on the Gradient overlay

Amir H. Payberah, Jim Dowling, Fatemeh Rahimian, and Seif Haridi

gradienTv: Market-based P2P live media streaming on the Gradient overlay

Amir H. Payberah[†‡], Jim Dowling[†], Fatemeh Rahimian[†‡], and Seif Haridi[†‡]

[†]Swedish Institute of Computer Science (SICS)
[‡]KTH - Royal Institute of Technology
{amir, jdowling, fatemeh, seif}@sics.se

Abstract

This paper presents gradienTv, a distributed, market-based approach to live streaming. In gradienTv, multiple streaming trees are constructed using a market-based approach, such that nodes with increasing upload bandwidth are located closer to the media source at the roots of the trees. Market-based approaches, however, exhibit slow convergence properties on random overlay networks, so to facilitate the timely discovery of neighbours with similar upload bandwidth capacities (thus, enabling faster convergence of streaming trees), we use the gossip-generated Gradient overlay network. In the Gradient overlay, nodes are ordered by a gradient of node upload capacities and the media source is the highest point in the gradient. We compare gradienTv with state-of-the-art NewCoolstreaming in simulation, and the results show significantly improved bandwidth utilization, playback latency, playback continuity, and reduction in the average number of hops from the media source to nodes.

5.1 Introduction

Live streaming using overlay networks is a challenging problem. It requires distributed algorithms that, in a heterogeneous network environment, improve system performance by maximizing the nodes' upload bandwidth utilization, and improve user viewing experience by minimizing the playback latency, and maximizing the playback continuity of the stream at nodes.

In this paper, we improve on the state-of-the-art NewCoolstreaming system [24] for these requirements by building multiple media streaming overlay trees, where each tree delivers a part of the stream. The trees are constructed using distributed algorithms such that a node's depth in each tree is inversely proportional to its relative available upload bandwidth. That is, nodes with relatively higher upload bandwidth end up closer to the media source(s), at the root of each tree. This reduces load on the source, maximizes the utilization of available upload bandwidth at nodes, and builds lower height trees (reducing the number of hops from nodes to the source). Although we only consider upload bandwidth for constructing the Gradient overlay in this paper, the model can easily be extended to include other important node characteristics such as node uptime, load and reputation.

Our system, called *gradienTv*, uses a market-based approach to construct multiple streaming overlay trees. Firstly, the media source splits the stream into a set of sub-streams, called *stripes*, and divides each stripe into a number of *blocks*. Sub-streams allow more nodes to contribute bandwidth and enable more robust systems through redundancy [12]. Nodes in the system compete to become *children* of nodes that are closer to the root (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. The parent then *pushes* to the child subsequent blocks in the stripe, as long as it remains the child's parent. Children can proactively switch parent when the market-modelled benefit of switching is greater than the cost of switching.

The challenge with implementing this market-based approach is to find the best possible matching between parents and children in a timely manner, while having as few parent switches as possible. In general, for a market-based system to work efficiently, information and prices need to be spread quickly between participants. Insufficient information at market participants results in inefficient markets. In a market implemented using an overlay network, where the nodes are market participants, the communication of information and prices between nodes is expensive. For example, finding the optimal parent for each node requires, in principle, flooding to communicate with all other nodes in the system. Flooding, however, is not scalable. Alternatively, an approach to find parents based on random walks or sampling from a random overlay produces slow convergence time for the market and results in excessive parent switching, as information only spreads slowly in the market. We present a fast, approximate solution to this problem based on the *Gradient overlay* [6]. The Gradient is a gossip-generated overlay network, built by sampling from a random overlay, where nodes organize into a gradient structure with the media source at the centre of the gradient and nodes with decreasing relative upload bandwidth found at increasing distance from the centre. A node's neighbours in the Gradient have similar, or slightly higher upload bandwidth. The Gradient, therefore, efficiently acts as a *market maker* that matches up nodes with similar upload bandwidths, enabling the market mechanisms to quickly construct stable streaming overlay trees. As nodes with low relative upload bandwidths are rarely matched with nodes with high relative upload bandwidths (as can be the case in a random overlay), there is significantly less parent-switching before streaming overlay trees converge.

We evaluate gradienTv by comparison with NewCoolstreaming, a successful and widely used media streaming solution. We show in simulation that our market-based approach ensures that the system's upload bandwidth can be near maximally utilized, the playback continuity at clients is improved compared to New-Coolstreaming, the height of the media streaming trees constructed is much lower than in NewCoolstreaming, and, as a consequence, playback latency is less than NewCoolstreaming.

5.2   Related work

There are two fundamental problems in building data delivery (media streaming) overlay networks: (i) what overlay topology is built for data dissemination, and (ii) how a node discovers other nodes supplying the stream.

Early data delivery overlays use a tree structure, where the media is pushed from the root to interior nodes to leave nodes. Examples of such systems include Climber [8], ZigZag [9] and NICE [10]. The short latency of data delivery is the main advantage of this approach [7]. Disadvantages, however, include the fragility of the tree structure upon the failure of nodes close to the root and the fact that all the traffic is only forwarded by the interior nodes. SplitStream [12] improved this model by using multiple trees, where the stream is split into sub-streams and each tree delivers one sub-stream. Orchard [13], ChunkySpread [14] and CoopNet [15] are some other solutions in this class.

An alternative to tree structured overlays is mesh structure, in which the nodes are connected in a mesh-network [7], and nodes request missing blocks of data explicitly. 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 [7]. SopCast [54], DONet/Coolstreaming [18], Chainsaw [19], BiToS [55] and PULSE [20] are examples of mesh-based systems.

Another class of systems combine tree and mesh structures to construct a data delivery overlay. Example systems include CliqueStream [22], mTreebone [23], NewCoolStreaming [24], Prime [25] and [26]. GradienTv belongs to this class, where the mesh is the Gradient overlay.

The second fundamental problem is how nodes discover the other nodes that supply the stream. CoopNet [15] uses a centralized coordinator, GnuStream [30] uses controlled flooding requests, SplitStream [12] and [26] use DHTs, while NewCoolstreaming [24], DONet/Coolstreaming [18] and PULSE [20] use a gossip-generated random overlay network to search for the nodes.

NewCoolstreaming [24] has the most similarities with gradienTv. Both systems have the same data dissemination model where a node subscribes to a sub-stream at a parent node, and the parent subsequently pushes the stream to the child. However, gradienTv's use of the Gradient overlay to discover nodes to supply the stream contrasts with NewCoolStreaming that samples nodes from a random overlay (referred to as the partner-list). A second major difference is that NewCoolStreaming only reactively changes a parent when a sub-stream is identified as being slow, whereas gradienTv proactively changes parents to improve system performance.

5.3   Gradient overlay

The Gradient overlay is a class of P2P overlays that arrange 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 [5, 6]. As can be seen in

Figure 7.4.2, the highest utility nodes (darkest colour) are found at the core of the Gradient, and nodes with decreasing utility values (lighter grays) are found at increasing distance from the centre.
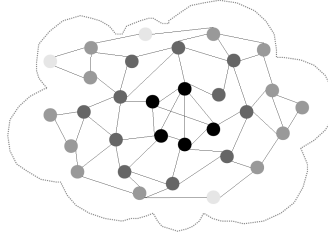

Figure 5.1: the Gradient overlay network

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 similar-view.

5.4   GradienTv system

In gradienTv, the media source splits the media into a number of *stripes* and divides each stripe into a sequence of *blocks*. GradienTv constructs a media streaming overlay tree for each stripe, where blocks are pushed from parents to children. Newly joined nodes discover stripe providers using the Gradient overlay and compete with each other to establish a parent-child relationship with providers. A node proactively changes its parent for a stripe, if it finds a lower depth parent for that stripe and if that parent either has a free *upload slot* or prefers this node to one of its existing children.

We use the term *download slot* to define a network connection at a node used to download a stripe. Likewise, an upload slot refers to a network connection at a node that is used to forward a stripe. If node p assigns its upload slot to node q's download slot, we say p is the *parent* of q and q is the *child* of p.

Our market model uses the following three properties, calculated at each node, to match nodes that can forward a stripe with nodes that want to download that stripe:

1. *Currency*: the total number of upload slots at a node, that is, the number of stripes a node is willing and able to forward simultaneously. A node uses its

currency when requesting to connect to another node's upload slot.

2. *Connection cost*: the minimum currency that should be provided for establishing a connection to receive a stripe. The connection cost to a node that has an unused upload slot is zero, otherwise the node's connection cost equals the lowest currency of its already connected children. For example, if node p has three upload slots and three children with currencies 2, 3 and 4, the connection cost of p is 2.

3. *Depth*: the shortest path (number of hops) from a node to the root for a particular stripe. Since the media stream consists of several stripes, nodes may have different depths in different trees. The lower the depth a node has for a stripe, the more desirable a parent it is for that stripe. Nodes constantly try to reduce their depth over all their stripes by competing with other nodes for connections to lower depth nodes.

5.4.1 Gradient overlay construction

Each node maintains two sets of neighbouring nodes: a random-view and a similar-view. Cyclon [40] is used to create and update the random-view and a modified version of the Gradient protocol is used to build and update the similar-view. The node references stored in each view contain the *utility value* for the nodes. The utility value of a node is calculated using two factors: a node's upload bandwidth and a disjoint set of discrete utility values that we call *market-levels*. A market-level is defined as a range of network upload bandwidths that have the same utility value. For example, in figure 5.2, we define some 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, Fibre (>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.

A node prefers to fill its similar-view with the 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 in a stream. Therefore, in order to enable low bandwidth nodes to utilize the spare 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). In Figure 5.2, 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.

Nodes bootstrap their similar-view using a bootstrap server, and, initially, the similar-view of a node is filled with random nodes that have equal or higher utility value. Algorithm 1 is executed periodically by the node p to maintain its similar-view. The algorithm describes how on every round, p increments the age of all the
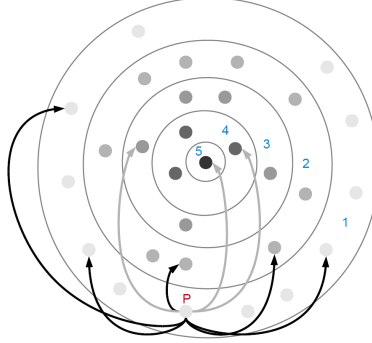
Figure 5.2: Different market-levels of a system, the similar-view of node p and its fingers

---

Algorithm 1 Updating the similar-view

1: **procedure UpdateSimilarView** $\langle this \rangle$
2:      $this.similarView.updateAge()$
3:      $q \leftarrow$ oldest node from $this.similarView$
4:      $this.similarView.remove(q)$
5:      $pView \leftarrow this.similarView.subset()$ ▷ a random subset from p's $similarView$
6:      **Send** $pView$ **to** q
7:      **Recv** $qView$ **from** q ▷ $qView$ is a subset of q's $similarView$
8:      **for all** $node_i$ **in** $qView$ **do**
9:          **if** $U_p(node_i) = U(p)$ **OR** $U_p(node_i) = U(p) + 1$ **then**
10:              **if** $this.similarView.contains(node_i)$ **then**
11:                  $this.similarView.updateAge(node_i)$
12:              **else if** $this.similarView$ has free entries **then**
13:                  $this.simialrView.add(node_i)$
14:              **else**
15:                  $node_j \leftarrow pView.poll()$ ▷ get and remove one entry from $pView$
16:                  $this.similarView.remove(node_j)$
17:                  $this.simialrView.add(node_i)$
18:              **end if**
19:          **end if**
20:      **end for**
21:      **for all** $node_a$ **in** $this.randomView$ **do**
22:          **if** $U_p(node_a) = U(p)$ **OR** $U_p(node_a) = U(p) + 1$ **then**
23:              **if** $this.similarView$ has free entries **then**
24:                  $this.simialrView.add(node_a)$
25:              **else**
26:                  $node_b \leftarrow (x \in this.similarView$ such that $U_p(x) > U(p) + 1)$
27:                  **if** $(node_b \neq null)$ **then**
28:                      $this.similarView.remove(node_b)$
29:                      $this.simialrView.add(node_a)$
30:                  **end if**
31:              **end if**
32:          **end if**
33:      **end for**
34: **end procedure**

Algorithm 2 Parent assignment

1: **procedure assignParent** $\langle\rangle$
2:     **for all** $stripe_i$ **in** $stripes$ **do**
3:         $candidates \leftarrow \text{findParent}(i)$
4:         **if** $candidates \neq null$ **then**
5:             $newParent \leftarrow$ a random node from $candidates$
6:             **send** $\langle\textsc{assignRequest} \mid i\rangle$ **to** $newParent$
7:         **end if**
8:     **end for**
9: **end procedure**

Algorithm 3 Select candidate parent from the similar-view and the fingers

1: **procedure findParent** $\langle i \rangle$
2:     $candidates \leftarrow \text{\textperthousand}$
3:     **if** $this.stripe_i.parnet = \text{null}$ **then**
4:         $this.stripe_i.parnet.depth \leftarrow \infty$
5:     **end if**
6:     **for all** $node_j$ **in** $(similarView \bigcup fingers)$ **do**
7:         **if** $node_j.stripe_i.depth < this.stripe_i.parent.depth$
8:         **AND** $node_j.connectionCost < this.currency$ **then**
9:             $candidates.\text{add}(node_j)$
10:         **end if**
11:     **end for**
12:     **return** $candidates$
13: **end procedure**

---

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 (lines 3-6). 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 (lines 8-20). What is more, 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 more than p's utility value plus one (lines 21-33).

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.

5.4.2 Streaming tree overlay construction

Algorithm 2 is called periodically by nodes to build and maintain a streaming overlay tree for each stripe. For each stripe i, a node p checks if it has a node in

its similar-view or finger list that has (i) a lower depth than its current parent, and (ii) a connection cost less than p's currency. If such a node is found, it is a added to a list of candidate parents for stripe i (Algorithm 3). Next, we use a random policy to select a node from the candidate parents, as it fairly balances connection requests over nodes in the system. In contrast, if we select the candidate parent with the minimum depth, then for even low variance in currency of nodes, it causes excessive connection requests to those nodes with high upload bandwidth.

Algorithm 4 Handling the assign request

```
 1: upon event ⟨AssignRequest | i⟩ from p
 2:     if has free uploadSlot then
 3:         assign an uploadSlot to p
 4:         send ⟨AssignAccepted | i⟩ to p
 5:     else
 6:         worstChild ← lowest currency child
 7:         if worstChild.currency ≥ p.currency then
 8:             send ⟨AssignNotAccepted | i⟩ to p
 9:         else
10:             assign an uploadSlot to p
11:             send ⟨Release | i⟩ to worstChild
12:             send ⟨AssignAccepted | i⟩ to p
13:         end if
14:     end if
15: end event
```

Algorithm 4 is called whenever a receiver node q receives a connection request from node p. If q has a free upload slot, it accepts the request, otherwise if p's currency is greater than the connection cost of q, q abandons one of its children with the lowest currency and accepts p as a new child. In this case, the abandoned node has to find a new parent. If q's connection cost is greater than p's currency, q declines the request.

5.5   Experiments and evaluation

In this section, we compare the performance of gradienTv with NewCoolstreaming under simulation. In summary, we define three different experiment scenarios: join-only, flash-crowds, and catastrophic failure, and, we show that gradienTv outperforms NewCoolstreaming in all of these scenarios for the following metrics: playback continuity, bandwidth utilization, playback latency, and path length. [1]

Experiment setup
We have implemented both gradienTV and NewCoolstreaming using the Kompics platform [56]. Kompics provides a framework for building P2P protocols, and simulation support using a discrete event simulator. Our implementation of New-Coolstreaming is based on the system description in [24, 57]. We have validated

---

[1]The source code and the results are available at: http://www.sics.se/∼amir/gradientv

our implementation of NewCoolstreaming by replicating, in simulation, the results from [24].

In our experimental setup, we set the streaming rate to 512 *Kbps* and unless stated otherwise, experiments involve 1000 nodes. The stream is split into 4 stripes and each stripe is divided into a sequence of 128 *KB* blocks. The media source is a single node with 40 upload slots. Nodes start playing the media after buffering it for 30 seconds. This is comparable with the most widely deployed P2P live streaming system, SopCast's that has average startup time of 30-45 seconds [54]. The size of a node's partial view (the similar-view in gradienTv, the partner list in NewCoolstreaming) is 15 nodes.

The number of upload slots for the non-root nodes is picked randomly from 1 to 10, which corresponds to upload bandwidths from 128 *Kbps* to 1.25 *Mbps*. As the average upload bandwidth of 704 *Kbps* is not much higher than the streaming rate of 512 *Kbps*, nodes have to find good matches as parents in order for good streaming performance. We assume all the nodes have enough download bandwidth to receive all the stripes simultaneously. In gradienTv, we define 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 one upload slot (128 *Kbps*) are the members of the first market-level, nodes with two upload slots (256 *Kbps*) are located in the second market-level, and the media source with 40 upload slots (>5 *Mbps*) is the only member of the 11th market-level.

Latencies between nodes are modelled using a latency map based on the King data-set [58]. In the experiments, we measure the following metrics:

1. *Playback continuity*: the percentage of blocks that a node received before their playback time. In our experiments to measure playback quality, we count the number of nodes that have a playback continuity of greater than 90%;

2. *Bandwidth utilization*: the ratio of the total number of utilized upload slots to the total number of requested download slots;

3. *Playback latency*: the difference in seconds between the playback point of a node and the playback point at the media source;

4. *Path length*: the minimum distance in number of hops between the media source and a node for a stripe.

We compare our system with NewCoolstreaming using the following scenarios:

1. *Join-only*: 1000 nodes join the system following a Poisson distribution with an average inter-arrival time of 100 milliseconds;

2. *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;

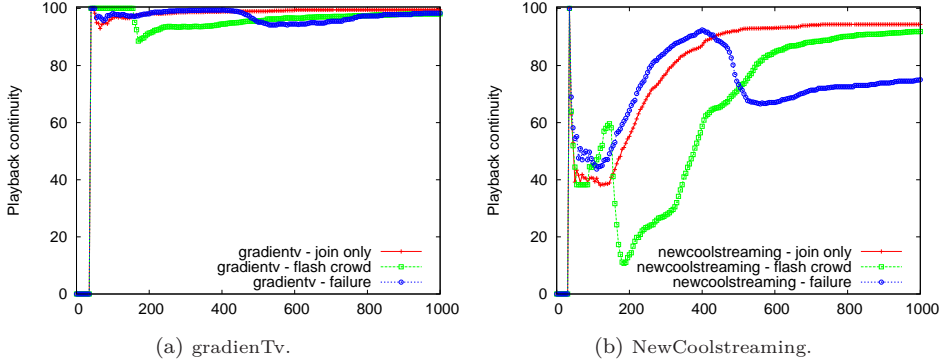(a) gradienTv.   (b) NewCoolstreaming.

Figure 5.3: Playback continuity in percent (Y-axis), against time in seconds (X-axis).

3. *Catastrophic failure*: as in the join-only scenario, 1000 nodes join the system
   following a Poisson distribution with an average inter-arrival time of 100 mil-
   liseconds. Then, 400 existing nodes fail following a Poisson distribution with
   an average inter-arrival time 10 milliseconds. The system then continues its
   operation with only 600 nodes.

In addition to these scenarios, we also evaluate the behaviour of gradienTv when
varying two key parameters: (i) the playback buffering time and (ii) the number of
nodes.

Playback Continuity
In this section, we compare the playback continuity of gradienTv and NewCool-
streaming in three different scenarios: join-only, flash crowd and catastrophic fail-
ure. In figures 5.3a and 5.3b, the X-axis shows the time in seconds, while the Y-axis
shows the percentage of the nodes in the overlay that have a playback continuity
more than 90%. We can see that gradienTv significantly outperforms NewCool-
streaming for the whole duration of the experiment in all scenarios. Moreover, af-
ter the system stabilizes, we observe a full playback continuity in gradienTv. This
out-performance is due to the faster convergence of the streaming overlay trees
in gradienTv, where high-capacity nodes can quickly discover and connect to the
source using the similar-view, while in NewCoolstreaming nodes take longer to find
parents as they search by updating their random view through gossiping. Another
reason for out-performance is the difference in policies used by a child to pull the
first block from a new parent. In gradienTv, whenever a node p selects a new parent
q, p informs q of the last block it has in its buffer, and q sends subsequent blocks to
p, while in NewCoolstreaming, the requested block is determined by looking at the
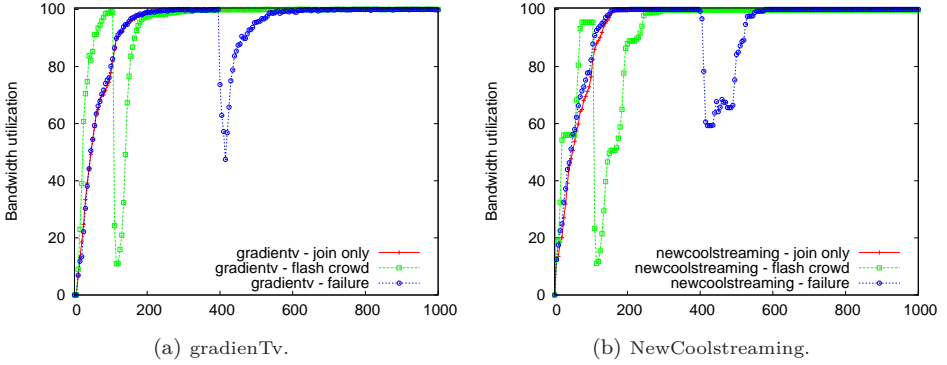head of the partners. This causes NewCoolstreaming to miss blocks when switching

(a) gradienTv.

(b) NewCoolstreaming.

Figure 5.4: Bandwidth utilization in percent (Y-axis), against time in seconds (X-axis).

parent.

Bandwidth Utilization
Our second experiment compares the bandwidth utilization of gradienTv (figure 5.4a) and NewCoolstreaming (figure 5.4b). We observe that when the system has no churn, as in the join-only scenario, both systems equally utilized the bandwidth. In the flash crowd and catastrophic failure scenarios, the performance of the both systems drops significantly. However, gradienTv recovers faster, as nodes are able to find parents more quickly using the Gradient overlay.

Path Length
In the third experiment, we compare the average path length of both streaming overlays. Before looking at the experiment results, we calculate the minimum depth of a k-ary tree with n nodes using $log_k(n)$. In our experiments, there are on average 5 upload slots per node (as upload slots are uniformly distributed from 1 to 10), and the minimum depth of the trees is expected to be $log_5(1000) \approx 4.29$. Figures 5.5a and 5.5b show tree depth of the system for gradienTv and NewCoolstreaming. We observe that gradienTv constructs trees with an average height of 4.3, which is very close to the minimum height. The figures also show that the depth of the trees in gradienTv are half the depth of the trees in NewCoolstreaming. Shorter trees enable lower playback latency.

What is more, we observe that the average depth of the trees is independent of the inter-arrival time of the joining nodes. This can be seen in figures 5.5a and 5.5b, where the depth of the trees, after the system stabilizes, is the same. More interestingly, in the catastrophic failure scenario, we can see a sharp drop in New-Coolstreaming tree depth, as a result of the drop in the number of nodes remaining

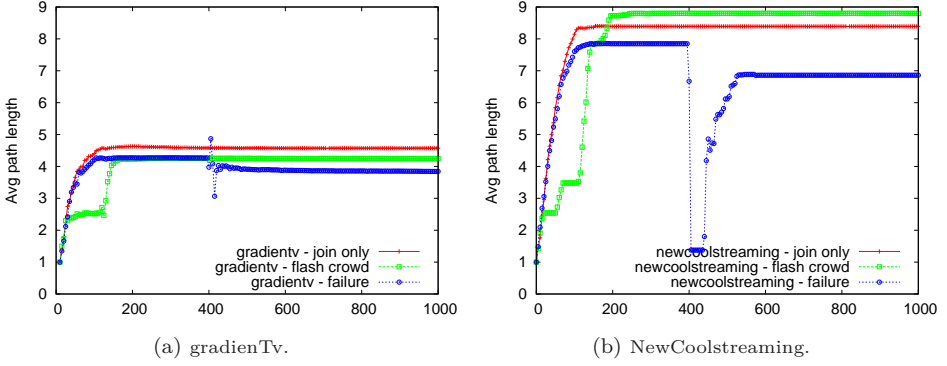(a) gradienTv.   (b) NewCoolstreaming.

Figure 5.5: Average path length in number of hops (Y-axis), against time in seconds (X-axis).

in the system and the fact that many remaining nodes do not have any path to the media source. The same behaviour is observed in gradienTv, but since the nodes can find appropriate nodes to connect to more quickly, the fluctuation in the average depth of trees is less than in NewCoolstreaming.

Playback Latency

This experiment shows how the average playback latency of nodes changes over time in our three scenarios (figures 5.6a and 5.6b). In the join-only scenario, we can see that 200 seconds after starting the simulation, the playback latency in gradienTv converges to just over 30 seconds, close to the initial buffering time, set at 30 seconds. For the join-only scenario, gradienTv exhibits lower average playback latency than NewCoolstreaming. This is because its streaming trees have lower depth, and, therefore, nodes receive blocks earlier than in NewCoolstreaming. This is also the case for the two other experiment scenarios, flash crowd and catastrophic failure. Here, we can see an increase in the average playback latency for both systems. This is due to the increased demand for parents by new nodes and nodes with failed parents. While the nodes are competing for parents, they may fail to receive the media blocks in time for playback. Therefore, they have to pause until a parent is found and the streaming is resumed. This results in higher playback latency. Nevertheless, when both systems stabilize, nodes will ignore the missing blocks and fast forward to the play from the block where the streaming from the new parent is resumed. Hence, the playback latency will improve after the system has settled down.

There is a significant difference between the behaviour of gradienTv and New-Coolstreaming upon an increase in the playback latency. In gradienTv, if playback latency exceeds the initial buffering time and enough blocks are available in the

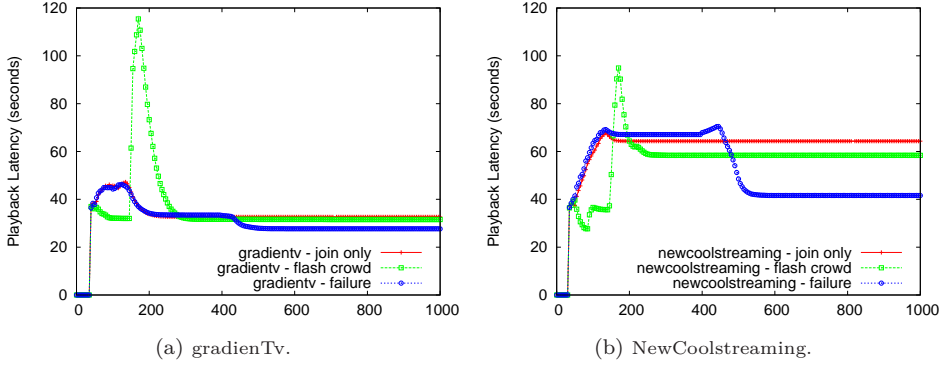(a) gradienTv.   (b) NewCoolstreaming.

Figure 5.6: Average playback latency in seconds (Y-axis), against time in seconds (X-axis).

buffer, nodes are given a choice to fast forward the stream and decrease the play-
back latency. In contrast, NewCoolstreaming jumps ahead in playback by switching
parent(s) causing it to miss blocks, thus it negatively affects playback continuity.

Buffering Time
We now evaluate the behaviour of gradienTv for different initial playback buffering
times. We compare four different settings: 0, 10, 20 and 30 seconds of initial
buffering time. Two metrics that are affected by changing the initial buffering
time are playback continuity and playback latency. Figure 5.7a shows that when
there is no initial buffering, the playback continuity drops to under 20% after 50
seconds of playback, but as the system stabilizes the playback continuity increases.
Buffering 10 seconds of blocks in advance results in less playback interruptions when
nodes change their parents, but better playback continuity is achieved for 20 and
30 seconds of buffering.

Figure 5.7b shows how playback latency increases when the buffering time is
increased. Thus, the initial buffering time is a parameter that trades off better
playback continuity against worse playback latency.

Number of Nodes
In this experiment, we evaluate the performance of the system for different system
sizes. We simulate systems with 128, 256, 512, 1024, 2048, and 4096 nodes, where
nodes join the system following a Poisson distribution with an average inter-arrival
time of 100 milliseconds. In figure 5.8a, we show the bandwidth utilization after all
the nodes have joined (for the different system sizes). We define d as the time when
all nodes have joined for a particular size. This means that for the system with
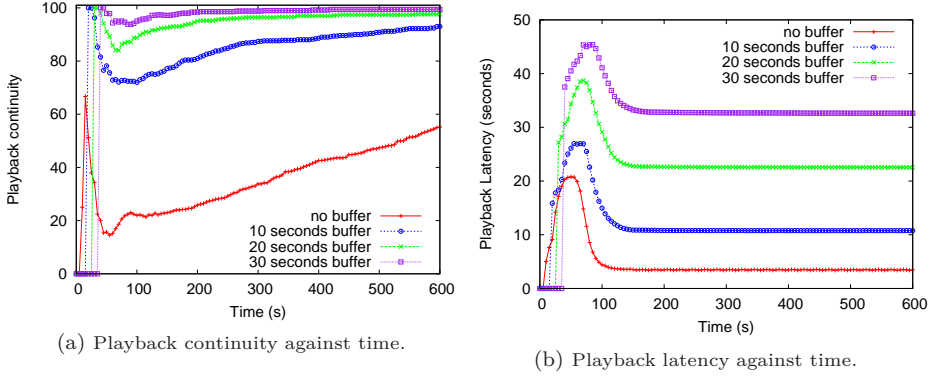
(a) Playback continuity against time.

(b) Playback latency against time.

Figure 5.7: The behaviour of gradienTv for different playback buffer lengths (in seconds).

(a) Bandwidth utilization against time.
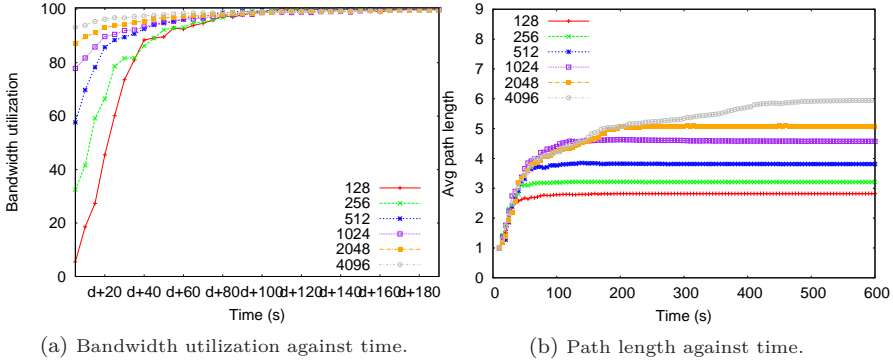

(b) Path length against time.

Figure 5.8: Bandwidth utilization and path length for varying numbers of nodes.

128 nodes, d is 13 seconds, while for the system with 4096 nodes d is 410 seconds. This experiment shows that, regardless of system size, nodes successfully utilize the upload slots at other nodes. This implies that convergence in terms of matching upload slots to download slots, appears to be independent of the number of nodes in the system. A necessary condition, of course, is that there is enough available upload and download bandwidth to deliver the stream to all nodes.

In the second experiment, we measure the tree depth while varying system sizes. We can see in figure 5.8b that the depth of the trees are very close to the theoretical minimum depth in each scenario. For example, the average depth of the trees with 1024 nodes is 4.34, which is very close to $log_5(1024) \approx 4.30$.

5.6 Conclusions

In this paper, we presented gradienTv, a P2P live streaming system that uses both the Gradient overlay and a market-based approach to build multiple streaming trees. The constructed streaming trees had the property that the higher a node's upload capacity, the closer that node is to the root of the tree. We showed how the Gradient overlay helped nodes efficiently find good neighbours for building these streaming trees. Our simulations showed that, compared to NewCoolstreaming, gradienTv has higher playback continuity, builds lower-depth streaming trees, has better bandwidth utilization performance, and lower playback latency.

Chapter 6

Sepidar: Incentivized Market-Based P2P Live-Streaming on the Gradient Overlay Network

Amir H. Payberah, Jim Dowling, Fatemeh Rahimian, and Seif Haridi

Sepidar: Incentivized Market-Based P2P Live-Streaming on the Gradient Overlay Network

Amir H. Payberah[†‡], Jim Dowling[†], Fatemeh Rahimian[†‡], and Seif Haridi[†‡]

[†]Swedish Institute of Computer Science (SICS)
[‡]KTH - Royal Institute of Technology
{amir, jdowling, fatemeh, seif}@sics.se

Abstract

Live streaming of video content using overlay networks has gained widespread adoption on the Internet. This paper presents Sepidar, a distributed market-based model, that builds and maintains overlay network trees, which are approximately minimal height, for delivering live media as a number of sub-streams. A streaming tree is constructed for each substream such that nodes that contribute higher amounts of upload bandwidth are located increasingly closer to the media source at the root of the tree. While our distributed market model can be run against a random sample of nodes, we improve its convergence time to stabilize a tree by executing against a sample of nodes that contribute similar amounts of upload bandwidth. We use the Gradient overlay network to generate samples of such nodes. We address the problem of free-riding through parent nodes auditing the behaviour of their child nodes. We evaluate Sepidar by comparing it in simulation with state-of-the-art New-Coolstreaming. Our results show significantly improved playback latency and playback continuity under churn, flash-crowd, and catastrophic failure experiment scenarios. We also show that using the Gradient improves convergence time of our distributed market model compared to a random overlay network. Finally, we show that Sepidar punishes the performance of free-riders, and that nodes are incentivized to contribute more upload bandwidth by relatively improved performance.

6.1 Introduction

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 incentivised to contribute resources through improved relative performance, and nodes that attempt to freeride, 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 paper, we meet these requirements by building 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 [59]. The media stream is split into a set of sub-streams, called *stripe*, and each tree delivers one sub-stream. Multiple sub-streams allow more nodes to contribute bandwidth and enable trees to be more robust [12].

Our system, called *Sepidar*, models the problem of constructing and maintaining minimal height overlay trees as an assignment problem [60], where the stripes that can be uploaded by nodes (*upload slots*) are matched to the stripes that nodes attempt to download (*download slots*), such that the height of the tree (the cost function for all nodes) is minimized. We introduce a new market model, a distributed algorithm inspired by auction algorithms [4], where, for each stripe, nodes continuously compete to become *children* of nodes providing stripes that are closer to the root (the media source). *Parents* supplying stripes prefer children nodes who offer to forward the highest number of copies of the stripes. Children proactively switch parents, when the market-modelled benefit of switching is greater than the cost of switching, until the trees stabilize. Our market model works in the presence of freeriders by parents periodically auditing children. Children are audited by querying the children's children (*grandchildren*) to validate that the child is forwarding the copies of stripes it claims to forward.

To improve the speed of convergence of the trees, nodes execute the market model in parallel using samples taken from the *Gradient overlay* [6]. The Gradient is a gossip-generated overlay network where nodes organize into a gradient structure with the media source at the centre of the gradient and nodes with decreasing relative upload bandwidth found at increasing distance from the centre. When nodes sample from their neighbours in the Gradient, they receive nodes with similar upload bandwidths. In a converged minimal height streaming overlay tree, the sampled nodes will be located at similar depths in the tree. Although we only consider upload bandwidth for constructing the Gradient and overlay trees in this paper, the model can easily be extended to include other characteristics such as node uptime, load and reputation.

We evaluate Sepidar by comparison with NewCoolstreaming, a successful and widely used media streaming solution [24]. We show in simulation, under churn, flash-crowd, and massive-failure scenarios, that our market-based approach improves the playback continuity and decreases the average playback latency at clients compared to NewCoolstreaming. We also evaluate the performance of Sepidar when varying key system parameters such as block size, number of stripes, playback buffering time, and freerider detection sensitivity. Finally, we evaluate the performance improvement for the market model in sampling from the Gradient overlay compared to sampling from a random overlay.

We build on our previous work in [1] by providing a distributed market model that works in the presence of freeriders and dynamic upload bandwidths.

6.2   Related work

There are two fundamental problems in building the media streaming overlay networks: (i) how to disseminate data, and (ii) how to discover other nodes supplying the stream.

Early data delivery overlays use a tree structure, where the media is pushed from the root to interior nodes to leave nodes. Examples of such systems include Climber [8], ZigZag [9], NICE [10], and [11]. The short latency of data delivery is the main advantage of this approach [7]. Disadvantages, however, include the fragility of the tree structure upon the failure of nodes close to the root and the fact that all the traffic is only forwarded by the interior nodes. SplitStream [12] improved this model by using multiple trees, where the stream is split into substreams and each tree delivers one sub-stream. Orchard [13], ChunkySpread [14] and CoopNet [15] are some other solutions in this class.

An alternative to tree structured overlays is the mesh structure, in which the nodes are connected in a mesh-network, and nodes request missing blocks of data explicitly. 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 [7]. SopCast [54], DONet/Coolstreaming [18], Chainsaw [19], and PULSE [20] are examples of mesh-based systems.

Another class of systems combine tree and mesh structures to construct a data delivery overlay. Example systems include CliqueStream [22], mTreebone [23], New-CoolStreaming [24], Prime [25] and [26].

The second fundamental problem is how nodes discover the other nodes that supply the stream. CoopNet [15] uses a centralized coordinator, GnuStream [30] uses controlled flooding requests, SplitStream [12] and [26] use DHTs, while NewCoolstreaming [24], DONet/Coolstreaming [18] and PULSE [20] use a gossip-generated random overlay network to search for the nodes. Sepidar uses the Gradient overlay for this purpose.

NewCoolstreaming [24] has the most similarities with Sepidar. Both systems have the same data dissemination model where a node subscribes to a sub-stream at a parent node, and the parent subsequently pushes the stream to the child. However, Sepidar's use of the Gradient overlay to discover nodes to supply the stream contrasts with NewCoolStreaming that samples nodes from a random overlay. A second major difference is that NewCoolStreaming only reactively changes a parent when a sub-stream is identified as being slow, whereas Sepidar proactively changes parents to improve system performance.

The problem of reducing freeriding in P2P systems has been addressed by many existing incentive mechanisms and reputation models [13, 59, 61]. Our solution for freerider identification is influenced by Give-to-Get [62], that first used transitive dependencies to a child's children in order to audit children nodes. In contrast to Sepidar, Give-to-Get is a video-on-demand protocol built on a mesh network, and based on BitTorrent.

Our market model is inspired by auction algorithms. The first widely-used auc-

tion algorithm was designed by Bertsekas [4], and has an equivalent representation as a weighted bipartite matching problem [60]. However, in contrast to auction algorithms, our market model does not assume that prices always rise - freeriders cause the price of an upload slot to be reset to zero. Also, our market model assumes local views of the system at nodes and that the discovery of nodes and price information is expensive.

Tan and Jarvis describe a payment-based approach to solving freeriding for live streaming [59]. Nodes run periodic auctions for their resources and earn points that can be used to access resources. Whereas we incentivize nodes to provide more resources to get better video performance, they incentivise nodes to remain in the system even when not viewing video to acquire an increased number of points. Similar to Sepidar, they also support a strategy for preferring the lowest depth parent resulting in the construction of a height-balanced tree. Another related approach to matching nodes for live streaming is based on finding maximal bipartite matchings using a flow algorithm by Li and Mahanti [63]. They transformed the traditional min-cost media flow dissemination problem into an auction problem.

6.3   Problem description

We assume the video is treated as a constant-rate bitstream that is divided into *blocks* of equal size without any coding, where every block has a sequence number to represent its playback order in the stream. The blocks are delivered to nodes over multiple sub-streams, called *stripes*, that each deliver an equal number of blocks per unit time. Nodes can retrieve any stripe independently from any other node that can supply the stripe. We define the number of copies of stripes that nodes are willing and able to forward as its number of *upload slots*. Nodes do not upload more stripes than they have upload slots. Each node has a number of upload slots, that is proportional to the amount of upload bandwidth capacity it contributes to the system. Every node has the same number of *download slots*, equal to the number of stripes. We assume all nodes have sufficient download bandwidth capacity to receive all stripes. A *parent* can forward a copy of any stripe over an upload slot, and a *child* node, that connects its download slot to an upload slot, requests a specific stripe for an upload slot. Nodes are not assumed to be cooperative; nodes may execute protocols that attempt to download the stream without forwarding it to other nodes. We do not, however, address the problem of nodes colluding to receive the video stream, although this can be addressed by a reputation management scheme [64].

The problem we address in this paper is how to deliver a video stream from a source as multiple stripes over multiple approximately minimal height trees. This problem can be represented as the *assignment problem* [50]. Centralized solutions to this problem are possible for small system sizes. For example, if all nodes send their number of upload slots to a central server, the server can use any number of algorithms that solve linear sum assignments, such as the auction algorithm [4], the

Hungarian method [51], or more recent high-performance parallel algorithms [50].

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 value. The 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 8.6. In the next section, we introduce our market model that finds approximate solutions using the partial views sampled from the Gradient overlay.

6.4 Sepidar system

Our distributed market model uses the following three properties, calculated at each node, to build trees:

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

2. *Price*: the minimum currency that should be bid when establishing 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 currency of its already connected children. For example, if node p has three upload slots and three children with currencies 2, 3 and 4, the price of p is 2.

3. *Cost*: the cost of an upload slot at a node for a particular stripe is the distance from that node to the root for that stripe. Since the media stream consists of several stripes, nodes may have different costs for different stripes. The lower the depth a node has for a stripe (the lower its cost), the more desirable a parent it is for that stripe.

Our market model is based on minimizing costs through nodes iteratively bidding for upload slots. This model could be best described as an approximate auction algorithm, where there is a *continuous auction* and *no reserve price*. For each stripe, child nodes place bids of their entire currency for upload slots at the parent nodes with lowest cost (depth). Child nodes always bid with their entire currency to avoid the complexity of price-setting. A parent node sets a price of zero for an upload slot when at least one of its upload slots is unassigned or when it has a free-riding child. Thus, the first bid for an upload slot will always win (no reserve price), 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 currency of its child with the lowest number of upload slots. If a child with more currency 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 currency with the new

child. A child that has lost an upload slot has to discover new nodes and bid for their upload slots. In contrast to the auction algorithm, there are no bidding and assignment phases, thus, we call it a continuous auction.

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 freerider, that is, if the node 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. Another 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 in the system with whom they can bid for upload slots. We use the Gradient overlay 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 us to limit exploration to the set of nodes with a similar number of upload slots.

6.4.1   Gradient overlay construction

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 [5, 6]. The highest utility nodes are found at the centre of the Gradient topology, while nodes with decreasing utility values are found at increasing distance from the centre.

The Gradient is built by both gossiping and sampling from a random overlay network (we use Cyclon [40]). 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 Sepidar, the utility value of a node is calculated using two factors: a node's upload bandwidth and 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 7.2, 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
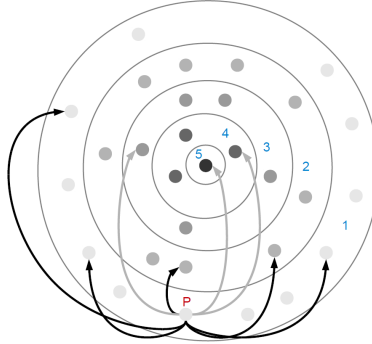
Figure 6.1: Different market-levels of a system, and the similar-view and fingers of *p*.

have enough upload bandwidth to simultaneously deliver all stripes in a stream. Therefore, in order to enable low bandwidth nodes to utilize the spare 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 7.2. 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. Each node *p* periodically increments the age of all the nodes in its similar-view, 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. What is more, *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.
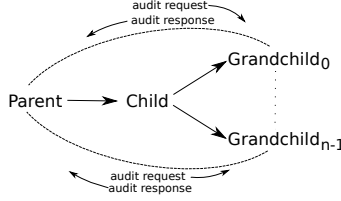
Figure 6.2: Transitive auditing by parents querying grandchildren about the performance of children.

6.4.2   Streaming tree overlay construction

Nodes periodically send their currency, cost, price, number of children and *buffer map* to their similar-view nodes. The buffer map shows the last blocks that a node has in its buffer. For each stripe i, a node p periodically checks if it has a node in its similar-view and finger list that has (i) a lower cost (depth) than its current parent, (ii) a price less than its currency and (iii) blocks ahead of its block in stripe i. If such a node is found, it is added to a list of candidate parents for stripe i. Next, the node sorts the candidates by the term $S = \frac{numOfChildren}{currency}$, and selects the node with smallest S. That is, it biases selection towards nodes with fewer children and higher currency. If two nodes have the same S, it selects the one with higher currency.

If a node q receives a connection request from node p for stripe i, has a free upload slot, it accepts the request, otherwise if p's currency is greater than the price of q, q abandons its child that has the lowest currency, and accepts p as a new child. If q has a freeriding child (see section 6.4.3), it abandons that node as the child with the lowest currency. The disconnected node has to find a new parent. If q's price is greater than or equal to p's currency, q declines the request.

6.4.3   Freerider detection and punishment

Freeriders are nodes that supply less upload bandwidth than claimed. To detect freeriders, we introduce the *freerider detector* component with *strong completeness* property. By strong completeness property, we mean that, if a *non-freerider* node does not have free upload slots, eventually it detects all its freeriding children.

Nodes identify freeriders through transitive auditing using their children's children (Figure 6.2). To do this, a non-freerider parent p periodically sends an *audit request*, about its child q, to q's claimed children. Whenever a grandchild receives a message from p, it checks if q 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 p that shows whether these conditions are satisfied or not.

We now show how strong completeness property is satisfied for the freerider detector. Assume a node q claims it has k upload slots, such that m of them

are assigned to other nodes and n of them are free upload slots, $k = m + n$. Its parent p periodically sends audit requests to q's m claimed children. Before the next iteration of sending audit requests, p calculates F as 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 F is more than $M\%$ of k, q is suspected as a freerider. If q becomes suspected in N consecutive iterations, it is detected as a freerider. For example, if N equals 2, a node is detected as a freerider if it is suspected on two consecutive iterations of the freerider detector. The higher the value of N, the more accurate but slower the detection is.

In a converged tree, for nodes not in the two bottom levels (market-levels one and two), we expect that at least $M\%$ of their upload slots are meeting their contracted obligation to correctly supply a substream over that upload slot. M is a threshold for freerider suspicion. For example, if M is 90%, then node q is suspected as a freerider, 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 freerider, the parent node p, decreases its own price (p's price) to zero and as a *punishment* considers the freerider node q as its child with the lowest currency. On the next bid from another node, p replaces the freerider node with the new node. So, if a node claims it has more upload bandwidth than it actually supplies, it will be detected and punished. In a converged tree, many members of the market-level one and two may have no children, because they are the leaves of the trees. So, the nodes in these two market-levels are not suspected as freeriders. Freeriders can use the extra resources in the system without any punishment if they just join as a member of market-level one or two.

6.5 Experiments and evaluation

In this section, we establish the performance of Sepidar for different system parameter settings, and then compare the performance of Sepidar with NewCoolstreaming under simulation.

6.5.1 Experiment setup

We have implemented both Sepidar and NewCoolstreaming using the Kompics platform [56]. Kompics provides a framework for building P2P protocols and a discrete event simulator for simulating them using different bandwidth, latency and churn models. Our implementation of NewCoolstreaming is based on the system description in [24, 57]. We have validated our implementation of NewCoolstreaming by replicating, in simulation, the results from [24].

In our experimental setup, we set the streaming rate to $512Kbps$. The stream is split into 8 stripes and each stripe is divided into a sequence of $16Kb$ blocks. Nodes start playing the media after buffering it for 15 seconds. The size of a node's partial view (the similar-view in Sepidar and the partner list in NewCoolstreaming) is 15 nodes. The number of upload slots for the non-root nodes equals $2i$, where i is

picked randomly from the range 1 to 10. Considering that the size of an upload slot
equals $64Kbps$, this number corresponds to an upload bandwidth between $128Kbps$
and $1.25Mbps$. As the average upload bandwidth of $704Kbps$ 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
80 upload slots. We assume all the nodes have enough download bandwidth to
receive all the stripes simultaneously. Here, we define 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 slot ($128Kbps$) 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 80 upload slots ($> 5Mbps$) is the only
member of the 11th market-level. Latencies between nodes are modelled using a
latency map based on the King data-set [58]. The failure detector settings are
$N = 2$ and $M = 50\%$.

   In the experiments, we measure the following metrics:

1. *Playback continuity*: the percentage of blocks that a node received before their
   playback time. In our experiments to measure playback quality, we count the
   number of nodes that have a playback continuity of greater than 90%;

2. *Playback latency*: the difference in seconds between the playback point of a
   node and the playback point at the media source.

We use the following scenarios in the experiments:

1. *Join only*: 1000 nodes join the system following a Poisson distribution with
   an average inter-arrival time of 100 milliseconds;

2. *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;

3. *Catastrophic failure*: 1000 nodes join the system following a Poisson distribu-
   tion 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. The system then continues its operation with only 500 nodes;

4. *Churn*: 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;

5. *Freerider*: 1000 nodes join the system following a Poisson distribution with
   an average inter-arrival time of 100 milliseconds, such that 20% of the nodes
   are freeriders.

(a) Different block sizes.

(b) Different numbers of stripes.
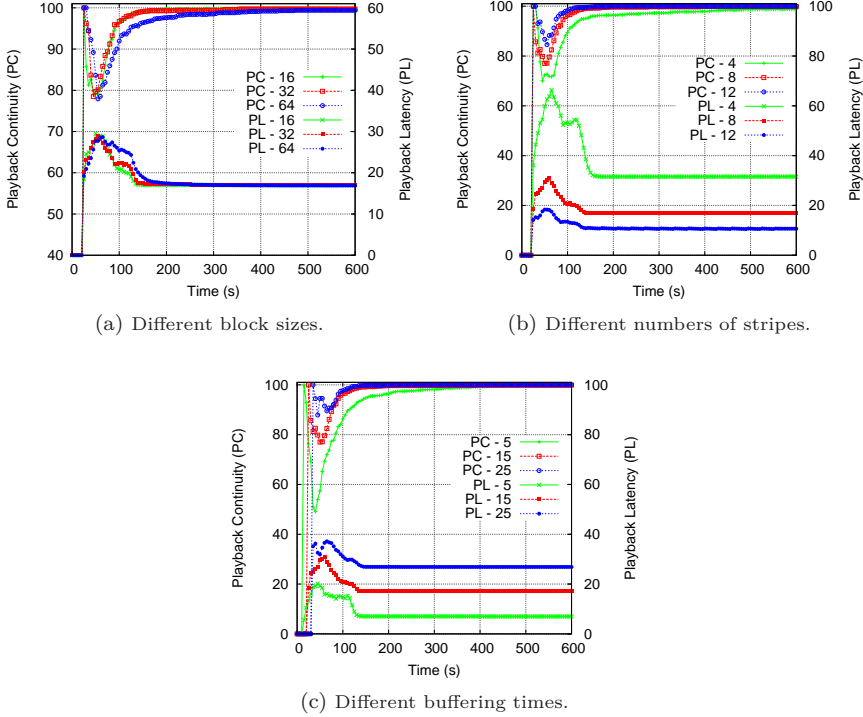
(c) Different buffering times.

Figure 6.3: System performance for different system settings.

6.5.2 Establishing parameters for good system performance

Here, we evaluate the performance of the system for different system settings. These experiments are based on the join only scenario. In the first experiment, we measure the performance of the system for varying block sizes: $16Kb$, $32Kb$ and $64Kb$. Figure 6.3a shows better playback continuity and less playback latency for smaller block sizes. The same result is shown in the NewCoolstreaming paper [24], and our subsequent experiments comparing Sepidar with Newcoolstreaming are based on a block size of $16Kb$.

Another system parameter is the number of stripes. As can be seen in figure 6.3b, as more stripes are used, playback continuity increases and playback latency reduces. For a media stream split into K stripes, a node that receives the whole stream should assign its K download slots to K upload slots. If a node misses M of its parent connections, its misses $\frac{M}{K}$ of the stream. So, as K increases, nodes lose less of the stream for a single failed parent connection.

Figure 6.3c shows the behaviour of Sepidar for different initial playback buffering

times. We compare three different settings: 5, 15 and 25 seconds of initial buffering time. Buffering 5 seconds of blocks in advance results in playback interruptions when nodes change their parents, but better playback continuity is achieved for 15 and 25 seconds of buffering. We can also see that playback latency increases when the buffering time is increased. Thus, the initial buffering time is a parameter that trades off better playback continuity against worse playback latency.

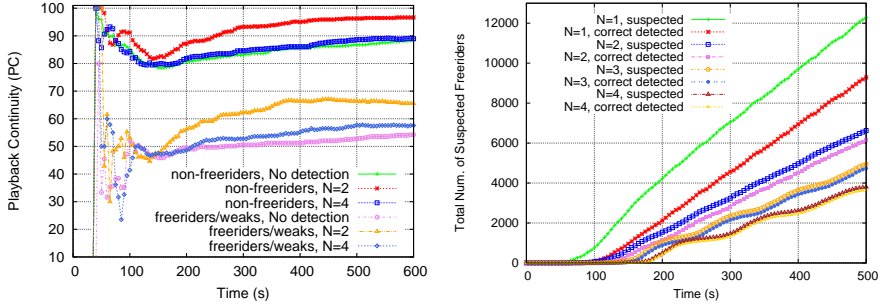### 6.5.3   Freerider detector settings

Here, we measure the playback continuity of nodes for different freerider detector settings. This experiment is based on the join only scenario. We consider the case where 30% of all nodes are freeriders and 20% are *weak* nodes, such that the ratio of the number of upload slots to download slots is less than one. Weak nodes are members of the market-level one or two, that is, nodes who only have enough upload bandwidth to forward at most half of the media stream. Nodes that are neither weak nor freerider nodes are called *non-freeriders*. Our experiments vary the freerider detector parameter N, while we measure the playback continuity of the different nodes.

Figure 6.4a shows the playback continuity of nodes for three values of N: $N = 0$, that is, no freerider detection, $N = 2$, and $N = 4$. We set M to 50% in all the simulations to take into account delayed replies by children and to decrease the false positive detection threshold for freeriders. We measured the playback continuity for other values of N, but to aid the readability of the plots we left them out. Although higher values of N increase the accuracy of the detector, the late detection of freerider decreases the playback continuity of nodes. Figure 6.4a confirms our conclusions as we see that the playback continuity of nodes when $N = 0$ and $N = 4$ are almost the same. The figure shows that $N = 2$ provides better playback continuity for the all nodes. Another important result here is the lower playback continuity of freeriders/weak nodes compared to non-freeriders. If a node detects one of its children as a freerider, it selects the freerider node as its child with the lowest currency, and replaces it with other nodes as soon as it receives a request. Losing a parent decreases the playback continuity of freeriders.

In figure 6.4b, we measure the total number of suspected nodes and the nodes that are correctly detected. As we see here, when N has lower values, the fraction of nodes that are correctly detected as freeriders decreases.

6.5.4   Sepidar vs. NewCoolstreaming

In this section, we compare the playback continuity and playback latency of Sepidar and NewCoolstreaming in the churn (figures 6.4c), catastrophic failure (figures 6.4d), flash crowd (figures 6.4e), and freerider (figures 6.4f) scenarios. In these figures, the Y1-axis (PC) shows the percentage of the nodes in the overlay that have
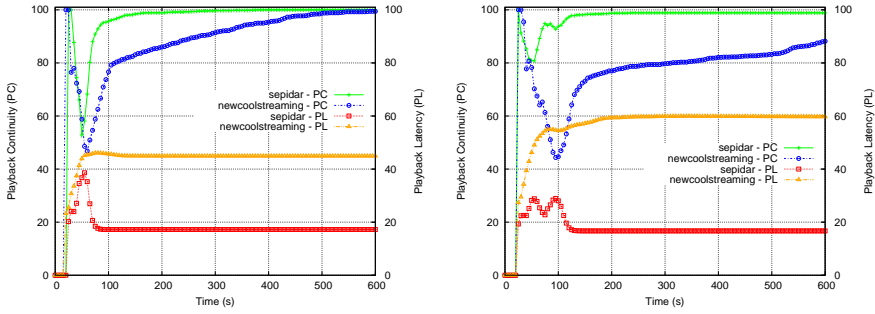
(a) Playback continuity of (non-)freerider nodes.

(b) Different freerider detector settings.
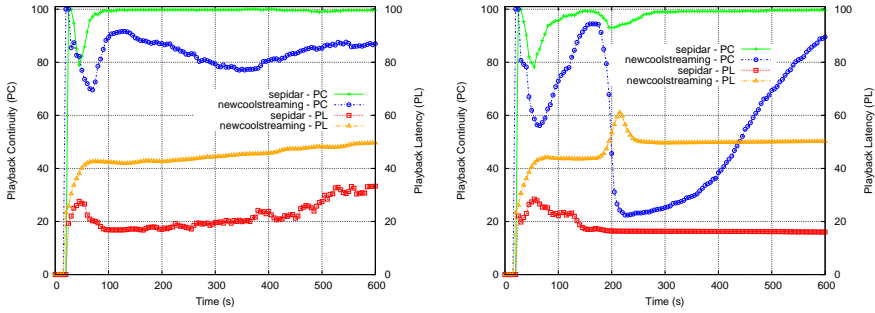
(c) Churn.

(d) Catastrophic failure.

(e) Flash crowd.

(f) Freeriders present.

Figure 6.4: Figures 6.4a and 6.4b show the behaviour of Sepidar in different settings of the freerider detector, and the other figures show the performance of Sepidar and Newcoolstreaming in different scenarios.

a playback continuity higher than 90%, and the Y2-axis (PL) shows the average playback latency.

We see that Sepidar significantly outperforms NewCoolstreaming in playback continuity for the whole duration of the experiment in all scenarios. This outper-formance is due to quicker discovery of appropriate parents and faster construction of overlay trees in Sepidar. In Sepidar, high capacity nodes can quickly discover and connect to the source using the similar-view, while in NewCoolstreaming nodes take longer to find parents as they search by updating their random view through gossiping. In addition, nodes in NewCoolstreaming do not consider the available upload bandwidth at the parent node when selecting a new parent, so nodes change their parents more often. This is the reason for the slow convergence of playback continuity in NewCoolstreaming. Another reason for outperformance is the dif-ference in policies used by a child to pull the first block from a new parent. In Sepidar, whenever a node p selects a new parent q, p informs q of the last block it has in its buffer, and q sends subsequent blocks to p, while in NewCoolstreaming, the requested block is determined by looking at the buffer head of the partners [24]. This causes NewCoolstreaming to miss blocks when switching parent.

As we see in all the scenarios, NewCoolstreaming keeps its playback latency constant, which is because of reactively changing parents when nodes playback latency is greater than the predefined threshold. There is a trade-off between playback continuity and playback latency in NewCoolstreaming, such that lower playback latency results in lower playback continuity [24]. In Sepidar the nodes have higher playback latency in the beginning, but they decrease it very soon when they finds appropriate parents, by ignoring the missed blocks and fast forwarding the stream to play from the block where streaming from the new parent is resumed.

An important point of difference between the two systems is the behaviour of Sepidar and NewCoolstreaming upon an increase in the playback latency. In Sepidar, if playback latency exceeds the initial buffering time and enough blocks are available in the buffer, nodes are given a choice to fast forward the stream and decrease the playback latency. In contrast, NewCoolstreaming jumps ahead in playback by switching parent(s) even it misses several blocks, thus negatively affecting playback continuity [24].

6.5.5   Incentivizing nodes to contribute upload bandwidth

Here, we investigate the level of incentives for nodes to contribute more upload bandwidth by measuring the performance of the top 10% of upload bandwidth nodes (*strong* nodes) and the bottom 10% of upload bandwidth nodes (*weak* nodes). We use the churn scenario explained in section 8.6.1. Since, weak nodes have lower upload bandwidth (and lower currency) compared to strong nodes, it takes longer for them to find an appropriate parent, and as a consequence their playback continuity decreases and their playback latency increases. Figure 6.5 compares the playback continuity and playback latency of strong nodes and weak nodes. As we can see, the strong nodes receive the stream with higher playback continuity and
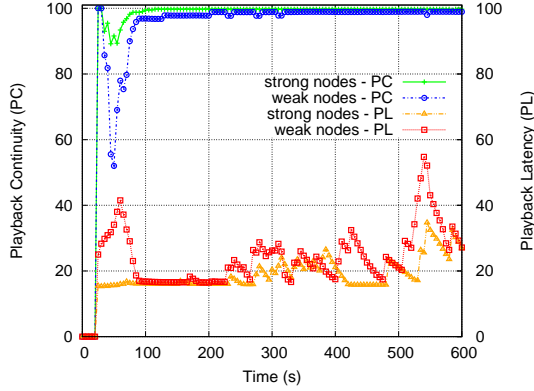
Figure 6.5: Playback continuity and playback latency of strong nodes vs. weak nodes.

lower playback latency compared to weak nodes. Moreover, while there is churn in the system, we see less fluctuation in the playback continuity of strong nodes. As such, nodes are strongly incentivized to contribute more upload bandwidth through receiving improved relative performance.

Figure 6.6: CDF of number of parent switches.
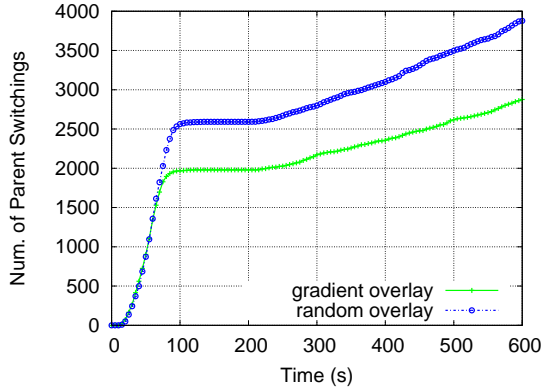
6.5.6 Comparing the Gradient with random neighbor selection

In the last experiment, we measure the convergence speed of our market model, in terms of number of parent switches in the Gradient overlay and a random network. Again, we compare them using the churn scenario. Our market model is run using (i) samples taken from the Gradient overlay, where the sampled nodes have similar

upload bandwidth or currency, and (ii) samples taken from a random network, where the sampled nodes have random amounts of currency. Since in the Gradient overlay, nodes receive bids from a set of nodes with almost the same currency, the difference between received bids is less than the expected difference for the random network. Figure 6.6 shows the CDF of number of parent switches for both overlays against time, and we can see that the Gradient overlay has a substantially lower number of parent switches.

6.6 Conclusions

In this paper, we presented Sepidar, a P2P live streaming system that uses both the Gradient overlay and a distributed market-based approach to build multiple minimal height trees, where nodes with higher available upload bandwidth are positioned higher in the tree. Sepidar addresses the problem of free-riding through parent nodes auditing the behaviour of their children nodes by querying their grandchildren. We showed how the Gradient overlay helped nodes efficiently find good neighbours for building these streaming trees. Our simulations showed that, compared to NewCoolstreaming, Sepidar has higher playback continuity and lower playback latency.

Chapter 7

GLive: The Gradient overlay as a market maker for mesh-based P2P live streaming

Amir H. Payberah, Jim Dowling, and Seif Haridi

GLive: The Gradient overlay as a market maker for mesh-based P2P live streaming

Amir H. Payberah[†‡], Jim Dowling[†], and Seif Haridi[†‡]

[†]Swedish Institute of Computer Science (SICS)
[‡]KTH - Royal Institute of Technology
{amir, jdowling, seif}@sics.se

Abstract

Peer-to-Peer (P2P) live video streaming over the Internet is becoming increasingly popular, but it is still plagued by problems of high playback latency and intermittent playback streams. This paper presents *GLive*, a distributed market-based solution that builds a mesh overlay for P2P live streaming. The mesh overlay is constructed 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 introduce a market-based approach that matches nodes willing and able to share the stream with one another. However, market-based approaches converge slowly on random overlay networks, and we improve the rate of convergence by adapting our market-based algorithm to exploit the clustering of nodes with similar upload bandwidths in our mesh overlay. We address the problem of free-riding through nodes preferentially uploading more of the stream to the best uploaders. We compare GLive with our previous tree-based streaming protocol, Sepidar, and NewCoolstreaming in simulation, and our results show significantly improved playback continuity and playback latency.

7.1 Introduction

Media streaming over Internet is becoming increasingly popular. Currently, most media is delivered using global content-delivery networks, providing a scalable and robust client-server model. For example, Youtube handle more than one billion hits per day[1]. However, content delivery infrastructures have very high cost, and an approach to reduce the cost of media delivery is to use peer-to-peer (P2P) overlay networks, where nodes share responsibility for delivering the media to one another.

Live media streaming using overlay networks is a challenging problem. Nodes should receive the stream with minimal delay over a best-effort network with varying bandwidth capacity, while adapting to other nodes joining, leaving and failing. From a system perspective, the overlay network should continuously optimize its structure to minimize the playback latency and maximize the timely delivery of the

---

[1]http://www.thetechherald.com/article.php/200942/4604/YouTube-s-daily-hit-rate-more-than-a-billion
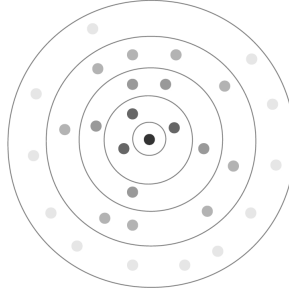
Figure 7.1: The mesh overlay with different layers of the nodes (for legibility, the links between nodes are not shown). The nodes in each layer have similar upload bandwidth. The darker the node is, the higher upload bandwidth it has. The media source is located at the center of the overlay.

stream, by adapting to system and network dynamics. Furthermore, nodes should be intentivized to contribute and share their resources, through improved relative performance.

In this paper, we present *GLive*, a P2P streaming overlay network that uses both the Gradient overlay network and a distributed market mechanism to adaptively optimize its topology to minimize playback latency and maximize the timely delivery of the stream. The Gradient overlay network constructs a topology where (i) nodes with higher available upload bandwidth are positioned closer to the media source, and (ii) nodes with similar upload bandwidth become neighbours, producing logical *layers* (see Figure 7.1). As nodes with relatively higher upload bandwidth can forward more copies of the stream to more nodes, positioning them closer to the media source will reduce the average number of hops from nodes to the media source, reducing both the probability of streaming disruptions and playback latency at nodes. Nodes are also incentivized to provide relatively more upload bandwidth, as nodes that contribute more upload bandwidth will have relatively higher playback continuity and lower latency than the nodes in lower layers.

In GLive, we divide the media stream into a sequence of *blocks*, and each node *pulls* the blocks of the stream from a set of nodes called *parents*. Nodes use a distributed market model, first introduced in [1], to choose parents from among the nodes in the system. A major problem with market-based approaches that select parents from random nodes is that they exhibit slow convergence properties. We improve the speed of convergence by nodes selecting from a small number of neighbouring nodes with similar upload bandwidth, i.e., a node either in its layer or in a layer closer to the media source (see Figure 7.1). The gossip-generated *Gradient overlay* network [5, 6] is used to enable nodes to sample neighbours with similar upload bandwidth, and, thus, it acts as a market-maker for our market model.

We evaluate GLive by comparing its performance with Sepidar [2] and the state-of-the-art NewCoolstreaming [24]. We show in simulation that GLive provides

better playback continuity and lower playback latency than these systems under churn, flash-crowd, and massive-failure scenarios. We also evaluate the performance of GLive and Sepidar when a high percentage of nodes are free-riders. Finally, we evaluate the convergence of our market model when the node samples are taken from the Gradient overlay compared to a random overlay.

Our work is an extension of our previous work on multiple-tree live streaming [1, 2], and the contributions of this paper include:

- GLive, a distributed market-based solution to create a mesh overlay for P2P live media streaming,

- how mesh-based streaming outperforms multiple-tree streaming through comparing GLive and Sepidar [2],

- how the Gradient overlay can improve the convergence time of a mesh overlay in comparison with a random network,

- a *scoring* model to solve the free-rider problem in mesh overlays.

7.2 Related work

Many different overlay network topologies have been used for data delivery in P2P media streaming systems, but the two most widely used approaches are multiple-tree [1, 2, 12, 13] and mesh-based overlays [17, 21, 24]. Multiple-tree overlay networks use push-based content delivery over multiple tree-shaped overlays with the media source as a root of all trees. While multiple-tree overlay networks have the advantage of low latency data delivery, they are vulnerable to the failure of interior nodes. Rajaee et al. have shown in [16] that mesh overlays have consistently better performance than tree-based approaches for scenarios where there is node churn and packet loss.

Mesh-based approaches use swarming content delivery over a typically random overlay network. In mesh-based overlays, unlike tree-based structures that data is pushed through the tree, nodes pull data from their neighbours in the mesh. 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 [7]. Gossip++ [17], NewCoolStreaming [24], Chainsaw [19], and PULSE [20] are the systems that use random overlay meshes for data dissemination. Recently, there has been work on using gossiping to build non-random mesh topologies, where the topology stores implicit information about node characteristics, such as upload bandwidth. In [21], Fortuna et al. attempts to organize nodes with decreasing upload bandwidth at increasing distance from the source. As such, these systems have similarities with how GLive uses the Gradient overlay to structure nodes. However, GLive also uses a market model to optimize its partners for media streaming.

The problem of reducing free-riding in P2P systems has been solved by many existing incentive mechanisms and reputation models [13, 59, 61]. Of particular

relevance to GLive are Give-to-Get [62] and Sepidar [2] that use transitive dependencies to a child's children in order to audit children nodes. In contrast, GLive uses a scoring mechanism to identify free-riders.

Our market model is an example of a distributed auction algorithm with partial information. Our model differs from existing work, such as [65] amd [66], in that all nodes are decision makers, the set of tasks and resources are homogeneous and auctions are restartable. Finally, our block selection strategy is similar to BiTOS for video-on-demand [55].

7.3 Problem description

We assume a network of nodes that communicate through message passing. New nodes may join the network at any time to watch the video. Existing nodes may leave the system either voluntarily or by crashing. The video is divided into a set of B blocks of equal size without any coding. Every block $b_i \in B$ has a sequence number to represent its playback order in the stream. Nodes can pull any block independently from any other node that can supply it.

Each node has a *partner list*, a view of a small subset of nodes in the system. A node can create a bounded number of *download connections* to partners and accept a bounded number of *upload connections* from partners over which blocks are downloaded and uploaded, respectively. We define a node q as the *parent* of the *child* node p, if an upload connection of q is bound to a download connection of p. Children nodes continuously attempt to improve their download connections by changing to parents that are both closer to the media source and able to deliver blocks on time. Parents, who can accept or reject connection attempts, prefer children who have forwarded the most blocks within a recent time window. The result of these preference functions is that nodes who forward more blocks on time have shorter paths to the media source.

Nodes store a list of blocks that are available for download in a *buffer map*. Nodes periodically send their buffer map to their children (via their upload connections) to advertise their available blocks. Children can then *pull* any blocks they require from the node. As such, advertisements are not random, but rather are directed away from the source and down the gradient.

For each block, we now represent the problem of finding the best mapping of upload connections to download connections as an *assignment problem* [50]. We define the set of all download and upload connections as D and U, respectively. In order to receive the block, a node requires one of its download connection needs to be assigned to an upload connection over which the block will be copied. We define an assignment or a *mapping* m_{ijk}, from a node i to a node j for block b_k, as a triplet containing one upload connection at i and one download connection at j for block b_k:

$$m_{ijk} = (u_i, d_j, b_k) : u \in U, d \in D, b \in B, i, j \in N, i \neq j \tag{7.1}$$

where N is the set of all nodes, b_k is block k from the set of all blocks B, and the connection from i to j is between two different nodes. A *cost function* is defined for a mapping m_{ijk} as the minimum distance from node i to the media source in terms of numbers of hops, that is,

$$c(m_{ijk}) : m_{ijk} \rightarrow number\, of\, hops\, from\, i\, to\, source. \tag{7.2}$$

We define a *complete assignment A* for a block b as a set of mappings, where, there exists at least one download connection at every node that is assigned to an upload connection over which b is downloaded. That is, for a block b, each node has a download connection over which it can pull the block before the block expires. The total cost of a complete assignment is calculated as follows:

$$c(A) = \sum_{m \in A} c\,(m) \tag{7.3}$$

The goal of our system is to minimize the cost function in equation 7.3 for every block $b \in B$, such that a shortest path tree is constructed over the set of available connections for every block.

If the set of nodes, connections, and the upload bandwidth of all nodes is static for all blocks B, then we can solve the same assignment problem $\mid B \mid$ times. However, P2P systems, typically have churn (nodes join and fail) and available bandwidth at nodes changes over time, so we have to solve a slightly different assignment problem every time a node join, exits or a node's bandwidth changes.

Centralized solutions, such as the auction algorithm [4], are possible in principle, where nodes bid to connect their download connections to better upload connections using the amount of blocks they forward as currency. However, nodes that offer upload connections may not deliver a block over a connection in time. As such, the problem can be viewed as a *restartable auction*, where the auction is restarted because a bidder did not have sufficient funds to complete the transaction. But, in general, it is not feasible to use centralized solutions in large and dynamic networks with real-time constraints. An alternative naive decentralized implementation of the auction algorithm that communicates will all nodes through flooding would not scale either. Approximate decentralized solutions, based on random walks or sampling from a random overlay, have slow convergence time, as we show in our evaluation.

In the next section, we introduce our market model that finds approximate solutions to the assignment problem using partial views sampled from the Gradient overlay (to improve convergence time compared to a random overlay). Nodes are not assumed to be cooperative; nodes may execute protocols that attempt to download the stream without forwarding it to other nodes. We do not, however, address the problem of nodes colluding to receive the video stream.

7.4   GLive system

We now present our distributed market-model, a modified version of the distributed auction algorithm with partial information introduced for tree-based live streaming in [2]. The following properties are used by the model and calculated locally at each node:

1. *Money*: the total number of blocks uploaded to children during the last 10 seconds. A node uses its money to bid for a binding to a partner's upload connection.

2. *Price*: the minimum amount of money that should be bid when binding to an upload connection. The price of a node that has an unbound upload connection is zero, otherwise the node's price equals the lowest amount of money at its existing children. For example, if node p has three upload connections and three children with monies 2, 3 and 4, the price of p is 2.

3. *Cost*: the cost of a node is the distance from that node to the media source via its shortest path. The shorter the path length (i.e., the lower its cost), the more desirable a parent it is.

Our market-model is based on minimizing costs (the path length of nodes to the media source) through nodes iteratively bidding for upload connections. Each node periodically sends its money, cost and price to all its partners. The partners of a node include all the nodes in its *similar-view* and *finger-list* in the Gradient overlay, see subsection 7.4.2. For each of its download connections, a child node p sends a bid request to nodes that: (i) have lower cost than one of the existing parents assigned to download connections in p, and (ii) the price of a connection is less than p's money. Nodes bid with their entire money (although the money is not used up, it can be reused for other bids for other connections).

A parent node who receives a bid request accepts it, if: (i) it has a free upload connection (its cost is zero), or (ii) it has assigned an upload connection to another node with a lower amount of money. If the 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. When a child node receives the acceptance message from another node, it assigns one of its download connections to the upload connection of the parent. Since a node may send more connection requests than its has download connections, it might receive more acceptance messages than it needs. In this case, if all its download connections are already assigned, it checks the cost of all its assigned parents and finds the one with the highest cost. If the cost of that parent is higher than the new received acceptance message, it releases the connection to that parent and accepts the new one, otherwise it ignores the received message.

Although there is no guarantee that the parent will forward all blocks over its connection to a child, parents who forward a relatively lower number of blocks will be removed as children of their parents. Nodes that claim that they have forwarded

more blocks than they actually have forwarded are removed as children, and, an auction is restarted for the removed child's connection. Nodes are incentivized to increase the upper bound on the number of their upload connections, as it will help increase their upload rate and, hence, their attractiveness as children for parents closer to the root.

7.4.1 Auction restarting - free-rider detection and punishment

Whenever a node assigns a download connection to the upload connection 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 childrens' children, or *grandchildren* for short.

Free-riders are nodes that forward a much lower number of blocks than they claimed they forward when connecting to a parent. We implment a *scoring* mechanism to detect free-riders, and thus motivate nodes to forward blocks. Each child assigns a score to each of its parents, which is initially set to zero, for a time window covering 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. Thus, 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 [67] and does not unneccessarily 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, and hence they have lower scores compared to others.

When a node with no free upload connection receives a connection request, it sorts its children based on their latest scores. If an existing child has a score less than a threshold *s*, then the 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 whose score is less than *s*, then the lowest score is selected. If all children have a score higher than *s*, then the parent accepts the connection if the connecting node has offers more money than the lowest money of its existing children. When the parent accepts such a connection, it then abandons (removes the connection to) the child with the lowest money. The abandonned child then has to search for and bid for a new connection to a new parent.

A crucial difference between our market-model and the classical auction algorithm is that our solution is decentralized; nodes have only a partial (changing) view of a small number of nodes in the system with whom they can bid for upload connections. We use the Gradient overlay to provide nodes with a constantly changing partial view of other nodes that have a similar number of upload connections. Thus, rather than have nodes explore the whole system for better parent nodes, the Gradient enables us to limit exploration to the set of nodes with a similar number

of upload connections.

7.4.2   Gradient overlay construction

Nodes search for parents by sampling partners from the Gradient overlay. The Gradient overlay is a gossip-generated overlay, where nodes are arranged according to their local utility function, such that the highest utility nodes are located topologically in the centre of the overlay, while lower utility nodes are located at increasing distance from the centre [5, 6].

Each node in the Gradient overlay maintains two sets of neighbours: *similar-view* and *random-view*. Similar-view is a partial list of the nodes in the system whose *utility values* are close to, but slightly higher than the utility value of the node. However, the nodes in the random-view are sampled from a random overlay network. We use Cyclon [40] to create and update the random-view. Nodes periodically gossip with each other and exchange their views. Upon receiving the views from a neighbour, a node merges it with its own similar-view and retains those entries that have closer (but higher) utility to its own utility value. The connections to the random nodes in random-view allow nodes to explore the network in order to discover other potentially similar neighbours.

In GLive, the utility value of a node is calculated using two factors: (i) a node's upload bandwidth, 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 7.2, 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 available 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. As a result, the nodes with similar utility value (almost the same upload bandwidth) become the neghibours of each other. In addition to similar-view and random-view, nodes maintain *finger-list* that contains at most one node from higher market levels (if one is available). Finger list reduces the probability of the overlay partitioning due to excessive clustering. Moreover, low bandwidth nodes often do not have enough upload bandwidth to simultaneously deliver all the stream. Therefore, in order to enable low bandwidth nodes to utilize the spare connections of higher bandwidth nodes, nodes can use the connections in finger-list (figure 7.2).

To update the similar-view, each node p periodically chooses one random node q from its similar-view, and sends it a random subset of the nodes from its similar-view. Upon receiving the list of nodes, q sends back a random subset of the nodes
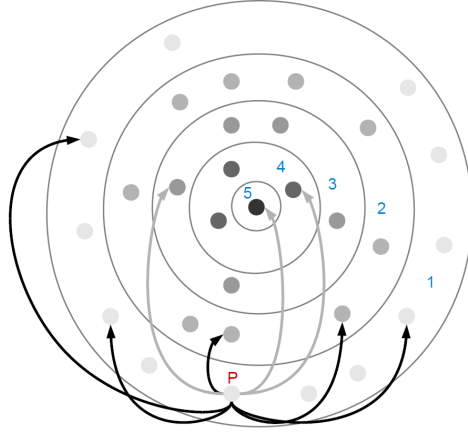
Figure 7.2: Different market-levels of a system, and the similar-view and fingers of p.

from its similar-view. When node p receives the q's view, first merges the received view with its existing similar-view by iterating through the received list of nodes, and preferentially selecting those nodes in the same market-level or at most one level higher. If its similar-view is not full, it adds the node, otherwise, it replaces one of the nodes it had sent to q with the selected node. Moreover, to allow nodes to find other potentially similar neighbours, p repeat the same procedure by merging its similar-view with its own local random-view.

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. For more details, you are kindly referred to our work in [1].

7.4.3 Data dissemination

Each parent node periodically sends its *buffer map* and its *load* to all its assigned children. The buffer map shows the blocks that a node has in its buffer, and 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. The main constraint in data dissemination in live media streaming is that the blocks should

(a) Churn.

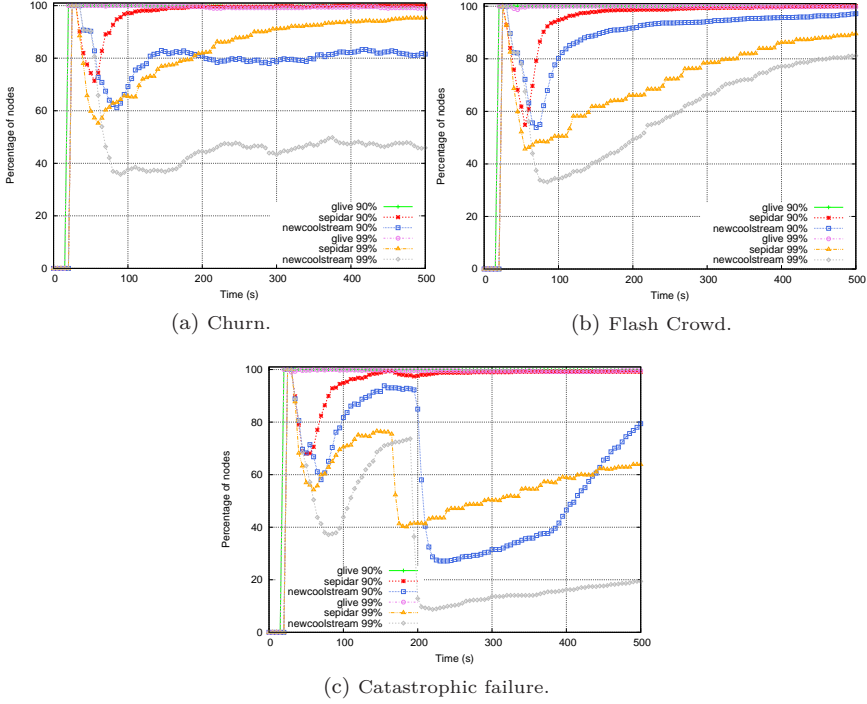(b) Flash Crowd.

(c) Catastrophic failure.

Figure 7.3: Playback continuity of the systems in different scenarios.

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, is to pull the rarest blocks in the system, as this is known to increase aggregrate network throughput [55].

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.

7.5 Experiments and evaluation

In this section, we compare the performance of GLive with P2P live-streaming systems Sepidar [2] and NewCoolstreaming [24] under simulation. Sepidar has a multiple-tree architecture and NewCoolstreaming has a random mesh-based architecture.

7.5.1 Experiment setup

We have used Kompics [56] to implement GLive, Sepidar 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 Sepidar and NewCoolstreaming based on the system descriptions from [2] and [57].

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 favourably to the 60 seconds of buffering used by state-of-the-art (proprietary) SopCast [54]. The size of similar-view in GLive and Sepidar and the partner list in NewCoolstreaming is 15 nodes. We assume all the nodes have the same number of download connections, which is set to 8. To model upload bandwidth, we assume that each upload connection has available bandwidth of $64Kbps$ and that the number of upload connections for nodes is set to $2i$, where i is picked randomly from the range 1 to 10. This means that nodes have upload bandwidth between $128Kbps$ and $1.25Mbps$. As the average upload bandwidth of $704Kbps$ 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 connections, providing five times the upload bandwidth of the stream rate. This setting is based on SopCast's requirement that the source has at least five times the upload capacity of the stream rate [54]. In our simulations we assume 11 market-levels, such that the nodes with the the same number of upload connections are located at the same market-level. For example, nodes with two upload connection ($128Kbps$) are the members of the first market-level, nodes with four upload connections ($256Kbps$) are located in the second market-level, and the media source with 40 upload connections ($2.5Mbps$) is the only member of the 11th market-level. Latencies between nodes are modeled using a latency map based on the King data-set [58].

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 the failure detector settings, we set the threshold of the score, s, to zero. The window used for our scoring mechanism is set to 10 seconds.

In the experiments, we measure the following metrics:

1. *Playback continuity*: 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%;

2. *Playback latency*: the difference in seconds between the playback point of a node and the playback point at the media source.

7.5.2   GLive vs. Sepidar vs. NewCoolstreaming

In this section, we compare the playback continuity and playback latency of GLive with Sepidar and NewCoolstreaming in the following scenarios:

1. *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;

2. *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;

3. *Churn*: 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;

Figures 7.3 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, at 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 connections is assigned. As we see in figure 7.3, both GLive and Sepidar outperform NewCoolstreaming in playback 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 7.3, 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,

(a) Churn.

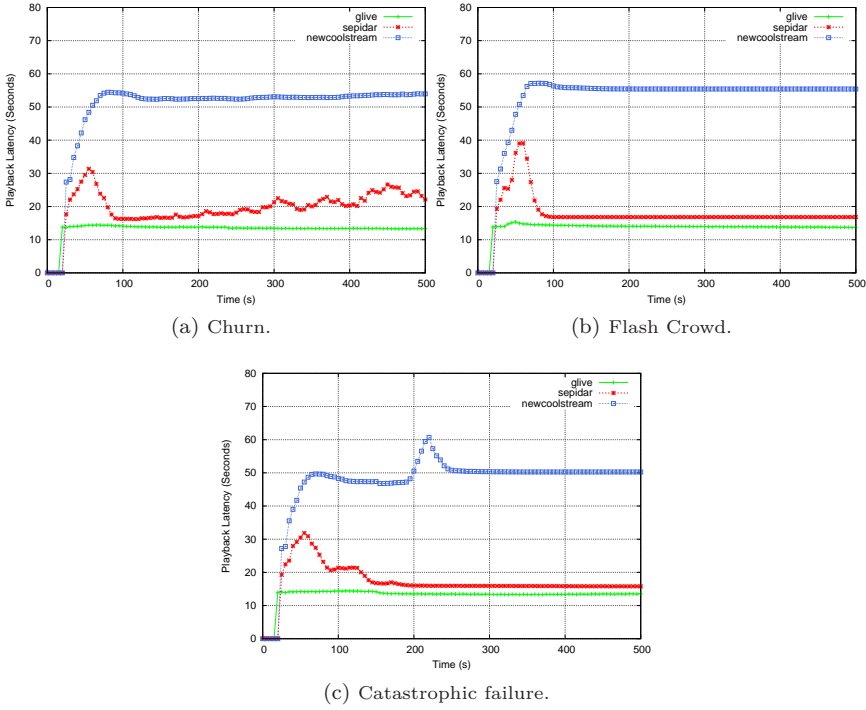(b) Flash Crowd.

(c) Catastrophic failure.

Figure 7.4: Playback latency of the systems in different scenarios.

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 stream is split into a number of sub-streams, and a node receives each sub-stream from a parent. Although, a node typically receives the blocks of each sub-stream independently, if the parent providing a sub-stream fails, then it loses the block from that sub-stream. While the node is trying to find a new parent for that sub-stream, it will miss the blocks for that sub-stream. 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 7.7 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 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,
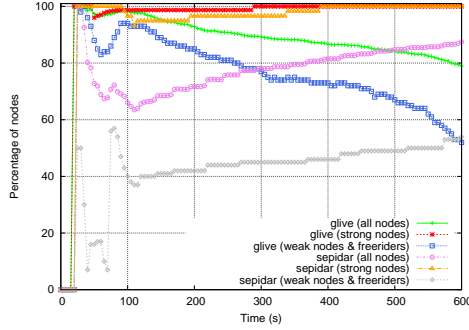
Figure 7.5: Playback continuity in the free-rider scenario.

the higher playback latency is a result of nodes only reactively changing parents when their playback latency is greater than a predefined threshold.

Another difference between GLive, Sepidar and NewCoolstreaming is the behavior of the systems when playback latency increases. In GLive and Sepidar, if playback latency exceeds the initial buffering time and enough blocks are available in the buffer, nodes are given a choice to fast forward the stream and decrease the playback latency. In contrast, NewCoolstreaming jumps ahead in playback by switching parent(s) even it misses several blocks, thus negatively affecting playback continuity [24].

7.5.3   Free-rider detector settings

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 amount of upload bandwidth in the system is less than total amount of download bandwidth required by nodes. Figure 7.5 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), the free-riders, and the *weak nodes* (the bottom 10% of upload bandwidth nodes).

Figure 7.5 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 [2]. In GLive, at 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
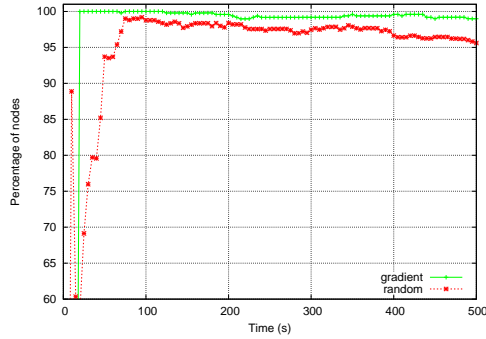
Figure 7.6: 99% of playback continuity of the GLive in the Gradient overlay and the random overlay.

the scores of their parents, and, thus, identify freeriders. 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 7.5 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 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 7.5, the existing free-riders in the system have a very low effect on the playback continuity of the strong nodes in 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.

7.5.4 Comparing the Gradient with random neighbour selection

In this experiment, we compare the convergence speed of our market model for the Gradient overlay and a random overlay. 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 upload bandwidth or money, and (ii) samples taken from a random network, where the sampled nodes have random amounts of money.

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 7.6 shows that 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.

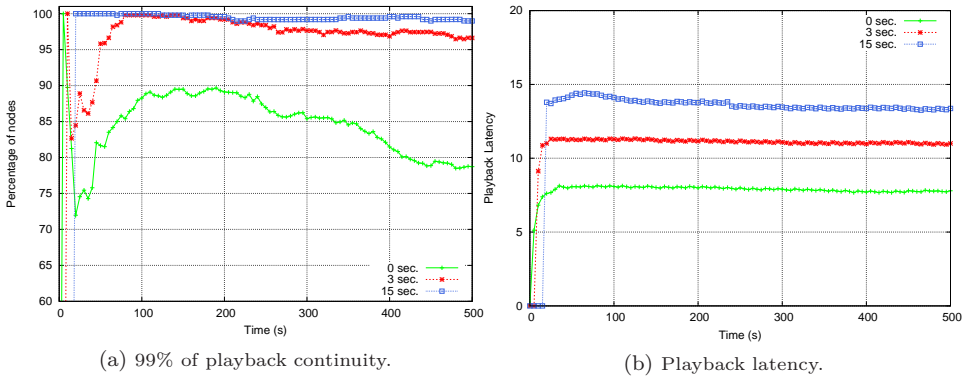(a) 99% of playback continuity.

(b) Playback latency.

Figure 7.7: The performance of the system in different initial buffering time.

7.5.5   Varying buffering time

Finally, we compare the performance of GLive for different buffering times. We compare three different settings: 0, 3 and 15 seconds of buffering time in the churn scenario. Buffering 0 seconds of blocks, means nodes start playing the media as soon as they receive the first block. As we see in figure 7.7a, the higher the buffering time, the higher the percentage of the nodes who receive blocks in time. However, higher initial buffering times increase the playback latency (figure 7.7b). As such, there is a trade-off between increasing playback continuity and decreasing playback latency.

7.6   Conclusions

In this paper, we presented GLive, a P2P live streaming system that uses a distributed market-model to construct a mesh overlay with two properties: (i) nodes with increasing upload bandwidth are located closer to the media source, and (ii) nodes with similar upload bandwidth are the neighbours of each other. Our distributed market-model leverages the structure of the Gradient overlay to efficiently assign suitable connections to other nodes. We address the problem of free-riding in GLive through parent nodes auditing the behaviour of their children nodes by

querying their grandchildren. We showed in simulation that the mesh-based implemention of our market-model has better performance in different scenarios compared to both a multiple-tree implementation of the system in Sepidar and New-Coolstreaming.

Chapter 8

Gozar: NAT-friendly Peer Sampling with One-Hop Distributed NAT Traversal

Amir H. Payberah, Jim Dowling, and Seif Haridi

Gozar: NAT-friendly Peer Sampling with One-Hop Distributed NAT Traversal

Amir H. Payberah[†‡], Jim Dowling[†], and Seif Haridi[†‡]

[†]Swedish Institute of Computer Science (SICS)
[‡]KTH - Royal Institute of Technology
{amir, jdowling, seif}@sics.se

Abstract

Gossip-based peer sampling protocols have been widely used as a building block for many large-scale distributed applications. However, Network Address Translation gateways (NATs) cause most existing gossiping protocols to break down, as nodes cannot establish direct connections to nodes behind NATs (private nodes). In addition, most of the existing NAT traversal algorithms for establishing connectivity to private nodes rely on third party servers running at a well-known, public IP addresses. In this paper, we present *Gozar*, a gossip-based peer sampling service that: (i) provides uniform random samples in the presence of NATs, and (ii) enables direct connectivity to sampled nodes using a fully distributed NAT traversal service, where connection messages require only a single hop to connect to private nodes. We show in simulation that Gozar preserves the randomness properties of a gossip-based peer sampling service. We show the robustness of Gozar when a large fraction of nodes reside behind NATs and also in catastrophic failure scenarios. For example, if 80% of nodes are behind NATs, and 80% of the nodes fail, more than 92% of the remaining nodes stay connected. In addition, we compare Gozar with existing NAT-friendly gossip-based peer sampling services, Nylon and ARRG. We show that Gozar is the only system that supports one-hop NAT traversal, and its overhead is roughly half of Nylon's.

8.1 Introduction

Peer sampling services have been widely used in large scale distributed applications, such as information dissemination [33], aggregation [34], and overlay topology management [5, 35]. A peer sampling service (PSS) periodically provides a node with a uniform random sample of live nodes from all nodes in the system, where the sample size is typically much smaller than the system size [68]. The sampled nodes are stored in a *partial view* that consists of a set of node descriptors, which are updated periodically by the PSS.

Gossiping algorithms are the most common approach to implementing a PSS [38–40]. Gossip-based PSS' can ensure that node descriptors are distributed uniformly at random over all partial views [41]. 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 [42].

The ability to establish direct connectivity with private nodes, using NAT traversal algorithms, has traditionally not been considered by gossip-based PSS'. 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. Nylon [42] was the first 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 (through its NAT) and coordinate NAT *hole punching* algorithms [42, 47]. As node descriptors spread in the system through gossiping, 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 (*churn*). Finally, hole punching is slow and can take up to a few seconds over the Internet [46].

This paper introduces *Gozar*, a gossip-based peer sampling service 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. Gozar uses public nodes as both *relay servers* [49] (to forward messages to private nodes) and *rendezvous servers* [47] (to establish direct connections with private nodes using hole punching algorithms).

Relaying and hole punching is enabled by private nodes finding public nodes who will act as both relay and rendezvous *partners* for them. For load balancing and fairness, public nodes accept only a small bounded number of private nodes as partners. When references to private nodes are gossiped in the PSS or sampled using the PSS, they include the addresses of their partner nodes. A node, then, can use these partners to either (i) gossip with a private node by relaying or (ii) establish a direct connection with the private node by using the partner for hole punching. We favour relaying over hole punching when gossiping with private nodes due to the low connection setup time compared to hole punching and also because the messages involved are small and introduce negligible overhead to public nodes. However, the hole punching service can be used by clients of the PSS to establish a direct connection with a sampled private node. NAT hole punching is typically required by applications such as video-on-demand [69] and live streaming [1, 2], where relaying would introduce too much overhead on public nodes.

A private node may have several redundant partners. Although redundancy introduces some extra overhead on public nodes, it also reduces latency when performing NAT traversal, as parallel connection requests can be sent to several partners, with the end-to-end connection latency being the fastest of the partners to complete NAT traversal. In this way, a more reliable NAT traversal service can be

built over more unreliable connection latencies, such as those widely seen on the Internet.

We evaluate Gozar in simulation and show how its PSS maintains its randomness property even in networks containing large fractions of NATs. We validate its behaviour through comparison with the widely used Cyclon protocol [40] (which does not support networks containing NATs). We also compare the performance of Gozar with the only other NAT-friendly PSS' we found in the literature, Nylon [42] and ARRG [70], and show how Gozar has less protocol overhead compared to Nylon and ARRG, and is the only NAT-friendly peer sampling system that supports one hop NAT traversal.

8.2 Related work

Dan Kegel explored STUN [43] as a UDP hole punching solution for NAT traversal, and Guha et al. extended it to TCP by introducing STUNT [71]. However, studies [47, 71] show that NAT hole punching fails 10-15% of the time for UDP and 30-40% of the time for TCP traffic. TURN [49] was an alternative solution for NAT traversal using relay nodes that works for all nodes that can establish an outbound connection. Interactive connectivity establishment (ICE) [72] has been introduced as a more general technique for NAT traversal for media streams that makes use of both STUN [43] and TURN [49]. All these techniques rely on third party servers running at well-known addresses.

Kermarrec et al. introduce in Nylon [42] a distributed NAT traversal technique that uses all existing nodes in the system (both private and public nodes) as rendezvous servers (RVPs). In Nylon, two nodes become the RVP of each other whenever they exchange their views. Later, if a node selects a private node for gossip exchange, it opens a direct connection to the private node using a chain of RVPs for hole punching. The chains of RVPs in Nylon are unbounded in length, making Nylon fragile in dynamic networks, and increasing traffic at intermediary nodes.

ARRG [70] supports gossip-based peer sampling in the presence of NATs without an explicit solution for traversing NATs. 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.

Renesse et. al [73] present an approach to fairly distribute relay traffic over public nodes in a NAT-friendly gossiping system. In their system, which is not a PSS, each node accepts exchange requests as much as it initiates view exchanges. Similar to Nylon, they use chains of nodes as relay servers.

In [74], D'Acunto et. al introduce an analytical model to show the impact of NATs on P2P swarming systems, and in [75] Liu and Pan analyse the performance of bittorrent-like systems in private networks. They show how the fraction of pri-

vate nodes affects the download speed and download time of a P2P file-sharing system. Moreover, authors in [76] and [46] study the characteristics of existing NAT devices on the Internet, and show the success rate, on the Internet, of NAT traversal algorithms for different NAT types. In addition, the distribution of NAT rule timeouts for NAT devices on the Internet is described in [76], and in [77] an algorithm is presented, based on binary search, to adapt the time required to refresh NAT rules to prevent timeouts.

8.3   Background

In gossip-based PSS', protocol execution at each node is divided into periodic cycles [41]. In each cycle, every node selects a node from its partial view to exchange a subset of its partial view with the selected node. Both nodes subsequently update their partial views using the received node descriptors. Implementations vary based on a number of different policies in node selection (rand, tail), view exchange (push, push-pull) and view selection (blind, heale, swapper) [41].

In a PSS, the sampled nodes should follow a uniform random distribution. To ensure randomness of a partial view in an overlay network, the overlay constructed by a peer sampling protocol should ensure that *indegree distribution*, *average shortest path* and *clustering coefficient*, are close to a random network [40, 41]. Kermarrec et al. evaluated the impact of NATs on traditional gossip-based PSS' in [42]. They showed that the network becomes partitioned when the number of private nodes exceeds a certain threshold. The larger the view size is, the higher the threshold for partitioning is. However, increasing the nodes' view size increases the number of stale node descriptors in views, which, in turn, biases the peer sampling.

There are two general techniques that are used to communicate with private nodes: (i) *hole punching* [47, 48] can be used to establish direct connections that traverse the private node's NAT, and (ii) *relaying* [49] 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 the two nodes and when slow connection setup times are not a problem. Relaying is preferable when the connection setup time should be short (typically less than one second) and small amounts of data will be sent over the connection.

In principle, existing PSS' could be adapted to work over NATs. This can be done by having all nodes run a protocol to identify their NAT type, such as STUN [43]. Then, nodes identified as private keep open a connection to a third party rendezvous server. When a node wishes to gossip with a private node, it can request a connection to the private node via the rendezvous server. The rendezvous server then executes a hole punching technique to establish a direct connection between the two nodes. Aside from the inherently centralized nature of this approach, other problems include the success rate of NAT hole punching for UDP is only 85-90% [47, 71], and the time taken to establish a direct connection using hole

punching protocols is high and has high variance (averaging between 700ms and 1100ms on the open Internet for the company Peerialism within Sweden [46]). This high and unpredictable NAT traversal time of hole punching is the main reason why Gozar uses relaying when gossiping.

8.4   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, then relaying is considered an acceptable solution. If it is important to distribute load as fairly as possible between public and private nodes, then option 3 is attractive. In existing systems, it appears that Skype supports both options 1 and 2, and can considered to have a solution to the fairness problem that, by virtue of its widespread adoption, can be considered acceptable to their user community [52].

8.5   The Gozar protocol

Gozar is a NAT-friendly gossip-based peer sampling protocol with support for distributed NAT traversal. Our implementation of Gozar is based on the *tail*, *push-pull* and *swapper* policies for node selection, view exchange and view selection, respectively [41] (although we also run experiments, ommitted here for brevity, showing that Gozar also works with different policies introduced in [41]).

In Gozar, node descriptors are augmented with the node's NAT type (private or public) and the mapping, assignment and filtering policies determined for the NAT [46]. A STUN-like protocol is run on a bootstrap server when a node joins the system to determine its NAT type and policies. We consider running STUN once at bootstrap time acceptable, as, although some corporate NAT devices can change their NAT policies dynamically, the vast majority of consumer NAT devices have a fixed NAT type and fixed policies.

In Gozar, each private node connects to one or more public nodes, called *partners*. Private nodes discover potential partners 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 partner 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 partners 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' partners also provide a rendezvous service to enable applications that sample nodes using the PSS to connect to them using a hole punching algorithm (if hole punching is possible).

8.5.1   Partnering

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 to determine its NAT type and policies. 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 partner 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 partner, it returns a NACK. The private node continues sending *partnering* requests to public nodes until it finds a partner, upon which the private

node can now start gossiping. Private nodes proactively keep their connections to their partners open by sending *ping* messages to them periodically. Authors in [76] 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 partners. If a private node detects a failed partner, it restarts the partner discovery process.

8.5.2 Peer sampling service

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 partners, which are initially empty. When a node descriptor is gossiped or sampled, other nodes learn about the node's NAT type and any partners. Later on, a node can gossip with a private node by relaying messages through the private node's partners.

Each node p periodically executes algorithm 5 to exchange and update its view. The algorithm shows that in each iteration, 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 partners). If the selected node q is a public node, then p sends the *shuffle request* message directly to q, otherwise it sends the *shuffle request* as a *relay message* to one of q's partners, selected uniformly at random.

Algorithm 5 Shuffle view.

1: **procedure ShuffleView** $\langle this \rangle$
2:     $this.view.updateAge()$
3:     $q \leftarrow SelectANodeToShuffleWith(this.view)$ ▷ See algorithm 6
4:     $this.view.remove(q)$
5:     $pView \leftarrow this.view.subset()$ ▷ a random subset from p's *view*
6:     $pView.add(p, p.natType, p.partners)$
7:     **if** $q.natType$ is public **then**
8:         **Send** $ShuffleRequest(pView, p)$ **to** q
9:     **else**
10:         $qPartner \leftarrow$ random partner from $q.partners$
11:         **Send** $Relay(shuffleRequest, pView, q)$ **to** $qPartner$
12:     **end if**
13: **end procedure**

Algorithm 6 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 partner.

Algorithm 7 is triggered whenever a node receives a shuffle request message. Once node q receives the shuffle request, 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 *shuffle response* back directly to it, otherwise it uses one of p's

Algorithm 6 Select a node to shuffle with.

1: **procedure SelectANodeToShuffleWith** $\langle this.view \rangle$
2:      **for all** $node_i$ **in** $this.view$ **do**
3:          **if** $node_i.natType =$ public **OR** ($node_i.natType =$ private **AND** $node_i.partners \neq \wp$)
    then
4:              $candidates \leftarrow node_i$
5:          **end if**
6:      **end for**
7:      $q \leftarrow$ oldest node from $candidates$
8:      **Return** q
9: **end procedure**

Algorithm 7 Handling the shuffle request.

1: **upon event** $\langle$Shuffle Request $\mid pView$, $p \rangle$ **from** m ▷ m can be p or $q.partner$
2:      $qView \leftarrow this.view.subset()$ ▷ a random subset from q's $view$
3:      **if** $p.natType$ is public **then**
4:          **Send** $ShuffleResponse(qView, q)$ **to** p
5:      **else**
6:          $pPartner \leftarrow$ random partner from $p.partners$
7:          **Send** $Relay(shuffleResponse, qView, p)$ **to** $pPartner$
8:      **end if**
9:      $UpdateView(qView, pView)$
10: **end event**

Algorithm 8 Handling the shuffle response.

1: **upon event** $\langle$Shuffle Response $\mid qView$, $q \rangle$ **from** n ▷ n can be q or $p.partner$
2:      $UpdateView(pView, qView)$
3: **end event**

Algorithm 9 Updating the view.

1: **procedure UpdateView** $\langle sentView, receivedView \rangle$
2:      **for all** $node_i$ **in** $receivedView$ **do**
3:          **if** $this.view.contains(node_i)$ **then**
4:              $this.view.updateAge(node_i)$
5:          **else if** $this.view$ has free entries **then**
6:              $this.view.add(node_i)$
7:          **else**
8:              $node_j \leftarrow sentView.poll()$
9:              $this.view.remove(node_j)$
10:             $this.view.add(node_i)$
11:         **end if**
12:     **end for**
13: **end procedure**

Algorithm 10 Handling the relay message.

```
1: upon event ⟨RELAY | natType, view, y⟩ from x
2:     if natType is shuffleRequest then
3:         Send ShuffleRequest(view, x) to y
4:     else
5:         Send ShuffleResponse(view, x) to y
6:     end if
7: end event
```

Algorithm 11 NAT Traversal to private nodes.

```
1: procedure SendData ⟨q, data⟩
2:     if q.natType is public then
3:         Send data to q
4:     else
5:         RVP ← random partner from q.partners
6:                              ▷ Determine hole punching algorithm for the combination of NAT types
7:         hp ← hpAlgorithm(p.natType, q.natType)
8:                              ▷ Start hole punching at RVP using the hole punching algorithm hp.
9:         holePunch(hp, p, q, RVP)
10:        Send data to q
11:    end if
12: end procedure
```

---

partners to relay the response. Again, node q selects p's relaying node uniformly at random from the list of p's partners. Finally, node q updates its view. A node updates its view whenever it receives a shuffle response (algorithm 8).

Algorithm 9 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 10 is triggered whenever a partner node receives a relay message from another node. The node extracts the embedded message that can be a shuffle request or shuffle response, 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 11 that implements the hole punching service. Algorithm 11 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 partners as a rendezvous node (RVP), and determines the hole punching algorithm (hp) using the combination of its own NAT type and q's NAT type RVP [46]. Then, p starts the hole punching process through the RVP [46]. After successfully establishing a direct connection, node p sends data directly to q.

8.6   Evaluation

In this section, we compare in simulation the behavior of Gozar with Nylon [42] and ARRG [70], the only two other NAT-friendly gossip-based PSS' we found in the literature. In our experiments, we use Cyclon as a baseline for comparison, where Cyclon experiments are executed using only public nodes. Cyclon has shown in simulation that it passes classical tests for randomness [40].

8.6.1   Experiment setup

We implemented Gozar, Cyclon, Nylon and ARRG on the Kompics platform [56]. 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, Nylon and ARRG are based on the system descriptions in [40], [42] and [70], respectively. Nylon differs from Gozar in its node selection and view merging policies: Gozar uses tail and swapper policies, while Nylon uses rand and healer policies [42]. For a cleaner comparison with the NAT-friendly features of Nylon, we use the tail and swapper policies in our implementation of Nylon.

In our experimental setup, for all four systems, the size of a node's partial view is 10, and the size of subset of the partial view sent in each view exchange is 5. The iteration period for view exchange is set to one second. Latencies between nodes are modelled on Internet latencies, using a latency map based on the King data-set [58]. In all simulations, 1000 nodes join the system following a Poisson distribution with an inter-arrival time of 10 milliseconds, and unless stated otherwise, 80% of nodes are behind NATs. In Gozar, each private node has 3 public nodes as partners, and they keep a connection to their partners open by sending ping messages every 50 seconds.

The experiment scenarios presented here are a comparison of the randomness of Gozar with Cyclon, Nylon and ARRG; a comparison of the protocol overhead of Gozar and Nylon for different percentages of private nodes, and finally, we evaluate the behaviour of Gozar in dynamic networks.

8.6.2   Randomness

Here, we compare the randomness of the PSS' of Gozar with Nylon and ARRG. Cyclon is used as a baseline for true randomness. In the first experiment, we measure the *local randomness* property [41] of these systems. Local randomness shows the number of times that each node in the system is returned by the PSS for each node in the system. For a truly random PSS, we expect that the returned nodes follow a uniform random distribution. In figure 8.1a, we measure the local randomness of all nodes in the system, after 250 cycles. For a uniform random distribution, the expected number of selections for each node is 25. As we can see, Cyclon has an almost uniform random distribution, while Nylon's distribution is slightly closer to uniform random than Gozar's distribution. ARRG, on the other

(a) Local randomness.

(b) Indegree distribution.

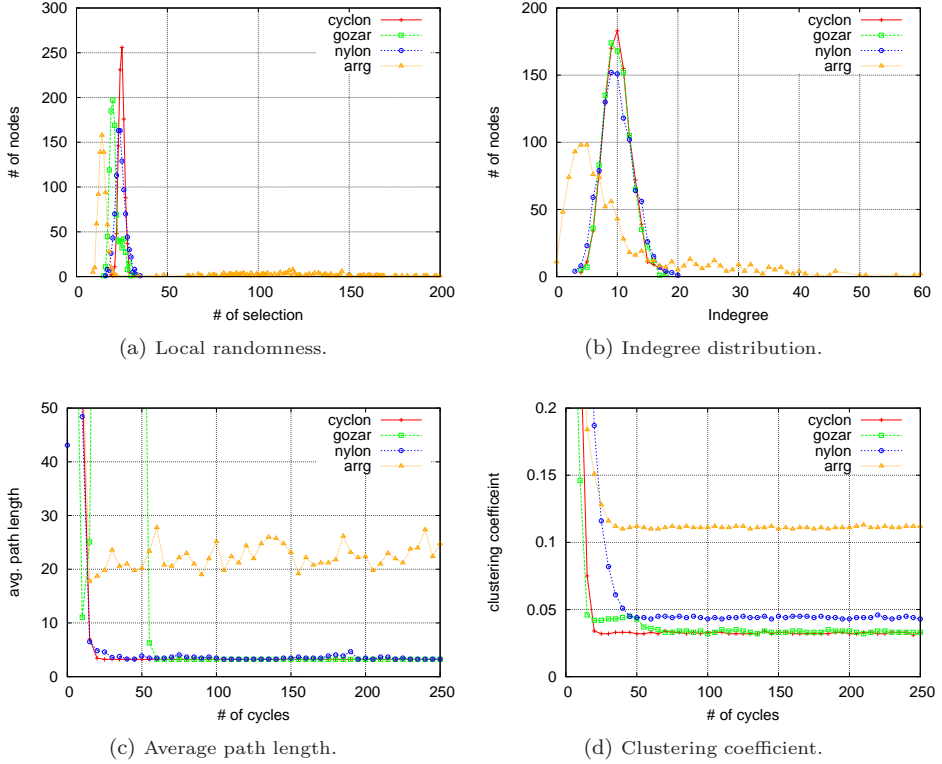(c) Average path length.

(d) Clustering coefficient.

Figure 8.1: Randomness properties.

hand, has a long-tailed distribution, where there are a few nodes that are sampled many times (the public nodes stored in private nodes' caches [70]). For Gozar, we can see two spikes: one representing the private nodes, that is roughly four times higher than the other consisting of the public nodes. This slight skew in the distribution results from the fact that public nodes are more likely to be selected during the first few cycles when private nodes have no partners.

In addition to the local randomness property, we use the *global randomness* metrics, defined in [41], to capture important global correlations of the system as a whole. The global randomness metrics are based on graph theoretical properties of the system, including the *indegree distribution*, *average path length* and *clustering coefficient*.

Figure 8.1b shows the indegree distribution of nodes after 250 cycles (the outdegree of all nodes is 10). In a uniformly random system, we expect that the indegree is distributed uniformly among all nodes. Cyclon shows this behaviour as
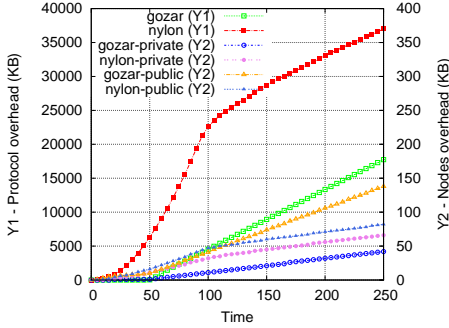
(a) Protocol overhead of Gozar vs. Nylon.
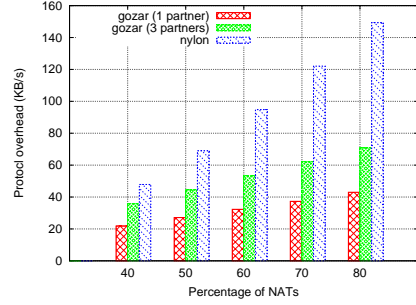
(b) Overhead traffic of Gozar vs. Nylon for varying percentages of private nodes.

Figure 8.2: Protocols overhead.

the node indegree is almost distributed uniformly among nodes. We can see the same distribution in Gozar and Nylon - their indegree distributions are very close to Cyclon. Again, due to high number of unsuccessful view exchanges in ARRG, we see that the node indegree is highly skewed.

In figure 8.1c, 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 8.1c also shows the average path length for the system in different cycles. Here, we can see the average path length of Gozar and Nylon track Cyclon very closely, but ARRG has higher average path length. As we can see, in the first few cycles, the path length of Gozar is high but after passing 50 cycles (50 seconds), the path length decreases. That is because of the time that private nodes need to find their partners and add them to their node descriptors.

Finally, we compare the clustering coefficient of the systems. The clustering coefficient of a node is the number of links between the neighbors of the node divided by all possible links. Figure 8.1d shows the evolution of the clustering coefficient of the constructed overlay by each system. We can see that Gozar and Nylon almost have the same clustering coefficient as Cyclon, while the value for ARRG is higher.

8.6.3   Protocol overhead

In this section, we compare the protocol overhead of Gozar and Nylon in different settings, where the protocol overhead traffic is the extra messages required to route messages through NATs. Protocol overhead traffic in Gozar consists of relay traffic and partner management, while in Nylon it consists of routing traffic. Figure 8.2a shows the protocol overhead when 80% of nodes are behind NAT. The Y1-axis

shows the total overhead, and the Y2-axis shows the average overhead of each public and private node. In this experiment, each private node in Gozar has three public nodes as partners, but only one partner is used to relay a message to a private node. Nylon, however, routes messages through more than two intermediate nodes on average (see [42] for comparable results). Figure 8.2a shows that after 250 cycles the relay traffic and partner management overhead in Gozar is $20000KB$, while the routing traffic overhead in Nylon is roughly $37000KB$.

Now, we compare the protocol overhead for Gozar and Nylon for different percentages of private nodes. To show the overhead in adding more partners, we consider two settings for Gozar: private nodes have one partner, and private nodes have three partners. In figure 8.2b, we can see that when 80% of nodes are behind NAT, the protocol overhead for all nodes in Nylon is around $150KBs$ after 250 cycles. The corresponding overhead in Gozar, when the private nodes have three and one partners, are around $70KBs$ and $40KBs$, respectively. The main contributory difference between the protocol overhead in the two different partner settings is that *shuffle request* and *shuffle response* messages become larger for more partners, as all partners addresses are included in private nodes' descriptors. The increase in traffic is a function of the percentage of private nodes (as only their descriptors include partner addresses), but is independent of the size of the partial view.

### 8.6.4 Fairness and connectivity after catastrophic failure

We evaluate the behaviour of Gozar if high numbers of nodes leave the system or crash. Our experiment models a catastrophic failure scenario: 20 cycles after 1000 nodes have joined, 50% of nodes fail following a Poisson distribution with inter-arrival time of 10 milliseconds.

Our first failure experiment shows the level of fairness between public and private nodes after the catastrophic failure. In figure 8.3a, the Y1-axis shows the average traffic on each public node and private node for different number of partners, and the Y2-axis shows the average number of unsuccessful view exchanges for each node. Here, 80% of nodes are private nodes and we capture the results 80 cycles after 50% of the nodes fail. As we can see in figure 8.3a, the higher the number of partners the private nodes have, the more overhead traffic generated, again, due to the increasing the size of messages exchanged among nodes. The Y2-axis shows that when the private nodes have only one partner, the average number of unsuccessful view exchanges is higher than when the private nodes have more than one partner. If a private node has more than one partner, then in case of failure of any of them, there are still other partners that can be used to communicate with the private node. An interesting observation here is that we cannot see a big decrease in the number of unsuccessful view exchanges when the private nodes has more than two partners. This observation, however, is dependent on our catastrophic failure model, and high churn rates might benefit more from more than two partners.

Finally, we measure the size of biggest cluster after a catastrophic failure. Here, we assume that each private node has three partners. Figure 8.3b shows the size
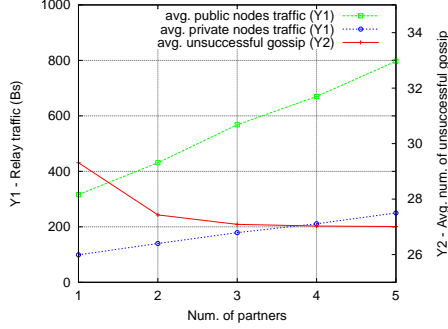
(a) Fairness after catastrophic failure: overhead for public and private nodes for varying numbers of parents.
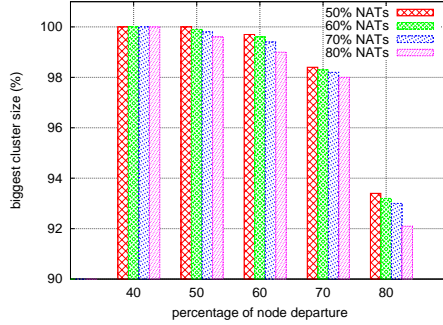
(b) Biggest cluster size after catastrophic failures.

Figure 8.3: Behaviour of the system after catastrophic failure.

of biggest cluster for varying percentages of private nodes, when varying numbers of nodes fail. We can see that Gozar is resilient to node failure. For example, in the case of 80% private nodes, when 80% of the nodes fail, the biggest cluster still covers more than 92% of the nodes.

8.7   Conclusion

In this paper, we presented *Gozar*, 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.

We showed in simulation that Gozar preserves the randomness properties of a gossip-based peer sampling service. We also show that the protocol overhead in our system is less than that of Nylon in different network settings and different percentages of private nodes. We also showed that the extra overhead incurred by public nodes is acceptable. Finally, we show that if 80% of the nodes are private, and when 50% of the nodes suddenly fail, more than 92% of nodes stay connected.

In future work, we will integrate our existing P2P applications with Gozar, such as our work on video streaming [1, 2], and evaluate their behaviour on the open Internet.

Bibliography

[1] A. H. Payberah, J. Dowling, F. Rahimian, and S. Haridi, "gradienTv: Market-based P2P Live Media Streaming on the Gradient Overlay," in *Lecture Notes in Computer Science (DAIS 2010)*, pp. 212–225, Springer Berlin / Heidelberg, Jan 2010.

[2] A. H. Payberah, J. Dowling, F. Rahimian, and S. Haridi, "Sepidar: Incentivized market-based p2p live-streaming on the gradient overlay network," *International Symposium on Multimedia*, vol. 0, pp. 1–8, 2010.

[3] A. H. Payberah, J. Dowling, and S. Haridi, "Glive: The gradient overlay as a market maker for mesh-based p2p live streaming," in *the 10th IEEE International Symposium on Parallel and Distributed Computing (ISPDC)*, July 2011.

[4] D. P. Bertsekas, "The auction algorithm: a distributed relaxation method for the assignment problem," *Ann. Oper. Res.*, vol. 14, no. 1-4, pp. 105–123, 1988.

[5] J. Sacha, B. Biskupski, D. Dahlem, R. Cunningham, R. Meier, J. Dowling, and M. Haahr, "Decentralising a service-oriented architecture," *Accepted for publication in Peer-to-Peer Networking and Applications*.

[6] J. Sacha, J. Dowling, R. Cunningham, and R. Meier, "Discovery of stable peers in a self-organising peer-to-peer gradient topology.," in *6th IFIP WG 6.1 International Conference Distributed Applications and Interoperable Systems (DAIS)* (F. Eliassen and A. Montresor, eds.), vol. 4025, (Bologna), pp. 70–83, June 2006.

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

[8] K. Park, S. Pack, and T. Kwon, "Climber: An incentive-based resilient peer-to-peer system for live streaming services," in *Workshop on Peer-to-Peer Systems (IPTPS)*, 2008.

[9] D. A. Tran, K. A. Hua, and T. T. Do, "Zigzag: An efficient peer-to-peer scheme for media streaming," in *INFOCOM*, 2003.

[10] S. Banerjee, B. Bhattacharjee, and C. Kommareddy, "Scalable application layer multicast," in *SIGCOMM '02: Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, (New York, NY, USA), pp. 205–217, ACM, 2002.

[11] S. Jarvis, G. Tan, D. Spooner, and G. Nudd, "Constructing Reliable and Efficient Overlays for P2P Live Media Streaming," in *21 st UK Performance Engineering Workshop*, p. 31, Citeseer, 2005.

[12] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh, "Splitstream: high-bandwidth multicast in cooperative environments," in *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, (New York, NY, USA), pp. 298–313, ACM Press, 2003.

[13] J. J. D. Mol, D. H. J. Epema, and H. J. Sips, "The orchard algorithm: P2p multicasting without free-riding," in *P2P '06: Proceedings of the Sixth IEEE International Conference on Peer-to-Peer Computing*, (Washington, DC, USA), pp. 275–282, IEEE Computer Society, 2006.

[14] V. Venkataraman, K. Yoshida, and P. Francis, "Chunkyspread: Heterogeneous unstructured tree-based peer-to-peer multicast," in *ICNP '06: Proceedings of the Proceedings of the 2006 IEEE International Conference on Network Protocols*, (Washington, DC, USA), pp. 2–11, IEEE Computer Society, 2006.

[15] V. N. Padmanabhan, H. J. Wang, P. A. Chou, and K. Sripanidkulchai, "Distributing streaming media content using cooperative networking," in *NOSSDAV '02: Proceedings of the 12th international workshop on Network and operating systems support for digital audio and video*, (New York, NY, USA), pp. 177–186, ACM, 2002.

[16] N. Magharei, R. Rejaie, and Y. Guo, "Mesh or multiple-tree: A comparative study of live p2p streaming approaches," in *INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE*, pp. 1424–1432, Ieee, 2007.

[17] D. Frey, R. Guerraoui, A. Kermarrec, and M. Monod, "Boosting Gossip for Live Streaming," in *Peer-to-Peer Computing (P2P), 2010 IEEE Tenth International Conference on*, pp. 1–10, IEEE, 2010.

[18] X. Zhang, J. Liu, B. Li, and T. shing Peter Yum, "Coolstreaming/donet: A data-driven overlay network for peer-to-peer live media streaming," in *IEEE Infocom*, 2005.

[19] V. Pai, K. Kumar, K. Tamilmani, V. Sambamurthy, A. E. Mohr, and E. E. Mohr, "Chainsaw: Eliminating trees from overlay multicast," in *Workshop on Peer-to-Peer Systems (IPTPS)*, pp. 127–140, 2005.

[20] F. Pianese, J. Keller, and E. W. Biersack, "Pulse, a flexible p2p live streaming system.," in *INFOCOM*, IEEE, 2006.

[21] R. Fortuna, E. Leonardi, M. Mellia, M. Meo, and S. Traverso, "QoE in Pull Based P2P-TV Systems: Overlay Topology Design Tradeoffs," in *Peer-to-Peer Computing (P2P), 2010 IEEE Tenth International Conference on*, pp. 1–10, IEEE, 2010.

[22] S. Asaduzzaman, Y. Qiao, and G. Bochmann, "CliqueStream: an efficient and fault-resilient live streaming network on a clustered peer-to-peer overlay," in *Proceedings of the 2008 Eighth International Conference on Peer-to-Peer Computing*, pp. 269–278, IEEE Computer Society, 2008.

[23] F. Wang, Y. Xiong, and J. Liu, "mtreebone: A hybrid tree/mesh overlay for application-layer live video multicast," in *ICDCS '07: Proceedings of the 27th International Conference on Distributed Computing Systems*, p. 49, 2007.

[24] B. Li, Y. Qu, Y. Keung, S. Xie, C. Lin, J. Liu, and X. Zhang, "Inside the new coolstreaming: Principles, measurements and performance implications," in *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*, pp. 1031–1039, 2008.

[25] N. Magharei and R. Rejaie, "Prime: Peer-to-peer receiver-driven mesh-based streaming," in *INFOCOM*, 2007.

[26] T. Locher, R. Meier, S. Schmid, and R. Wattenhofer, "Push-to-Pull Peer-to-Peer Live Streaming," in *21st International Symposium on Distributed Computing (DISC), Lemesos, Cyprus, Springer LNCS 4731*, September 2007.

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

[28] G. An, D. Gui-guang, D. Qiong-hai, and L. Chuang, "Bulktree: an overlay network architecture for live media streaming," 2006.

[29] "The Annotated Gnutella Protocol Specification v0.4". Accessed Jan-2008, Available: $http : //rfc − gnutella.sourceforge.net/developer/stable/index.html$.

[30] X. Jiang, Y. Dong, D. Xu, and B. Bhargava, "Gnustream: a p2p media streaming system prototype," in *ICME '03: Proceedings of the 2003 International Conference on Multimedia and Expo*, (Washington, DC, USA), pp. 325–328, IEEE Computer Society, 2003.

[31] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan, "Chord: A scalable Peer-To-Peer lookup service for internet applications," in *Proceedings of the 2001 ACM SIGCOMM Conference*, pp. 149–160, 2001.

[32] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," *Lecture Notes in Computer Science*, vol. 2218, pp. 329–??, 2001.

[33] P. T. Eugster, R. Guerraoui, S. B. Handurukande, P. Kouznetsov, and A.-M. Kermarrec, "Lightweight probabilistic broadcast," in *DSN '01: Proceedings of the 2001 International Conference on Dependable Systems and Networks (formerly: FTCS)*, (Washington, DC, USA), pp. 443–452, IEEE Computer Society, 2001.

[34] M. Jelasity, A. Montresor, and O. Babaoglu, "Gossip-based aggregation in large dynamic networks," *ACM Trans. Comput. Syst.*, vol. 23, no. 3, pp. 219–252, 2005.

[35] 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.

[36] R. Baldoni, M. Platania, L. Querzoni, and S. Scipioni, "Practical uniform peer sampling under churn," in *ISPDC '10: Proceedings of the 2010 Ninth International Symposium on Parallel and Distributed Computing*, (Washington, DC, USA), pp. 93–100, IEEE Computer Society, 2010.

[37] E. Bortnikov, M. Gurevich, I. Keidar, G. Kliot, and A. Shraer, "Brahms: Byzantine resilient random membership sampling," *Computer Networks*, vol. 53, pp. 2340–2359, March 2009.

[38] A. J. Ganesh, A.-M. Kermarrec, and L. Massoulie, "Peer-to-peer membership management for gossip-based protocols," *IEEE Transactions on Computers*, vol. 52, p. 2003, 2003.

[39] M. Jelasity and A. Montresor, "Epidemic-style proactive aggregation in large overlay networks," in *ICDCS '04: Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, (Washington, DC, USA), pp. 102–109, IEEE Computer Society, 2004.

[40] S. Voulgaris, D. Gavidia, and M. van Steen, "CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays," *Journal of Network and Systems Management*, vol. 13, no. 2, pp. 197–217, 2005.

[41] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. van Steen, "Gossip-based peer sampling," *ACM Trans. Comput. Syst.*, vol. 25, no. 3, p. 8, 2007.

[42] A.-M. Kermarrec, A. Pace, V. Quema, and V. Schiavoni, "Nat-resilient gossip peer sampling," in *ICDCS '09: Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems*, (Washington, DC, USA), pp. 360–367, IEEE Computer Society, 2009.

[43] J. Rosenberg, R. Mahy, P. Mathews, and D. Wing, "Rfc 5389: Session traversal utilities for nat (stun)," 2008.

[44] J. Rosenberg, J. Weinberger, C. Huitema, and R. Mahy, "Rfc 3489: Stun - simple traversal of user datagram protocol (udp) through network address translators (nats)," 2003.

[45] F. Audet and C. Jennings, "Network address translation (nat) behavioral requirements for unicast udp," in *RFC 4787 (best current practice)*, Internet Engineering Task Force, 2007.

[46] R. Roverso, S. El-Ansary, and S. Haridi, "Natcracker: Nat combinations matter," in *ICCCN '09: Proceedings of the 2009 Proceedings of 18th International Conference on Computer Communications and Networks*, (Washington, DC, USA), pp. 1–7, IEEE Computer Society, 2009.

[47] B. Ford, P. Srisuresh, and D. Kegel, "Peer-to-peer communication across network address translators," *CoRR*, vol. abs/cs/0603074, 2006.

[48] R. H. Huynh Cong Phuoc and A. McKenzie, "Nat traversal techniques in peer-to-peer networks," 2008.

[49] C. H. J. Rosenberg, R. Mahy, "Turn - traversal using relay nat," in *[Online]. Available: http://tools.ietf.org/id/draft-rosenberg-midcom-turn-08.txt, Sep. 2005.*

[50] C. N. Vasconcelos and B. Rosenhahn, "Bipartite graph matching computation on gpu," in *7th International Conference on Energy Minimization Methods in Computer Vision and Pattern Recognition*, (Berlin, Heidelberg), pp. 42–55, Springer-Verlag, 2009.

[51] H. W. Kuhn, "The Hungarian method for the assignment problem," *Naval Research Logistic Quarterly*, vol. 2, pp. 83–97, 1955.

[52] D. Bonfiglio, M. Mellia, M. Meo, D. Rossi, and P. Tofanelli, "Revealing skype traffic: when randomness plays with you," *SIGCOMM Comput. Commun. Rev.*, vol. 37, no. 4, pp. 37–48, 2007.

[53] A. H. Payberah, J. Dowling, and S. Haridi, "Gozar: Nat-friendly peer sampling with one-hop distributed nat traversal," in *the 11th IFIP international conference on Distributed Applications and Interoperable Systems (DAIS)*, June 2011.

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

[55] A. Vlavianos, M. Iliofotou, and M. Faloutsos, "Bitos: enhancing bittorrent for supporting streaming applications," in *In IEEE Global Internet*, pp. 1–6, 2006.

[56] C. Arad, J. Dowling, and S. Haridi, "Developing, simulating, and deploying peer-to-peer systems using the kompics component model," in *COMSWARE '09: Proceedings of the Fourth International ICST Conference on COMmunication System softWAre and middlewaRE*, (New York, NY, USA), pp. 1–9, ACM, 2009.

[57] S. Xie, B. Li, G. Keung, and X. Zhang, "Coolstreaming: Design, Theory and Practice," *IEEE Transactions on Multimedia*, vol. 9, no. 8, p. 1661, 2007.

[58] K. P. Gummadi, S. Saroiu, and S. D. Gribble, "King: Estimating latency between arbitrary internet end hosts," in *SIGCOMM Internet Measurement Workshop*, 2002.

[59] G. Tan and S. A. Jarvis, "A payment-based incentive and service differentiation scheme for peer-to-peer streaming broadcast," *IEEE Trans. Parallel Distrib. Syst.*, vol. 19, no. 7, pp. 940–953, 2008.

[60] D. B. West, *Introduction to Graph Theory (2nd Edition)*. Prentice Hall, August 2000.

[61] M. Meulpolder, J. A. Pouwelse, D. H. J. Epema, and H. J. Sips, "Bartercast: A practical approach to prevent lazy freeriding in p2p networks," in *IEEE International Symposium on Parallel&Distributed Processing*, (Washington, DC, USA), pp. 1–8, IEEE Computer Society, 2009.

[62] J. Mol, J. Pouwelse, M. Meulpolder, D. Epema, and H. Sips, "Give-to-get: Free-riding-resilient video-on-demand in p2p systems," in *Multimedia Computing and Networking 2008*, vol. 6818, SPIE Vol. 6818, January 2008.

[63] Z. Li and A. Mahanti, "A progressive flow auction approach for low-cost on-demand p2p media streaming," in *International conference on Quality of service in heterogeneous wired/wireless networks*, (New York, NY, USA), p. 42, ACM, 2006.

[64] R. Zhou, K. Hwang, and M. Cai, "Gossiptrust for fast reputation aggregation in peer-to-peer networks," *IEEE Trans. on Knowl. and Data Eng.*, vol. 20, no. 9, pp. 1282–1295, 2008.

[65] M. M. Zavlanos, L. Spesivtsev, and G. J. Pappas, "A distributed auction algorithm for the assignment problem," in *2008 47th IEEE Conference on Decision and Control*, pp. 1212–1217, IEEE, December 2008.

[66] C. Park, W. An, K. R. Pattipati, and D. L. Kleinman, "Distributed Auction Algorithms for the Assignment Problem with Partial Information," in *International Command and Control Research and Technology Symposium*, June 2010.

[67] B. Cohen, "Incentives build robustness in bittorrent," in *In Proceedings of the 1st Workshop on Economics of Peer-to-Peer Systems*, 2003.

[68] M. Jelasity, R. Guerraoui, A.-M. Kermarrec, and M. van Steen, "The peer sampling service: experimental evaluation of unstructured gossip-based implementations," in *Middleware '04: Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, (New York, NY, USA), pp. 79–98, Springer-Verlag New York, Inc., 2004.

[69] G. Berthou and J. Dowling, "P2p vod using the self-organizing gradient overlay network," in *SOAR '10: Proceeding of the second international workshop on Self-organizing architectures*, (New York, NY, USA), pp. 29–34, ACM, 2010.

[70] N. Drost, E. Ogston, R. V. van Nieuwpoort, and H. E. Bal, "Arrg: real-world gossiping," in *HPDC '07: Proceedings of the 16th international symposium on High performance distributed computing*, (New York, NY, USA), pp. 147–158, ACM, 2007.

[71] S. Guha and P. Francis, "Characterization and measurement of tcp traversal through nats and firewalls," in *IMC '05: Proceedings of the 5th ACM SIG-COMM conference on Internet Measurement*, (Berkeley, CA, USA), pp. 18–18, USENIX Association, 2005.

[72] J. Rosenberg, "Interactive connectivity establishment (ice): A methodology for network address translator (nat) traversal for offer/answer protocols," in *[Online]. Available: http://tools.ietf.org/html/draft-ietf-mmusic-ice-13, Jan. 2007.*

[73] J. Leitão, R. van Renesse, and L. Rodrigues, "Balancing gossip exchanges in networks with firewalls," in *Proceedings of the 9th International Workshop on Peer-to-Peer Systems (IPTPS '10)*, (San Jose, CA, U.S.A.), p. (to appear), 2010.

[74] L. DAcunto, M. Meulpolder, R. Rahman, J. Pouwelse, and H. Sips, "Modeling and analyzing the effects of firewalls and nats in p2p swarming systems," in *Proceedings IPDPS 2010 (HotP2P 2010)*, IEEE, April 2010.

[75] Y. Liu and J. a. Pan, "The impact of NAT on BitTorrent-like P2P systems," in *IEEE Ninth International Conference on Peer-to-Peer Computing, 2009. P2P'09*, pp. 242–251, 2009.

[76] L. DAcunto, J. Pouwelse, and H. Sips, "A measurement of nat and firewall characteristics in peer-to-peer systems," in *Proc. 15-th ASCI Conference* (L. W. Theo Gevers, Herbert Bos, ed.), (P.O. Box 5031, 2600 GA Delft, The Netherlands), pp. 1–5, Advanced School for Computing and Imaging (ASCI), June 2009.

[77] R. Price and P. Tino, "Adapting to NAT timeout values in P2P overlay networks," in *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pp. 1–6, IEEE, 2010.