ForestCast for Peer-to-Peer Live Streaming

A Solution to Heuristically Constructing Trees

AMIR PAYBERAH, FATEMEH RAHIMIAN

Master's Thesis at Royal Institute of Technology (KTH)
Supervisor: Ali Ghodsi
Examiner: Seif Haridi

KTH-ICT-ECS

Abstract

Media Streaming over Internet, as a popular high bandwidth application, is an expensive service in terms of resources. An approach to reduce the cost of this service is to use *peer-to-peer overlays*, in which each node shares its resources with the others. Thus, the capacity of system grows by the number of participating nodes. In this thesis we present *ForestCast*, a peer-to-peer live media streaming system. Within ForestCast, we have proposed a number of heuristics and examined their impact on the quality of service experienced by clients. To evaluate ForestCast, we have implemented a simulator, called *SICSSIM-B*, which is a stochastic discrete-event flow-level simulator that models bandwidth, link latencies and congestion. We have shown that by selecting appropriate heuristics, one can increase bandwidth utilization, quality of the media playback, and the fault-tolerance of the nodes, as well as decrease the playback latency and startup delay. These heuristics include considering the properties of the nodes when positioning them in the overlay, selecting multiple distinct suppliers for a node, fairly distributing data segments in the overlay, and continuously incorporating an incremental improvement to the overlay structure.

Key words: media streaming, peer-to-peer live streaming, application level multicast, overlay networks, peer-to-peer simulator.

Acknowledgments

We are deeply grateful to our supervisor, Ali Ghodsi, who basically introduced the main idea of ForestCast. He worked with us side by side and helped us with every bit of this research. He also taught us how to write a scientific text by patiently walking us through this thesis document. We are honored to have worked with our examiner, Professor Seif Haridi, and we thank him for his invaluable help and support during our work.

We are also thankful to Tallat Mahmood Shafaat, for the fruitful discussions and the knowledge he shared with us. He also has the main credit for building the first version of SICSSIM. We also thank Cosmin Arad, who helped us to setup our simulation environment, and Ahmad Al-Shishtawy for his suggestions and subtle hints. Besides, we are grateful to people of SICS Center for Networked Systems (CNS), and finally we acknowledge the financial support given to us by the CoreGrid project.

Contents

Abstract **iii**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Contribution . . . 2
1.3 Delimitation . . . 4
1.4 Outline . . . 5

2 Related work **7**
2.1 Classification Framework . . . 7
2.1.1 Finding Supplying Peers . . . 7
2.1.2 Data Delivery . . . 12
2.2 Existing Solutions . . . 16
2.2.1 Push Method Solutions (Single Tree) . . . 16
2.2.2 Push Method Solutions (Multiple Trees) . . . 18
2.2.3 Pull Method Solutions . . . 21
2.2.4 Push-Pull Method Solutions . . . 24
2.2.5 Other Solutions . . . 27
2.2.6 Related Work at a Glimpse . . . 29

3 ForestCast **31**
3.1 Solution . . . 32
3.1.1 Join Procedure . . . 33
3.1.2 Leave Procedure . . . 37
3.1.3 Failure Handling . . . 38
3.2 Incremental Improvement . . . 40
3.3 Remarks . . . 44

4 Simulator **47**
4.1 Challenges and Approaches . . . 47
4.2 SICSSIM-B . . . 49
4.3 Existing Solutions . . . 52
4.3.1 Criteria . . . 52
4.3.2 Simulators Review . . . 53

5 Evaluation **57**
5.1 Experimental Setting 58
5.2 Impact of Different Heuristics 59
5.2.1 Node Collection 60
5.2.2 Parent Selection 61
5.2.3 Startup Segment 63
5.2.4 Buffering Delay 66
5.3 Measurements at Scale 66
5.3.1 Bandwidth Utilization 68
5.3.2 Tree Depth 68
5.3.3 Startup Delay 69
5.3.4 Playback Latency 70
5.3.5 Received Quality 74
5.3.6 Disrupted Nodes 75
5.4 Impact of Incremental Improvement 76

6 Future Work **79**

7 Conclusion **81**

Bibliography **83**

Index **89**

List of Figures

1.1 Study road map of this documen 5

2.1 Locating supplying peers by a centralized method 8
2.2 Locating supplying peers in a hierarchical method 9
2.3 Locating supplying peer in DHT method 10
2.4 Locating supplying peers by controlled flooding method 11
2.5 Comparison of different methods for locating supplying peers 12
2.6 Data delivery through a single tree 13
2.7 Data delivery through multiple trees 14
2.8 Comparison of different methods of content delivery 15
2.9 Comparison of studied solution 29

3.1 Buffers are affected when nodes change their position 42
3.2 Lessen the buffer damage when nodes change their position 42
3.3 Arrange the subtrees when nodes change their position 43

4.1 Packet level simulation 48
4.2 Fluid based simulation 48
4.3 Nodes connection in a peer-to-peer overlay and in physical network . . . 49
4.4 SICSSIM-B overall structure 50
4.5 Bandwidth matrix 51
4.6 Properties of surveyed simulators 55
4.7 Complementary information about surveyed simulators 56

5.1 Bandwidth distribution measured in real scenarios 58
5.2 Playback latency by having BFS-DFS policy and BFS policy 61
5.3 Quality of received media by having BFS-DFS policy and BFS policy . . 62
5.4 Quality of received media by Distinct and Non-Distinct Parents policies 63
5.5 Head-segment policy 64
5.6 Mid-segment policy 64
5.7 Playback latency by Head-Segment and Mid-Segment policies 65
5.8 Quality of received media by Head-Segment and Mid-Segment policies . 66
5.9 Playback latency of peers for different buffering delay (join only) 67
5.10 Quality of received media for different buffering delay (join only) 67

5.11 Average tree depth . . . 69
5.12 Average time to join . . . 70
5.13 Average playback latency . . . 72
5.14 Playback latency of peers in join only scenario (join only scenario) . . . 72
5.15 Playback latency of peers in low churn scenario (low churn scenario) . . 73
5.16 Playback latency of peers in high churn scenario (high churn scenario) . 73
5.17 Average quality . . . 74
5.18 Fraction of nodes receiving varying levels of quality (low churn scenario) 75
5.19 Fraction of nodes receiving varying levels of quality (high churn scenario) 76
5.20 Average number of disrupted nodes . . . 77

Algorithm Listings

1 Join Procedure 34
2 Find Open Nodes 34
3 Find Open Nodes Per Stripe 35
4 Select Parents 36
5 Assign Priority To Stripes 36
6 Decide On The Startup Segment 37
7 Leave Procedure 39
8 Failure Handling Procedure 40
9 Incremental Improvement - Request Promotion 43
10 Incremental Improvement - Request Reconfiguration 44

Chapter 1

Introduction

Streaming media is a multimedia that is sent over a network and played as it is being received by end users, i.e. users do not need to wait to download all the media; instead, they can play it while the media is delivered by the provider. There are different approaches to media streaming. In this thesis work we look at some of them and introduce our solution to multicasting media over a network.

1.1 Motivation

Media Streaming over Internet is getting more popular everyday. Websites, such as *YouTube* [1], provide media content to millions of viewers. The conventional solution for such applications is the client-server model, which allocates servers and network resources to each client request. Since media streaming demands a high transmission rate and causes heavy load on servers, it is an expensive service in terms of resources. So the client-server model fails to provide such a service when the number of clients grows, unless resources are increased proportionally to the number of clients. Since there are few companies, like *Google*, who can afford providing such an expensive service at large scale, finding alternative solutions is an active field of research [67]. *IP multicast* is an efficient way to multicast a media stream over the network, but it is not used in practice due to its limited support by the Internet Service Providers. An alternative solution is *Application Level Multicast (ALM)* [7, 34], which uses *overlay networks* to distribute large-scale media streams to large number of clients. One type of overlay network is *peer-to-peer overlay*. Peer-to-peer overlay is a type of network in which each peer simultaneously functions as both client and server to the other peers on the network. In this model the peers who have all or part of the requested media can forward it to requesting peers. Since each peer contributes its own resources the capacity of whole system grows when the number of peers increases.

[1] www.youtube.com

Peer-to-peer streaming is challenging; because 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 the continuity of the playback is disrupted. What is more, in live streaming, it is expected that at any moment, clients receive points of the media that are close in time, ideally, 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 neighbors whooping it up for a goal, several seconds before they can see the goal happening. Satisfying these timing requirements is more challenging in a dynamic network, where (i) nodes join/leave/fail continuously, called *churn*, or (ii) network capacity changes due to network congestion etc.

Many different solutions have been already proposed for peer-to-peer media streaming, but few of them have been 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, you may ask users to 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.

The other problem with the existing solutions is that each solution comes with its own metrics and measurements. Hence a fair comparison of different approaches is difficult. Even for those solutions, which are mainly similar and differ only in some small parts, there is no clear way to truly compare how those little differences affect the quality of service.

1.2 Contribution

Our contributions to the media streaming problem are (i) designing an algorithm for peer-to-peer media streaming, called *ForestCast*, (ii) implementing a stochastic discrete event peer-to-peer simulator, which also models bandwidth and link latencies, called *SICSSIM-B*, and finally (iii) investigating how different heuristics will affect the quality of service experienced by clients.

In ForestCast we use a central server to handle the control messages and a peer-to-peer overlay to deliver the media stream to nodes. As we discussed, the large scale data, which is troublesome in media streaming, motivates us to move from the client-server model to peer-to-peer overlays. Constructing and maintaining such overlays incur a control overhead, which must be kept low so that it does not waste the access link capacities itself. This low overhead makes it feasible for a centralized directory to deal with a large number of control messages. Hence, we build the peer-to-peer overlay only for the bandwidth intensive part of the service, which is the data delivery to the nodes. Our proposed solution, differs from the existing solutions in that we solve the problem using a heuristic technique. ForestCast provides a framework, which can be configured with different constraints and objective

functions. In other words, we choose a performance centric approach that easily adopts to various design goals. So this solution can serve a wide range of media providers, who have their own requirements, priorities and objectives.

To evaluate our algorithm, we developed a simulator, called *SICSSIM-B*. Currently there are very few peer-to-peer simulators available. These simulators either have to consider all the details of packet transmissions in network, which makes them barely scalable for simulating huge number of data packets, as it is required in media streaming, or they have to abstract all the underlying layers of network, which leads to inaccurate measurements. Furthermore, none of the available simulators can simulate data streams and measure the efficiency of streaming in terms of timing requirements of data blocks. Our simulator enables such measurements while keeping the simulator scalable to evaluate the behavior of the overlay at large scale.

Our solution also serves as a platform to study how different heuristics can change the properties of data delivery overlays. What is desired in all the existing algorithms, is to construct and maintain an efficient data delivery overlay, but there is no common definition for being "efficient". This is why the algorithms are not fairly comparable to one another. If we use a central server with global knowledge, which can quickly make the best possible decisions with respect to certain criteria and objectives, we will end up in an optimum overlay for the respective configuration. This overlay can then be used as a reference of measurement, by the algorithm designers and can help them to decide on the best policies they can apply in order to improve their overlay.

Our experiments show that for positioning nodes in the trees, it is very important to consider the properties of the joining nodes. For example it is a wise decision to put nodes with petty bandwidth as far as possible from the media source, by using a Depth First Search policy. Even during the lifetime of an overlay, incorporating an incremental improvement, which pulls the stronger nodes up the trees and pushes the weaker nodes down, will result in a more efficient overlay. Note to be taken, that a strong node is not only a node with a high bandwidth, but the one who has also stayed long in the system.

We also observe that to maximize bandwidth utilization, it is of crucial importance to uniformly distribute different parts of the media, e.g. substreams, in the overlay. For example, nodes are better to prioritize forwarding a substream, which they have forwarded less. A fair distribution of data in the overlay, prevents a situation in which some part of the data becomes rare, and even though there is a free capacity in the network for data delivery, data is not accessible and the bandwidth can not be utilized.

What is more, we show that selecting distinct parents for a node, makes it more

fault tolerant and will decrease the number and severity of disruptions nodes may ever experience.

Another interesting result we come up with, is that a greedy approach to choose the startup segment for a joining node, would result in not only a better playback latency, but also, surprisingly, a better quality. More exactly, although one may think starting with an earlier point of the media would make a node more fault tolerant, we conclude that it is always better to provide the nodes with the most recent data available, that is the largest segment all the parents have in common.

1.3 Delimitation

As mentioned before, one of our motivations is to enable a fair comparison of different algorithms and see how close they are to an optimum solution. This is possible only if we can assume a global view of the whole overlay, e.g. by assuming a centralized directory that knows about all the peers and the structure of the overlay. Hence, even though the study of a fully distributed system without any centralized directory is interesting, we exclude it from this thesis document and leave it for the future work. See Chapter 6.

The centralized directory that we introduce in ForestCast can either be a single server or a set of distributed servers. Using a set of distributed servers would eliminate the single point of failure problem, which may be of interest for media providers. However, replacing a single server by a set of distributed servers is an independent field of research [54, 49, 17] and is out of scope of this document. So we only assume a single server as the centralized directory.

As we will see in Chapter 2, one approach to peer-to-peer media streaming, which is attaining more interest these days, is the combination of *push models* and *pull models*. In a push model, data is multicast through predefined paths in the overlay, but in pull model data is explicitly requested from other nodes who possess it. In this document we only focus on push model, i.e. how we can construct the multicast paths, so that we can guarantee a certain level of quality of service. This method can be further improved by incorporating a pull model for acquiring those data segments a node misses in the pure push model; but we leave it for future work. See Chapter 6.

Furthermore, a field of study, which is relevant to media streaming, is the encoding techniques. There is work still ongoing in this area [8]. One of the proposed solutions is *Multiple Description Coding* or *MDC* [18], which we use in ForestCast and will describe further in Chapter 2. Although ForestCast is independent of the media encoding technique, in this document we do not consider the use of other encoding techniques.

1.4 Outline

The rest of this document is organized as follows. Chapter 2 provides a framework for classifying and comparing solutions, and reviews the related work in terms of the framework. Chapter 3 presents ForestCast, our solution to media streaming over a peer-to-peer overlay. It also provides a number of heuristics, which can be applied to construct data delivery overlays. Chapter 4 presents the existing challenges in simulating peer-to-peer algorithms, as well as a brief description of the existing simulators. In this chapter we also introduce our simulator, SICSSIM-B, which we used for the purpose of evaluation. Chapter 5 shows how ForestCast performs in different scenarios and with different configurations. In Chapter 6 we hint on some future research directions, and finally in Chapter 7, we conclude the work.

Figure 1.1 shows a road map of how one can read this document. The arrows show the dependency between chapters, so readers may skip some of the chapters and go directly to what is of their interest.

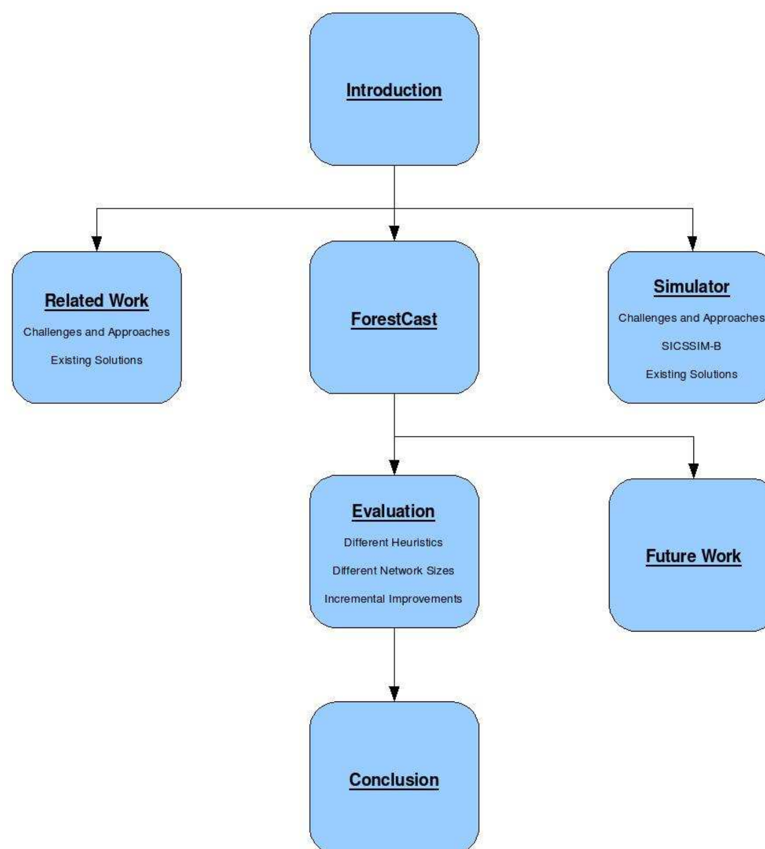

Figure 1.1: Study road map of this document

Chapter 2

Related work

In this chapter we review the related work, but before delving into details of individual solutions, we provide a framework in which solutions can be classified and compared.

2.1 Classification Framework

We believe each peer-to-peer media streaming solution should provide the answer to the following two main questions:

1. How to locate supplying peers?
2. How to deliver content to peers?

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

2.1.1 Finding Supplying Peers

The first question we need to answer when studying a peer-to-peer media streaming solution is that how a peer finds its supplying peers. According to [67] the main methods, which serve this purpose, are:

- Centralized method
- Hierarchical method
- DHT based method
- Controlled flooding method
- Gossip based method

Centralized method

The first solution for locating supplying peers is to use a *centralized directory*. In this method the information about all peers, e.g. their address or available bandwidth, is kept in a centralized directory. The centralized directory maintains the overall topology of the overlay. Two sample algorithms in this category are CoopNet [42] and DirectStream [19].

As we can see in Figure 2.1, whenever a new peer arrives, it sends a join request to the server and the server finds one or more proper data provider(s) for it, considering the local information about the overlay and the properties of the joining peer. Then the peer and its provider(s) can communicate with each other. When a node wants to leave, it sends a leave request to the server, and waits until it receives the server's grant. Meanwhile the server finds a substitute provider for the leaving node's children. If a node fails, it takes time before the server detects the failure and its information would be out-of-date meanwhile. To detect failures, server can periodically probes the nodes to see if they are alive. But as the number of nodes grows, the burden of server links increases. Another solution is to get help from peers to detect the failure. In this case, whenever a peer detects the failure of another peer, i.e. its parent, it informs the server of that failure.

The advantage of the centralized model is that 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 peer can join the system. But the central server can be replaced by a set of distributed servers. This is described in Section 3.3.

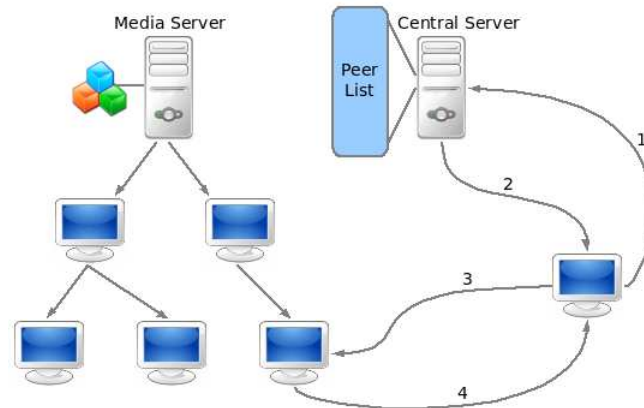

Figure 2.1: Peer location using centralized method. The peer sends its join request to the central server (1), the central server finds proper providers for the peer and informs the peer about them (2), the peer contacts them (3), and then the provider(s) sends data to peer (4).

Hierarchical method

The next method for locating supplying peers is using a *hierarchical model*, which is used in some systems such as Nice [3], ZigZag [57], and Bulk Tree [1]. In this method several *layers* are created in the overlay network. As we can see in Figure 2.2, the lowest layer (layer-0) contains all the peers. The peers in this layer are grouped into some clusters, according to a property defined in the algorithm, e.g. the latency between peers. One peer in each cluster is selected as a *head*. The selected head for each cluster becomes a member of one higher layer (layer-1). By clustering the peers in this layer and selecting a head in each cluster, we can form the next layer, and so on, until we end up in a layer consisting of a single peer. This single peer, which is a member of all layers is called the *rendezvous* point (The node in the layer-2 in Figure 2.2).

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 peers on the next down layer in the hierarchy. The new node probes the list of peers, and finds the most proper one and sends its join request to that peer. The process repeats until the new node finds a position in the structure, where it receives its desired content.

This model is scalable and it guarantees to find proper providers for the peers.

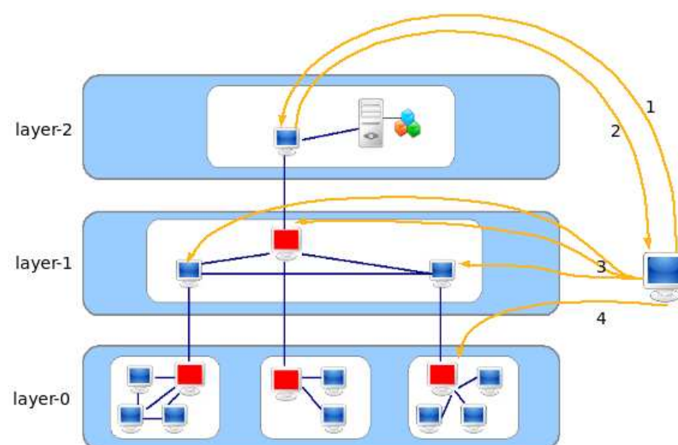

Figure 2.2: Peer location using hierarchical method. In each layer the light area shows the a group of peers (cluster). The dark node in each cluster is the head of the group, which is selected to be a member of higher layer. Each joining node sends its join request to the peer in the highest layer cluster (1), that peer introduces its clustermate at the lower layer (2), then the new node finds the best peer among them (3), and send its request again to that (4).

DHT based method

Third approach to locate supplying peers is based on *Distributed Hash Table (DHT)*. As described in [17],

> "A distributed hash table is a hash table, which is distributed among a set of cooperating computers. Just like a hash table, it contains key/value pairs. The main service provided by a DHT is the lookup operation, which returns the value associated with any given key. In the typical usage scenario, a client has a key for which it wishes to find the associated value. Thereby, the client provides the key to any one of the nodes, which then performs the lookup operation and returns the value associated with the provided key. Every node should be able to lookup the value associated with any key. Since all items are not stored at every node, requests are routed whenever a node receives a request that it is not responsible for. For this purpose, each node has a routing table that contains pointers to other nodes, known as the node's neighbors. Hence, a query is routed through the neighbors such that it eventually reaches the node responsible for the provided key. Figure 2.3 illustrates a DHT which maps file names to the URLs representing the current location of the files".

SplitStream [7] and Pulsar [30] are two sample algorithms that work over a DHT. In these systems each peer keeps a routing table including the address of some other peers in the overlay network. Whenever a new node comes to the system, it sends request message to find a supplying peer. The peers forward the message according to their local routing tables, until it reaches the responsible node for handling that request (Figure 2.3). 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.

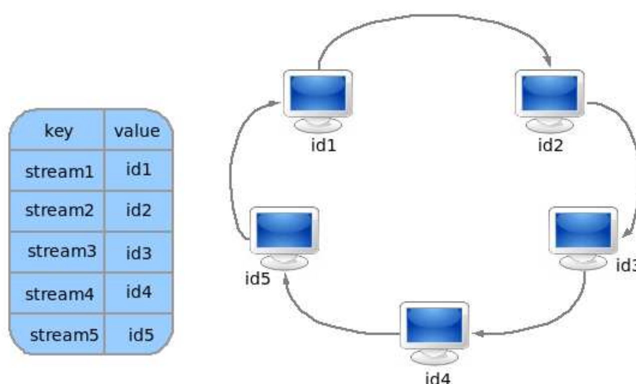

Figure 2.3: Locating supplying peer in DHT method. The media stream files are distributed among the peers, and the peers keep routing pointer to each other. If an application sends a lookup request to node $id4$ to find $stream1$, node $id4$ routes the request to node $id5$, because it does not contain that file, and node $id5$ also routes the request to node $id1$, which contains the $stream1$ and can response the request.

Controlled Flooding method

The forth method is *controlled flooding*, which is originally proposed by Gnutella [56]. GnuStream [23] is a system who used this idea to find supplying peers. In this system, each peer has a *neighbor set*. Whenever a peer seeks a provider, it sends its query to its neighbors. Each peer forwards the request to all of its own neighbors 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 peer, who receives the request satisfies the peer selection constraints, it will reply to the original sender peer (Figure 2.4).

This method has two main drawbacks. First, it generates a significant traffic and second, there is no guarantee for finding appropriate providers.

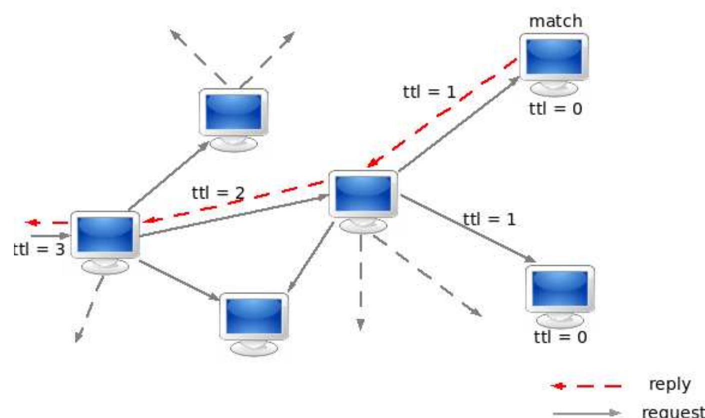

Figure 2.4: Locating supplying peers by controlled flooding method. One application sends its request with $TTL = 3$. Each peer decreases the TTL and rebroadcast it to all neighbors except the one who received the request from. The peers who matched the request response to the origin peer.

Gossip based method

The last approach to find supplying peer is the *gossip based method*. Many algorithms are proposed based on this model, e.g. CoolStreaming/DONet [68] and PULSE [46]. In this method each peer maintains a neighbor set, which is a partial view of the whole overlay. Whenever a new peer comes in to the system, it contacts a *bootstrapping point* and is given some random neighbors. Each peer periodically communicates with its neighbors to maintain a number of available partners in the presence of node departures. Peers also periodically send their data availability information to their neighbors to enable them find appropriate suppliers, who possess data they are looking for.

This protocol is scalable and failure-tolerant, but because the neighbor selection is random, sometimes the appropriate providers are not found in a reasonable time.

A brief comparison

Figure 2.5 shows a brief comparison of the above mentioned techniques. Since in the centralized method, the central server maintains the information of whole system and all peers send their information periodically to it, it has the lowest scalability. The controlled flooding method also is not very scalable because of generating a large traffic. In contrast, the hierarchical, DHT-based, and gossip-based methods have a good scalability.

In control flooding method, the system can not find supplying peers if they locate out of TTL scope. Small values of TTL deteriorates this problem. The gossip-based method also can not guarantee finding supplying peers in a bounded time, because of the random nature of neighbor selection.

In centralized method, the peers do not play any role in selecting their providers, and leave it to be decided by the central server. Hence, the peers do not maintain information about other peers in system and they only know the central server. So the order of information kept in them is $O(1)$. In other methods, though, peers have a neighbor set and a partial view of the system, so the order of information they maintain is $O(logN)$.

Approach	Scalability	Search guarantee	Order of information kept at a peer
Cenralized method	low	yes	O(1)
Hierarchical method	high	yes	O(log N)
DHT based method	high	yes	O(log N)
Controlled flooding method	low	no	O(log N)
Gossip based method	high	no	O(log N)

Figure 2.5: Comparison of different methods for locating supplying peers

2.1.2 Data Delivery

After finding supplying peers, the next question to be answered is how the peers deliver data to one another. There are three main methods for data delivery:

- Push method
- Pull method
- Push-Pull method

Push method

The main idea in push method is that each peer pushes the data it receives to a number of other peers, according to predefined paths. There are two main categories in push method: *single tree* and *multiple trees* [67]. In single tree model (Figure 2.6), a multicast tree is constructed among all the peers in system and each peer is responsible to send data to its children. The advantage of single tree structure is its simple topology and no redundancy in data delivery, because each peer receives the data from its single parent. The tree topology has some drawbacks though. Firstly, it does not utilize the upload bandwidth of leaves and only the interior nodes, which are a minority, carry the burden of data forwarding. Secondly, the tree is a fragile structure. Because each node is connected to a single parent, if the parent fails, the peer and the subtree under it will suffer a data loss.

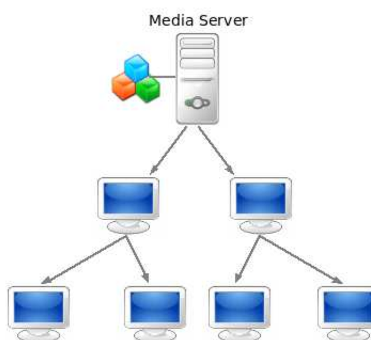

Figure 2.6: One single tree is created among the peers of the system, which is rooted at the media server

To mitigate these problems, the multiple trees topology is proposed (Figure 2.7). In this topology the stream is split into several substreams, called *stripes* or *descriptions*, and a separate tree is responsible for forwarding a single stripe. There are several techniques to split a stream into a number of stripes e.g. *layering technique* [8] or *Multiple Description Coding (MDC)* [18]. MDC is one of the splitting techniques, which fragments the media stream to multiple independent substreams. In order to decode the media stream, any stripe can be used, however, the quality improves with the number of stripes received in parallel.

In multiple trees model, the nodes can join as many trees as the number of stripes they want to receive. A node receives each stripe from different providers, and in case of failing any of its providers, the node can receive other parts of the media from its other providers. Hence a node's failure will not interrupt the stream of its children as long as they receive other stripes from different paths. Such events only causes a temporary loss of quality before the damage is fixed. Some solutions, like SplitStream [7], manage the nodes such that a node can be an interior node in one tree and leaf in other trees. So, because the leaves in one tree are interior nodes in

other trees, their upload bandwidth are utilized.

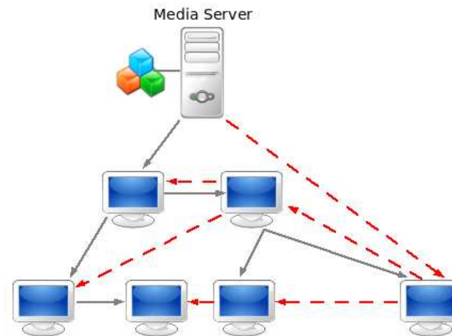

Figure 2.7: Two trees are created among the peers. Both trees are rooted at media server. One tree is shown by dashed line and the other one is shown by continues line. Having two trees means that the media stream is split in to two stripes, and each stripe is delivered by one tree.

Pull method

Another method for data delivery is pull method. The main idea of pull method is that each peer explicitly requests required data from other peers. Each peer has a neighbor set and it periodically exchanges data availability information with its neighbors. Whenever a peer receives such information from other peers, it learns about the data segments it has not received yet. It then requests the missing data from the peers in the neighbor set, who possess it. To motivate the peers in pull method to cooperate in data delivery and share their upload bandwidth, one way is to use *tit-for-tat* algorithm as used in Bittorrent [10].

Pull methods are divided into two main groups: *receiver-driven* and *chunk-driven* [61]. In the receiver-driven approach, the receiver requests the stream from other peers, and if a peer accepts the request, then a session will be established between them. From then on, the supplying peer will transfer the stream to the requesting peer. GnuStream [23] and CollectCast [21] use this approach. On the other hand, in chunk-driven approach, the media stream is divided into *chunks* or *data segments*, and the receiver requests each chunk explicitly from other peers, typically different ones. Put differently, there would be one request per data chunk. Coolstreaming/-DONet [68] and PULSE [46] use this approach.

Although the random partner selection in pull method improves resource utilization and load balancing, but it can not guarantee that the data blocks are received in time. Also, since neighbors should exchange data availability information periodically, the control overhead is higher than tree-based models.

Push-Pull method.
This method, which has gained more attention in recent years, combines the advantages of push and pull approaches. Data delivery in this method has two phases. In the first phase a tree structure (either single or multiple) is created and each peer pushes the data to its children. In the second phase each peer requests the missing data blocks from its neighbor. Pulsar [30] and Prime [31] are two examples that work with this method.

A brief Comparison
Figure 2.8 shows a brief comparison of different methods for data delivery. The single tree model has poor resilience to node failure, because the nodes receive data from a single parent and if that parent fails, its children can not receive any data any more. On the contrary multiple trees and pull model are more resilient, because in the former nodes receive data from multiple parents, and in the latter each node has a number of neighbor and can pull data from them.

The load balancing in single tree model is not fair, because we do not use the upload bandwidth of leaves and only interior nodes carry the burden of data forwarding. This problem is solved in multiple trees, since leaves in one tree can be interior nodes in other trees. Also in pull model, all nodes in a neighbor set can use the upload capacity of one another. As mentioned before to motivate nodes to share their upload bandwidth, we can use different policies, e.g. tit-for-tat.

In push model (single tree and multiple trees) after creating the tree(s), there is no extra overhead to deliver data, and nodes simply push data to their children; whereas in pull model the nodes should periodically inform the others about the availability of data and will produce some overhead. In push-pull method the data is delivered by push model and the nodes pull data only if they can not receive it in push phase. This model also has a low overhead.

Approach	Resilience to node failure	Multiple supplier	Load balancing	Data delivery overhead
Push method (single tree)	poor	no	no	low
Push method (multiple trees)	good	yes	yes	low
Pull method	good	yes	yes	high
Push-Pull Method	very good	yes	yes	low

Figure 2.8: Comparison of different methods of content delivery

2.2 Existing Solutions

In this section we briefly introduce some of the existing peer-to-peer media streaming algorithms. We have grouped these algorithms by the method they deliver data. For each solution we describe the major contributions, how nodes find their supplying peers, i.e. how the data delivery overlay is built, how it deals with node leaves and failures, and finally what the drawback are.

2.2.1 Push Method Solutions (Single Tree)

ZigZag

ZigZag [57] is a single tree algorithm for peer-to-peer media streaming, which uses hierarchical method to find supplying peers. ZigZag separates logical and physical connections between peers, namely *administrative organization* and *multicast tree*, respectively. Administrative organization builds and maintains the overlay structure i.e. the multicast tree, and data is delivered through this multicast tree. The main contribution of ZigZag is how to map peers into administrative organization, and build the multicast tree based on it, and how to update these two structures under network dynamics.

ZigZag organizes peers into a *multi-layer* hierarchy of bounded-size *clusters* ($[2, 3k]$ in the highest layer and $[k, 3k]$ in other layers that k is constant). The lowest layer (*layer-0*) contains all the peers. One peer in each cluster is selected as the *head* of that cluster, and becomes a member of the next upper layer; and so on until it reaches a layer with only one cluster. Note that, media server is a member of all layers and head of all clusters it belongs to. ZigZag builds a multicast tree over this organization.

Each peer periodically communicates with its clustermates, children and parent, to maintain its position in the multicast tree and the administrative organization. Two important values are sent to a parent by its children: *Reachable(X)* and *Addable(X)*. A Boolean flag *Reachable(X)* is true if and only if there exists a path in the multicast tree from node X to a layer-0 peer, and a boolean flag *Addable(X)* is true if and only if there exists a path in the multicast tree from node X to a layer-0 peer whose cluster's size is in $[k, 3k - 1]$.

Whenever a new peer P comes in to the system, it submits a join request to the media server. If the administration organization has only one layer and its cluster has enough capacity to accept a new member, the new node connects to the media server directly. Otherwise, the join request is redirected along the multicast tree downward until a proper peer is found. In this process, if a peer X, who receives the request, is a leaf, P is added to X's cluster and X's parent is selected as the parent of P in the multicast tree. Otherwise, if peer X is addable, it finds an addable peer Y among its children, whose distance to source is minimum, and forwards the

request of P to Y. If the peer is not addable, it finds a reachable peer Z among its children with shortest distance to source and forwards P's request to Z. This process continues until the peer reaches a leaf. Whenever size of a cluster exceeds $3k$, it is split into two clusters.

In case of failure of a peer X at layer-j, for each cluster in layer-(j-1), whose non-head members are children of X, the head of the cluster, Y, is responsible for finding a new parent for the orphaned peers. Y selects a layer-j non-head clustermate with the least degree as the new parent. Furthermore, since X used to be the head of j clusters (layers-0, layer-1, ..., layer-(j-1)), those clusters must find a new head. This is done by selecting a random clustermate of X at layer-0 and letting it to replaces X in all those j clusters. What is more, if the size of a cluster becomes less than k, that cluster is merged with another cluster of the same layer.

The main drawback of ZigZag is that it does not consider the upload bandwidth capacity of peers in join procedure. Also, because ZigZag creates single tree connection between peers, it has the general problems of single tree model, such as not using upload bandwidth of leaves and vulnerability to failure of interior nodes.

DirectStream
DirectStream [19] is a peer-to-peer video streaming system that provides video on-demand service with *VCR* operation support. It uses a centralized directory, called *AMDirectory*, to find providers for each peer. The AMDirectory itself, is implemented over Scribe [50] and Pastry [49], and provides a lookup service. It keeps track of all servers and clients participating in system, and helps new clients to obtain the required service.

DirectStream may comprises sevral media servers with their independant contents. For each stream a user community is created in AMDirectory service and the media servers register themselves in them as servers. New coming peers send their request to AMDirectory to find proper parent. After setting up the connection with parent, the peer registers itself in AMDirectory. A set of clients, who arrive close in time, form a forwarding tree. DirectStream uses a QoS-sensitive peer selection algorithm to construct the streaming overlay. Each client sends its request to AMDirectory service and receives a list of candidate suppliers. It then probe those nodes and selects the one with he smallest distance to bandwidth ratio as its parent.

A similar procedure is used to support client recovery and VCR functionality. By VCR we mean *jump forward*, *jump backward* or *pause* the media stream. If a client leaves the system without earlier notification or uses VCR functionality, its children lose their parent and cannot receive the stream any more. In this case, the children start the recovery process, which is the same as a new client joining process.

The main drawback of DirectStream is using single tree for each media stream.

In this model, while a recovery process is going on for some node, that node does not receive any data. It also does not utilize the upload bandwidth of leaves.

2.2.2 Push Method Solutions (Multiple Trees)

SplitStream
SplitStream [7] is one of the solutions to multicast a stream through multiple trees that uses a DHT-based method to find supplying peers. More exactly, the algorithm is implemented over *Pastry* [49] and *Scribe* [50]. The key idea of SplitStream is to split the stream into different independent stripes, using MDC [18], and multicast each stripe using a separate tree. To ensure that the forwarding load can be spread across all participating peers, a forest of stripe trees is constructed in a way that a node is an interior node in at most one stripe tree and is a leaf node in all the other ones. Such a set of trees is called *interior-node-disjoint*.

Nodes join as many trees as their desired indegree is, and they enforce their outbound bandwidth limit by rejecting children beyond their outdegree capacity. Since some nodes may be rejected, special mechanisms are introduced to make sure no node remains parent-less. These mechanisms are *push down* and *anycast*. When an overloaded node receives a request from a prospective child, it either rejects this child or accepts it and rejects one of its existing children, which is less desirable than the new child. A node is more desirable if its node id is closer to its parent node id. In both cases the rejected child contacts one of the children of the overloaded node. At each step the orphaned node, either finds a parent or is pushed down the tree once more. If the prospective child hits a leaf without finding a parent, then it uses an anycast to find a node with free capacity in the tree regardless of its node id, if such a node exists at all.

one of the main problems with SplitStream is the impact of nodes with heterogeneous bandwidth on its efficiency. In Bharambe et. al. [4] this problem is nicely discussed:

> "while the pushdown and anycast operations help Scribe cope with heterogeneous node bandwidth constraints, they may result in the creation of parent-child relationships which correspond to links that are not part of the underlying Pastry overlay. We term such links as non-DHT links. We believe these non-DHT links are undesirable because: (i) the route convergence and loop-free properties of DHT routing no longer apply if non-DHT links exist in significant numbers; and (ii) such links require explicit per-tree maintenance which reduces the benefits of DHTs in terms of amortizing overlay maintenance costs over multiple multicast groups (and other applications)."

Another problem is that in an interior-node-disjoint, nodes are purposefully placed in different distances from the root of multiple trees, which means they receive dis-

tinct stripes with different latencies. This is undesirable for a live media streaming application, which involves strict timing constraints. The problem seems negligible when the depth of the trees are small, e.g. three or four. But it is augmented when the system scales to trees with larger depth and nodes are placed in diverse distances from the source; because for a node to playback the media, it has to either wait long enough to receive all the stripes it is supposed to receive, or to ignore the late ones. The former will either increase the source-to-end delay or disrupt the continuity of the media; while the latter wastes the bandwidth of both sender and receiver and unnecessarily burdens the network.

Orchard

Orchard [34] is an *Application Level Multicast (ALM)* algorithm for peer-to-peer live streaming. Orchard is implemented over unstructured peer-to-peer system and uses gossip-based method to find supplying peer. Each peer maintains a *neighborset*, which is a partial view of the network and its member are selected randomly. The peers learn about each other by performing limited broadcast. Orchard uses MDC [18] to split a video stream into several substreams, called *descriptions*, and creates a dynamic tree for each of them. Each description is represented by a colour, and the peers get the colour of the first description they receive.

The main goal of Orchard is to overcome the *free riding* problem. It copes with this problem by forcing peers to strike *deal* with other peers to receive descriptions. It means that for each description received, one description should be sent, possibly to a third peer. The mechanism of deals ensures that peers contribute as much outgoing bandwidth as their incoming bandwidth is.

A new peer P, which has not received any description yet, is blank. This node, finds a non-blank peer Q in its neighbor set and sends its join request to it. If Q accepts the request, it redirects one of the streams, which it is already forwarding to a third node R, to P and P would send that steam to R in return. This mechanism is called *redirection*. To get other descriptions, P checks its neighbor set to find other peers who have a colour that it does not have. Then it tries to strike a deal with them to *exchange description*. The only exception, when a node can receive some description for free, is when it connects to the media source.

Whenever a peer leaves the system or fails, other peers will stop receiving the descriptions they get from that peer. In such a case, all the deals with that peer and any data forwarding based on those deal are canceled. This deal cancellation may then propagate to other trees as well. In other words, a single node failure may easily turn to a catastrophic failure. To handle this problem, Orchard uses *backup parents*. Each peer keeps track of those peers in its neighbour set that can serve as a substitute parent, if the current parent fails. If the failed node is a peer close to source, finding a backup parent may be more difficult. So when the children of failed node try to rejoin the system, they might have to rejoin a parent of a different

colour. This makes the colour of the failed peer becomes rare in the network. To tackle this problem, each peer finds the rare colours, by tracking the colours in its neighbor set, and changes its colour whenever its beneficial. A peer benefits from switching colour, if it enables it to strike more exchange deals than it could before switching.

The first drawback of Orchard is its unrealistic assumption about incoming and outgoing bandwidth of peers. It assumes all peers have sufficient incoming and outgoing bandwidth to receive and send all descriptions. All the introduced mechanism for stiking deals highly depend on this assumption, whereas in real world we have peers with heterogeneous bandwidths. Hence the algorithm can easily result in unsuccessful joins in real scenarios. Also it does not use the outgoing bandwidth of peers properly, because it insists that all peers have a strictly balanced download and upload rate.

ChunkySpread

ChunkySpread [59] is a multitree unstructured peer-to-peer multicast. More clearly it consists of two parts: (i) an unstructured random graph, which it uses to find supplying peers by, and (ii) a multitree structure, which it uses to deliver data through. ChunkySpread assumes the media is split into several stripes and each stripe is multicast on a separate tree. Nodes gossip to select their parents with respect to certain requirements. The first requirement is to avoid *loop*. To do so, any data packet is marked with the identities of every node who has forwarded the packet. The other important criteria for parent selection is satisfying the target load of a node (the number of stripes a node is willing to transmit), as well as the maximum load of a node (the maximum number of stripes a node can transmit). Complementary considerations are tit-for-tat and latency, which can be used for further improvements of the structure.

In ChunkySpread at first a random graph is built over peers by using *SwapLinks* protocol [60]. Whenever a new node comes into the system, in an initial node discovery phase, it contacts a rendezvous node and is provided by a set of existing nodes. Then it takes random walks over the existing graph through those nodes, and randomly selects its neighbors. Nodes periodically exchange local information, i.e. load, latency, and looping information, with their neighbors. A node uses the information about its neighbors, to select the best possible parent for each stripe it is willing to receive.

The first drawback of ChunkySpread arises from a shortcoming in SwapLinks: in a dynamic environment where nodes leave or fail randomly, maintaining uniform neighbor sets, which respect the indegrees and outdegrees of the nodes, incurs a large control overhead. Another drawback is that ChunkySpread can not guarantee that a node finds at least as many different stripes as it requires in its neighbor set. Because there is no such consideration, whatsoever, while the graph is constructed.

Note that adding this constraint to the previous ones, not only increases the overhead of graph construction, but also violates the randomness of neighbor selection, which may further results in losing the nice properties of a random graph. What is more, ChunkySpread always favors satisfying load constraints over latency constraints. It neither considers the deadlines for experiencing a smooth media playback, nor the latency deviation by which a node receives different stripes from different paths. Hence, the objectives defined in ChunkySpread do not seem to be aligned with the inherent timing requirements of live streaming applications.

2.2.3 Pull Method Solutions

CoolStraming/DONet

CoolStreaming/DONet [68] is a chunk-driven overlay network for live media streaming. Peers in DONet gossip to find a number of *partners* and they use pull method to retrieve the stream data from their partners. The core idea of DONet is very simple: each peer periodically informs its data availability information to its partners, retrieves unavailable segments from them, and also provide segments to them. DONet tries to solve three practical challenges: (i) how to select partners for each peer, (ii) how peers acknowledge the availability of *data segments* to their partners, and (iii) how peers retrieve segment from their partners. Neither the partnerships nor the data transmission directions are fixed in DONet. A peer can be either a receiver or a supplier, or both, depending dynamically on the data availability information of each peer. An exception is the source node, which is always a supplier, and is referred to as the *origin node*.

Every DONet peer maintains a partial view of the identifier of other overlay nodes (*mCache*), and the availability of the data segments in their buffer (*Buffer Map* or *BM* for short). Each node periodically exchanges its *BM* with its partners, and then schedules which segment is to be fetched from which partner accordingly. There are two constraints for scheduling: (i) the playback deadline for each segment and (ii) the heterogeneous streaming bandwidth from the partners. If the first constraint can not be satisfied, then the number of segments missing deadlines should be kept minimum.

When a node joins to the system, it first contacts the origin node, which randomly selects a deputy node from its *mCache* and redirects the new node to the deputy. The new node then obtains a list of partner candidates from the deputy, and contacts these candidates to establish its partnerships in the overlay. To create and update the *mCache*, each peer periodically generates a *membership message* to announce its existence. DONet uses *SCAM* [15], which is a scalable gossip membership protocol to distribute membership messages among peers.

When a node wants to leave, it issues a departure message. But if a node fails, a partner that detects that failure will issue the departure message on behalf the

failed node. The departure message is gossiped similarly to the membership message. Each node, who receives this message, removes the entry for the departed node from its mCache.

The main drawback of CoolStreaming/DONet is that notifying peers and subsequently requesting segments potentially results in long delays before any data is exchanged. Also due to the random selection algorithm, the quality of service cannot be guaranteed. Another drawback of CoolStreaming/DONet is that it assumes that all the peers can (and are willing to) cooperate in the replication of the stream, while it is possible to have selfish peers in system that do not want to share their upload bandwidth.

PULSE

PULSE [46] is a chunk-driven peer-to-peer live streaming system that uses gossip-based method to find supplying peers. The data delivery approach used in PULSE is pull model. PULSE assumes that: (i) peers have some knowledge of the other peers, (ii) peers can estimate their point in the stream with respect to media clock, and (iii) they can estimate their maximum outbound bandwidth. The idea of PULSE is to place peers in system according to their current *trading performances*, e.g. to locate resource-rich peers near the media source.

The media source splits the stream into a series of *chunks*. The core algorithm of PULSE answers three main questions: (i) how each peer selects its partner for data exchange, (ii) how chunks are chosen and scheduled to be sent, and (iii) which chunks should be requested from which partners. The idea used for peer selection is an altruistic tit-for-tat algorithm similar to the one used in BitTorrent, and a simple cumulative trust metric. The connection between peers are the result of pairwise negotiations and data exchanges. The chunks to be sent are selected comparing the requests received from each peer to the chunks currently held in the whole local buffer. The selected chunks are then sorted using "Least Sent First, Random", and the first one is chosen for sending. The algorithm for chunk requests is similar to the heuristic used in CoolStreaming/DONet. Its purpose is to request the rarest chunks among those that are locally available, and to distribute the requests across different possible providers.

The main problem of PULSE, like other pull-based algorithms, is that there is no guaranteed level of quality of service due to the random selection algorithm.

ChainSaw

Chainsaw [43] is a chunk-driven peer-to-peer overlay multicast that uses a randomly constructed graph with a fixed minimum node degree. It uses gossip-based method to construct the random graph. Every peer connects to a set of nodes that are called *neighbors*. Data is divided into small size packets and disseminated using a simple request-response protocol. Whenever a peer receives a packet, it sends a

notify message to its neighbors. Each peer creates a list of required packets and their availability in its neighbor set. Then it randomly selects some packets from this list and requests them via a *request* message. To insure that a peer does not request the same packet more than once, the peer keeps track of what packets it has already requested from every neighbor. It also limits the number of outstanding requests with a given neighbor, to ensure that requests are spread over all neighbors.

In Chainsaw, the media source node, called *seed*, generates a series of new packets with monotonically increasing sequence numbers at constant rate. The seed maintains a list of packets that have never been uploaded before. If the list is not empty and the seed receives a request for a packet that is not on the list, the seed ignores the requested sequence number, sends the oldest packet on the list instead, and deletes that packet from the list. This mechanism, called *Request Overriding*, ensures that at least one copy of every packet is uploaded, and the seed will not spend its upload bandwidth on uploading packets that could be obtained from other peers, unless it has spare bandwidth available.

The main problem of Chainsaw is that it can potentially incur high network and CPU overheads due to per packet notifications. The other problem is that, due to the random packet selection, it can not handle special cases e.g. when a packet is rare or some of the packets have priority over the others. Although the authors mentions this problem themselves, they do not propose any solution for it.

GnuStream

GnuStream [23] is a receiver-driven peer-to-peer media streaming system, which is built on top of Gnutella [56]. GnuStream uses a controlled flooding method to find supplying peers, and uses pull method for data delivery. Each peer who is interested in receiving the media stream, finds multiple supplying peers to provide it with the stream. Each streaming session is controlled by the receiver peer.

Whenever a peer $P1$ looks for a media stream, it calls the Gnutella search service, and as a result it is given a number of suppliers (for example $P2 - P5$). Then it selects a number of peers, from the list of its suppliers, such that the aggregated bandwidth of them is sufficient for the stream (for example $P2 - P4$). It also puts the unselected peers as standby senders (for example $P5$), which are called upon to take over the load of degrading/disconnected senders during the streaming session. Each peer in GnuStream uses periodic probing to detect changes in the status of its suppliers. If a peer detects any problem, e.g. failure or bandwidth degradation of a supplier, it will migrate all or part of that supplier's streaming load to another supplier or a standby peer.

GnuStream uses two policies to distribute the load among senders: *even allocation* and *proportional allocation*. By using even allocation, the streaming load is distributed evenly among all senders. This policy is suitable for homogeneous en-

vironments. However, the proportional allocation is more flexible and suitable for dynamic and heterogeneous environments. In this policy the stream load is distributed among senders in proportion to the current capability of them.

The main drawback of GnuStream is its high network traffic. Depending on the degree of connectivity among peers, the flooding of queries can generate a lot of network traffic. Besides, objects located out of the search scope (defined by TTL in Gnutella lookup service) would not be found in the system.

PPLive

PPLive [61, 62] is one of the most well-known deployed IPTVs in the world. It is a chunk-driven peer-to-peer media streaming overlay, which uses gossip-based method to find suppliers. According to [48], "As of May 2006, PPLive had over 200 distinct on-line channels, a daily average of 400,000 aggregated users, and most of its channels had several thousands of users at their peaks". PPLive is not open-source, so little of its internal design mechanisms are known. PPLive streams live TV and video data through overlays of cooperative peers. It has multiple overlays, each belonging to one channels. Each channel streams either live TV programs or *episode-based* programs i.e. a fixed preset program set, which is repeated periodically.

A user can join one channel at a time. A new node first retrieves a list of channels from a *channel management server* via HTTP connection. Then it chooses one channel, i.e. a single overlay, and retrieve a small set of member nodes, namely *partner list*, from *membership servers* via UDP connection. Next it uses this partner list to learn about other candidate partners. It can also update its partner list by contacting the membership servers periodically. Partners are of two kinds: *candidates* and *real* partners. The real partners are used for exchanging video streams, while the candidate partners are used to replace real partners who have become unresponsive. A video stream is exchanged between each node and its real partners via TCP connections.

The main drawback of PPLive is its start-up delay. When PPLive first starts, it requires some time to search for peers and then tries to download data from active peers. The observed delay is around $20-30$ seconds for popular channels, and up to 2 minutes for less popular channels. The other problem of PPlive is the possibility of transmitting duplicate media contents, which wastes the network bandwidth [62].

2.2.4 Push-Pull Method Solutions

Pulsar

Pulsar [30] is a solution to live streaming, which combines the advantages of push-based approaches with the benefits of pull-based approach. The overlay consists of an unstructured and a structured part. Initially, a peer is assigned a random set of neighbors by a *network entry point*. Over time, a refinement process takes place as

peers learn about other peers from their neighbors and add them to their routing table depending on the latency measured to these peers. To ensure the connectivity of the overlay, it uses a DHT-like topology,that is the structured part of the overlay and specifies the path on which data is pushed to the peers. However, the protocol cannot guarantee that the push mechanism disseminate data to all peers, specially in a dynamic network, such as Internet.

Thus, a second mechanism is used where peers notify their neighbors about the corresponding sequence numbers they have newly received. When a peer does not receive a data block through push mechanism, it explicitly requests it from a neighbor which posses that block.

Prime

Prime [31] is a solution which uses a randomly connected and directed mesh overlay for data delivery. Prime uses the centralized method to find supplying peers. The central point is a bootstrapping node, who knows about all the participating peers in the system. New nodes contact this bootstrap node to learn about some other peers. On the other hand, all the connection for data transition are initiated by the receiving peer, who tries to maintain a sufficient number of suppliers that can collectively fill its incoming access link bandwidth. In order to maximize the bandwidth utilization of both incoming and outgoing links of all participating peers, authors suggest that all connections in the overlay have roughly the same ratio of bandwidth to node degree, called *bandwidth per flow*.

Data is delivered in two phases: *diffusion phase* and *swarming phase*. In diffusion phase, media source pushes distinct data segments to its children and so each subtree is provided with a set of distinct packets. Each node requests new data packets from one of its parents, which is the closest one to the media source in terms of hop counts. So after a certain interval, all the nodes will receive a new set of data packets. In the swarming phase, nodes from different subtrees contact each other to find the missing data packets. in both phases, each node runs a packet scheduling algorithm to determine what packets to request, from which parent and in what order.

Prime has a few shortcomings. First, the argument on forcing a certain bandwidth per flow for all the connection is not well persuaded; even the simulation results are imprecise (consisting very large confidence intervals) and so not supportive. Second, although the authors emphasize the importance of avoiding content bottleneck, in which peers can not find missing data blocks in their neighborhood, no way is provided to avoid such situation and the solution only counts on the randomness of mesh construction. If such bottleneck happens nodes have to wait long in order to find their required data units after a few swarming phases, when that data becomes available in their neighborhood. Briefly, there is no guarantee for a reasonable level of streaming quality. Third, the behavior of system in presence of churn is not discussed at all, and there is no such consideration in the algorithm. Finally, the

evaluation part comes with inappropriate assumptions that weaken the evaluation outcomes, e.g. using MDC with 10 descriptions, which incurs a remarkable data redundancy, and peers with high bandwidth capacities, which is not valid in real world.

mTreeBone

mTreeBone [64] is a peer-to-peer media streaming algorithm, which combines push and pull model for data delivery. It constructs a tree over stable nodes in the network, which are the nodes having an age over a certain threshold. Since nodes, who have stayed longer in the overlay, are less likely to leave or fail, this tree is expected to have a rather low churn. Nodes that are not members of this tree, connect to it as a leaf and cannot adopt any child.

Data is delivered by two means. At first it is pushed through the tree and each node has a *tree-push pointer* which indicates the latest data block it has received in the push phase. Nodes also maintain a partial view of the network, using a gossiping technique, called *SCAMP* [15]. They establish connection with other peers to find those data blocks they did not receive in the push phase. Likewise, they maintain a *mesh-pull pointer*, which indicates the latest data blocks they received in the pull phase. In order to prevent redundant data transfers, mesh-pull pointer is always kept behind the tree-push pointer, which means request for pulling missing data are sent when there is no hope to receive them in push phase. Since the tree structure plays an important role in efficiency of data delivery, mTreeBone tries to incrementally improve the tree in terms of tree depth and nodes' latency to source.

mTreeBone has some drawbacks. First, although nodes have multiple suppliers, they do not really benefit from parallel data transfers which can result in low transmission delays and low playback latencies. Second, bandwidth of leaves often remains unutilized, which further results in unsuccessful joins, although the overlay may have bandwidth to serve new nodes. What is more, while it nicely considers the improvement of its main tree, it does not consider timing constraints of nodes, when preempts a node's position and offers it to another node with higher bandwidth. In such cases not only the disconnected node, but also the subtree below it may suffer from playback disruption or degraded quality.

Bullet

Bullet [6] is an algorithm that enables nodes to self-organize into a high bandwidth mesh overlay. Bullet tries to maximize the amount of bandwidth delivered to receivers. To find the supplying peers, it assumes an underlying tree is already constructed, using one of the existing mechanism of tree construction and maintenance. Nodes gather knowledge about their neighbors by exchanging random subsets of nodes using a special protocol, called *RanSub* [26]. RanSub works in two phases: *collect phase* and *distribute phase*. Collect phase starts when the leaves start to propagate *collect message* up the tree, leaving state at each node along the path to the root. Each node uses a *compact operation* to choose a random subset of nodes,

which are representatives of all members of the sub-tree rooted at that node. When the root receives all the collect messages from all its children it starts the distribute phase. In this phase each node sends a *distribute message*, which contains a random subset of nodes with disjoint data, to its children. There are different ways to construct this distribute set. For example one way is to use a *RanSub-non-descendants operation*, which sends a random subset of nodes to each child, excluding its descendants. Nodes then sends peering requests to each other, whenever they find out a peering would be beneficial. This is how a mesh structure is formed across the nodes.

For a node to estimate how beneficial a peering is, it needs to detect the resemblance between its own content and the other node's content. Each node maintains a *working set*, which contains the sequence numbers of packets which have been received successfully. By use of this working sets and by installing *Bloom Filter*[5], nodes acknowledge those data blocks they have received as well as those data blocks they are interested in. A sending peer, will install the Bloom Filter of its receiving nodes, and sends them those data blocks that they do not have. The receiving nodes will periodically update their Bloom Filter at the sending peers.

To summarize, first disjoint subsets of data blocks are pushed across the tree and then, running RanSub protocol, nodes choose their neighbors, and receive missing data blocks from multiple suppliers.

Although the general idea of Bullet is very nice, what is overlooked is how the underlying tree is constructed, and if the properties of this tree influence the efficiency of the algorithm, especially in the presence of churn. For example, if the tree turns to be a deep tree, then a large number of nodes, which are far from the source, would receive the pushed data with large delays. This delay will propagate further, since this data feeds the next phase, when nodes transfer data over the constructed mesh. Hence a lot of nodes may experience large latencies, which is not desirable in live media streaming. Moreover, the deadlines for acquiring data blocks is not considered in Bullet, which is again critical for live streaming applications. Finally, although the algorithm tries to reduce the number of redundant data packets, it can not avoid them. Because nodes do not explicitly request individual data blocks, it is likely that they receive the same blocks from their parent in the tree and another peer over the mesh. Since media streaming is a bandwidth intensive application, it is of great importance that an algorithm avoids such redundant data transfers, which wastes the network resources.

2.2.5 Other Solutions

SAAR

SAAR [37] is a shared control overlay for cooperative end-systems multicast. It has a control overlay, which is a multicast tree built on top of a DHT, e.g. Scribe [50] on top of Pastry [49], and is responsible for building and maintaining the data overlay.

The data overlay can be a single tree, multiple trees, or even a swarming (block based) overlay. It means that SAAR can be placed in any other solution categories except the push-pull method group.

All the nodes in a data overlay form a *group*, which is associated with a *group identifier* and a set of *state variables*. For each state variable it introduces an *aggregation operator* and two *propagation frequencies*: one for upward and the other for downward propagation. A state variable can be any aspect of a peer's status, for example forwarding capacity of a peer or its latency to media source. Each leaf in the multicast tree, i.e. the control overlay, calculates the value of its state variables and propagates them up to its parent in the control tree. Each interior node aggregates its own value with the values received from all its children, using the aggregation operator, and sends it upward, until it hits the root of the control tree. After that the aggregated value at the root is propagated down the tree to provide each node with an approximation of the values in the whole group. These upwards and downwards propagation are on going processes with predefined frequencies.

SAAR has a single *anycast* primitive, which takes the group identifier G, a constraint p, an objective function m and a traversal threshold t as its arguments, and returns a member of G whose state variables satisfy p and maximize m. Each anycast conduct a depth first search over the control tree and prune the subtrees whose aggregated value is less than the best known value up to that point. In order to build a certain kind of data overlay it is enough to define the set of state variables and their corresponding aggregated operators, as well as the appropriate constraint and objective function for peer selection.

What is interesting in SAAR is that it can be easily configured to serve different objectives and no huge effort in the implementation is required if the priorities or the objectives change. What is more, a single control overlay can be used to support several data overlays, which provide different data streams. This can be very useful in an IPTV with several channels, because the channel switching would be very quick.

The first drawback of SAAR is that due to network and peers dynamism in the Internet, having an ongoing optimization procedure, for improving the data overlay structure, seems inevitable. Such optimization is not well discussed in SAAR, which means although for each decision it tries to maximize the objective function of a single peer, there is no consideration that over time the global utility of the data overlay would be maintained. Also having a single anycast primitive would narrow the way to build overlays with mixed approaches, such as push-pull overlays, which are shown to be successful for media streaming.

2.2.6 Related Work at a Glimpse

Figure 2.9 depicts all the reviwed solutions in the framework we introduce in Section 2.1. The general trend shows a moment towards using pull methods. Though more recently, algorithms, which are combining the advantages of the two methods are going more interest.

Finding supplying peers \ Data Delivery	Push method (Single tree)	Push method (Multiple trees)	Pull method	Push-Pull method
Centralized method	DirectStream (2006)			Prime (2007) mTreeBone (2007)
Hierarchical method	ZigZag (2003)			mTreeBone (2007)
DHT-based method	SAAR (2007)	SAAR (2007) SplitStream (2003)	SAAR (2007)	Pulsar (2007) mTreeBone (2007)
Controlled flooding method			GnuStream (2003)	
Gossip-based method		Orchard (2006) ChunkySpread (2006)	CoolStreaming (2005) PULSE (2006) ChainSaw (2005) PPLive (2004)	Bullet (2003)

Figure 2.9: How the studied solution find the supplying peers and how they deliver data

Chapter 3

ForestCast

In this Chapter, we would like to introduce an algorithm to stream a large scale media over a peer-to-peer overlay. This problem is challenging as one typically needs to:

- Maximize the *total utilization of upload bandwidth*
- Minimize the *playback latency*
- Minimize the *start-up delay*
- Maximize the *received quality* by the nodes

The first requirement stresses that the solution needs to ensure the actual available upload bandwidth at each node should be utilized as much as possible. Any solution must, therefore, adapt to the given upload bandwidth of the individual nodes. This implies that even nodes with small upload bandwidth should be utilized. The second requirement puts focus on latencies between the nodes and media source. It also implies that the depth of the multicast trees should be shallow to minimize latencies. The third requirement emphasize the importance of fast startup, which is specially important when users wish to frequently switch between different channels of an IPTV. Finally, the solution should provide a high quality media stream to the clients. The quality does not only refer to the media rate the clients receive, but the continuity of the playback should also be taken into account.

Note that some of the above mentioned goals are conflicting. For example, a low startup delay can be achieved by making each peer play the media as soon as it receives the first data segment. In such solution, when nodes do not buffer the media at all, a node will immediately suffer a quality loss or disruption, the moment it fails to receive a small piece of data; because there would be no time to makeup for that data loss. Hence, there is a trade-off between starup delay and quality/continuity. Moreover, sometimes we have to sacrifice a few nodes to improve the overall structure of the overlay. For example suppose a node with a very poor upload bandwidth

connects directly to the media server and prevents other nodes to join the system. In that case we desire to push this node down the trees and replace it with a node with higher bandwidth capacity, even if it costs the weak node to loose its quality or continuity. Nevertheless, before making such decisions we need to see how beneficial the change is.

More generally, the problem at hand is an optimization problem with a number of objectives, which are sometimes conflicting. Solving this problem is the process of simultaneously optimizing these conflicting objectives subject to certain constraints. *ForestCast* is an algorithm that heuristically moves towards these objectives, and enable providers to discover a solution which best fits their priorities. Depending on the heuristics one may choose, overlays with different properties are shaped up. In this section we introduce the main algorithm along with some of the heuristics, which can be applied. Later, in the evaluation section, we will see how these heuristics affect the resulting overlay.

3.1 Solution

ForestCast is a peer-to-peer media streaming system that uses multiple trees to deliver data, and a centralized directory to find supplying peers. There are three types of node in ForestCast: (i) *peers*, which are the nodes that download and/or upload the stream, (ii) *media server*, which has the media to be streamed, and (iii) *central server*, which has complete information about every peer and the overlay network and constructs the trees. Without losing of generality, we assume the use of MDC to split the media into substreams of equal size and equal value, namely *stripes*. These stripes are further fragmented into equal size *segments*, which are numbered by the media server sequentially. Each stripe is delivered through a separate tree, rooted at the media server. In this section we will see how this data delivery overlay is built and maintained by the central server. To start, we need to agree on some terms and definitions we are going to use hereafter in the text.

- *nodeId*: a unique id, given to each node for further references.
- *stripeId*: a unique id, given to each stripe.
- *startup segment*: the first segment that providers of a peer send to it.
- *head of buffer*: the largest segment number a node has in its buffer.
- *tail of buffer*: the smallest segment number a node has in its buffer.
- *playback point*: the segment number a node is playing.
- *media point*: the segment number the media server is playing/broadcasting.

- *head-to-play distance*: the difference between head of buffer and playback point of a node.
- *playback latency*: the difference between the playback point of a node and the media point.
- *open node*: a node whose available upload bandwidth is enough for sending at least one stripe.
- *peer profile*: a set of information each peer periodically sends to the central server. This profile includes the following attributes:
  - playback point
  - stripeIds the peer is receiving
  - nodeId of the parent for each stripe
  - link latency to each parent
  - head of buffer for each stripe
  - tail of buffer for each stripe

3.1.1 Join Procedure

Whenever a peer enters the system, it contacts the central server and provides it with its own upload/download bandwidth. The central server is in charge of finding appropriate parents for this node. How this decision is made, will significantly affect the efficiency of the system. The decision will be based on the existing trees and the properties of the joining peer e.g. its available upload bandwidth. The server then instructs the selected parents to start sending stream to the joining peer. Algorithm 1 presents the join procedure, that is the steps the central server goes through upon receiving a join request. Briefly, the main steps are:

- Find a list of open nodes in each stripe tree,
- Assign a priority to the selected open nodes and select the most appropriate parent for each stripe,
- Decide about a startup segment,
- Decide about the head-to-play distance of the peer,
- Instruct the selected parents to start sending data to the new peer.

Note that, to accomplish the above mentioned tasks, several techniques may be used by the server. Here we introduce a few heuristics for the methods in Algorithm 1. Later in Section 5, we will explain these heuristics in more details and investigate their effects on the overlay properties.

Algorithm 1 Join Procedure

```
1: upon event ⟨JOIN-REQUEST | nodeId⟩
2:     openNodeList ← findOpenNodeList(node, N_nodeId)
3:     parentsStripes ← selectParents(openNodeList, N_nodeId, 0)
4:     segment ← decideOnStartupSegment(parentsStripes)
5:     headToPlay ← decideOnHeadToPlay()
6:     for all ⟨parent, stripe⟩ in parentsStripes do
7:         send ⟨START-SEND-TO | node, stripe, segment⟩ to parent
8:     end for
9:     send ⟨JOIN-RESULT | headToPlay⟩ to node
10: end event
```

findOpenNodeList(*node*, N): It collects N open nodes in each stripe tree, and returns list of selected open nodes as output. Algorithm 2 and Algorithm 3 show the associated pseudocode. The important question in $findOpenNodeList$ is that from where it starts picking the nodes and in what order. Heuristic 1 suggests two possible answers to this question. Having the requesting node as an input parameter, the server is be able to choose different strategies for the nodes with different properties.

▷ **Heuristic 1.** A simple heuristic is to start from top of the trees where the media server is located and traverse trees in Breadth First Order (BFS). This approach places the peers as close as possible to the media server regardless of their bandwidth capacities. However, we can apply another approach, which performs a Depth First Order (DFS) search instead of BFS, for peers with no upload capacity. In Section 5.2.1, we will show how these two policies influence the overlay.

Algorithm 2 Find Open Nodes

```
1: procedure findOpenNodeList ⟨node, N⟩
2:     for all stripeTrees do
3:         openNodes ← findOpenNodeListInStripe(node, stripeTree, N)
4:         openNodeList.add(openNodes)
5:     end for
6:     return openNodeList
7: end procedure
```

selectParents(*openNodesList*, N, i): This method selects a parents for each stripe from the open node list. N is the number of stripes and the third parameter, i, implies that this peer is willing to receive any segment greater than i. This latter parameter is handy when a node, which is already receiving some of the stripes, wants to join other stripe trees, e.g. due to its parent departing. Apparently it matters for this node to receive data segments from a certain point on. Algorithm

Algorithm 3 Find Open Nodes Per Stripe

```
1: procedure findOpenNodeListInStripe ⟨node, stripeTree, N⟩
2:     peer ← root of stripeTree
3:     while openNodes.size < N and there is a peer to examine do
4:         if peer is open and peer is not leaving then
5:             openNodes.add(peer, stripeTree)
6:         end if
7:         peer ← get next node in a BFS order not including the node itself and
   its descendants
8:     end while
9:     return openNodes
10: end procedure
```

4 shows the related pseudocode. There are two important heuristics one can apply while selecting parents, and both are closely related. First one is about to prioritize the open nodes (Heuristic 2) and second one is about what makes a node more appropriate parent. (Heuristic 3).

▷ **Heuristic 2.** There are many factors that can be taken into account for assigning a priority value to the peers in the open node list, e.g. a peer's available upload bandwidth, or the number of its children. In Algorithm 5 we define our *priority function* to be $\frac{b^{\alpha}}{f_i^{\beta} \cdot l_i^{\gamma}}$, where b is the available upload bandwidth of the peer, f_i is the peer's *fanout*, i.e. the number of its children, in stripe tree i, and l_i is the latency of the peer to the media source in the respective tree. Moreover α, β and γ are configurable parameters, which define the importance of the mentioned factors in the function. Peers with higher values have more chance to be selected as a parent. By considering the fanout of nodes, we persuade an even distribution of different stripes in the overlay and give more priority to the peers who have sent fewer stripes in each stripe tree. Note that, an individual peer may be given different priorities in different trees and will favor forwarding a stripe, which it has forwarded fewer times. In other words, we avoid a situation where some of the stripes become rare in the system. In Section 5.2.2 we will see how the overlay is influenced when we consider fanout of nodes and when we do not.

▷ **Heuristic 3.** After we assign a value to each node using the previous heuristic, the next step is to select one parent for each stripe. One solution is to select the best N nodes from the list, where N is the number of stripes. But by this naive choice we may choose a node repeatedly as a parent in different trees. This makes the child very vulnerable to the failure of the chosen parent. Another solution to select N *distinct* top nodes. In Section 5.2.2 we will compare the outcome of using these two heuristics. Algorithm 4 shows a sample implementation for parent selection.

Algorithm 4 Select Parents

```
1: procedure selectParents ⟨openNodeList, N, segment⟩
2:     for all ⟨peer, stripe⟩ in openNodeList do
3:         assignValue(⟨peer, stripe⟩)
4:     end for
5:     while selectedParentsStripes.size < N andopenNodeList is not empty do
6:         ⟨parent, stripe⟩ ← pickOneParent(openNodeList)
7:         selectedParentsStripes.add(⟨parent, stripe⟩)
8:         node.addParent(⟨parent, stripe⟩)
9:         openNodeList.excludeEnteriesFor(stripe)
10:        update the value of peer in openNodeList for other stripes
11:    end while
12:    return selectedParentsStripes
13: end procedure
```

Algorithm 5 Assign Priority To Stripes

```
1: procedure assignValue ⟨peer, stripe⟩
2:     b ← peer.getfreeUploadBW()
3:     f ← peer.getFanout(stripe)
4:     l ← peer.getTotalLatency(stripe)
5:     ⟨peer, stripe⟩.value ← b^α/(f^β × l^γ)
6: end procedure
```

decideOnStartupSegment($parentsStripes$**)**: After selecting one parent for each stripe, we should decide on the first segment number, i.e. the *startup segment*, to be forwarded to the joining node. This segment number determines a specific point in the media that the joining peer will start to play. Note that selected parents have different segments at their buffers. Algorithm 6 shows the pseudocode of this method and Heuristic 4 suggests a few ideas that can be considered while making this decision.

▷ **Heuristic 4.** Startup segment can be any segment that all the parents have in common. For instance it can be the minimum of head of buffer of all parents. By such selection the joining node would be able to watch the media with least possible playback latency. Another approach is to choose the segment to be the maximum of tail of buffers of all parents, which result in more tolerance to failure of the predecessors. In other words there is a trade-off between playback latency and failure tolerance. The startup segment can be any point between the two previous values. As another example, Algorithm 6 shows a pseudocode of this method, which considers the range of segments that all the parents have in their buffer and chooses the middle segment. In Section 5.2.3 we will show how selecting different startup segments will affects the quality of the received media by peers.

Algorithm 6 Decide On The Startup Segment

1: **procedure decideOnStartupSegment** $\langle selectParentsStripes \rangle$
2: **for all** $(parent, stripe)$**in** $selectedParentsStripes$ **do**
3: $head \leftarrow parent$.headOfBuffer($stripe$)
4: $headOfBuffers$.add($head$)
5: $tail \leftarrow parent$.teailOfBuffer($stripe$)
6: $tailOfBuffers$.add($tail$)
7: **end for**
8: $latestSegment \leftarrow headOfBuffers$.min()
9: $earliestSegment \leftarrow tailOfBuffers$.max()
10: $selectedSegment \leftarrow (latestSegment + earliestSegment)/2$
11: **return** $selectedSegment$
12: **end procedure**

decideOnHeadToPlay(*segment***)**: Finally, the server should decide on the head-to-play distance of the node, i.e. how long a peer should wait before it starts to playback the media. Heuristic 5 suggests some relevant heuristics. If a peer buffers more data before starts to playback, it would be more resilient to the failure of its parents; because it would have more time to make up for the possible data loss. But the more this buffering delay is, the higher startup delay and playback latency the node will have.

▷ **Heuristic 5.** A simple solution is to have the same buffering delay for all nodes. The question, then, is how long this buffering delay is. However in a more complicated solution, the position of node in different trees may be considered, for example a node, which it directly connects to the media server does not need to wait very long to buffer data. A different approach is to leave this decision to be made by the node itself. For example the node can estimate the latency to its different parents and waits at least for the time required to receive a segment from the parent with the most latency. In Section 5.2.4 we use the first technique with different values for buffering delay.

3.1.2 Leave Procedure

When a node wants to leave the system, it sends a leave request to the central server, and stays in the system until it receives the central server's grant. Meanwhile the server finds substitute parents for the children of the leaving node. Algorithm 7 presents the leave procedure. Main steps of this algorithm are:

- For each child of the leaving node, find the last segment it has already received from the leaving parent.

- For each child, find a substitute parent, which have the rest of the segments soon enough to be able to forward them to the child, without the child missing the segments' playback deadline.
- If a new parent is found, instruct the new parent to start sending data from the last segment on to the child, and also inform the leaving node to stop sending data to that child. Otherwise, retry after a short interval.
- If a substitute parent could be found for all the children, instruct the parents of the leaving node to stop sending data to it.
- Finally, grant the requesting node to leave the system.

3.1.3 Failure Handling

In ForestCast peers are responsible for detecting the failure of their parents. If they detect a parent failure, they report it to the central server immediately. We argue that in this application we can assume a perfect failure detector. Because live streaming inherently has certain timing constraints. These constraints can be used to define an upper bound for the acceptable transmission delay in the overlay, and this upper bound can be further used to introduce a perfect failure detector. But for the sake of simplicity, now let us assume we only have an eventually failure detector. We also assume that server is also able to detect the failure of nodes.

Failure handling is not very different from the leave procedure, except that after rejoin, children of the failed node may be missing some of the segments. More exactly a node may rejoin a new parent which does not have the subsequent segments of the last segment it has received; because unlike leave, if a rejoin is unsuccessful there would be no time for retry and the server will choose a substitute parent that causes the least data loss, that is the shortest gap between the last delivered segment and next accessible one. Note that, since all the decisions are made by the central server, the damage will be fixed very fast. Moreover, since we tried to connect nodes to distinct parents, at the first place, failure of a single parent causes its children to lose at most one stripe meanwhile and they would receive rest of the stripes from other nodes. Algorithm 8 shows how the central server handles failures. The main steps are:

- Instruct the parents of the failed peer to stop sending data to it.
- Find the last segment of the stripes that the failed node has sent to its children.
- If possible, find a substitute parent for each child such that it does not miss the next segment's deadline; otherwise find a parent which causes the least missing segments.
- Instruct the new parents to start sending data to their new children.

Algorithm 7 Leave Procedure

```
1: upon event ⟨LEAVE-REQUEST | node⟩
2:     successfulyRejoinedChild ← true
3:     successfulyRejoinedAllChildren ← true
4:     node.leaving ← true
5:     for all stripe do in node.getStripes()
6:         successfulyRejoinedChild ← true
7:         requiredSegment ← node.lastSegment
8:         for all child do in node.getChildren()
9:             segment ← child.lastSegment
10:            newParent ←findNewParent(child, stripe, segment + 1)
11:            if newParent ≠ null then
12:                send ⟨STOP-SEND-TO | stripe, segment − 1, child⟩ to node
13:                send ⟨START-SEND-TO | stripe, segment, child⟩ to newParent
14:                if requiredSegment < segment then
15:                    requiredSegment = segment
16:                end if
17:            else
18:                successfulyRejoinedChild ← false
19:                successfulyRejoinedAllChildren ← false
20:            end if
21:        end for
22:        if successfulyRejoinedChild = true then
23:            parent ← node.getParent()
24:            if parent ≠ null then
25:                send ⟨STOP-SEND-TO | stripe, requiredSegment, node⟩ to parent
26:            end if
27:            stripeTree.removeNode(node)
28:        end if
29:    end for
30:    if successfulyRejoinedAllChildren = true then
31:        send ⟨LEAVE-GRANTED | node⟩ to node
32:    else
33:        send ⟨LEAVE-REQUEST | node⟩ to self
34:    end if
35: end event
```

- If some of the children are still parent-less, retry after a short interval.

Algorithm 8 Failure Handling Procedure

```
1: upon event ⟨FAILURE-NOTIFICATION | node⟩
2:   for all stripe in node.getStripes() do
3:     parent ← node.getParent(stripe)
4:     send ⟨STOP-SEND-TO | node, stripe, 0⟩ to parent
5:     for all child in node.getChildren(stripe) do
6:       segment ← child.lastSegment
7:       newParent ← findNewParent(child, stripe, segment)
8:       if newParent = null then
9:         newParent ← findShortestGapParent(child, stripe, segment)
10:      end if
11:      if newParent = null then
12:        retry later to find new parent for this child
13:      else
14:        send ⟨START-SEND-TO | child, stripe, segment⟩ to newParent
15:      end if
16:    end for
17:    stripeTree.removeNode(node)
18:  end for
19: end event
```

3.2 Incremental Improvement

It is desirable to put nodes with higher bandwidth upper on the trees, because that would result in shallower trees and nodes would experience lower latencies averagely. Moreover, since the failure of a node up on a tree would influence a greater number of nodes than a node down the tree, we desire to put more stable nodes closer to the root. We propose to use a combination of these two properties to define the notion of being a more desirable node. There are several studies, which show that a node which has stayed long in the system is less likely to leave or fail [52, 58]. Hence, we define a node's *strength* in a tree to be the age of that node, *age*, times the number of its children in that tree, $fanout$, plus its available upload bandwidth, $freeBw$:

$$Strength(A_i) = age_A \cdot (fanout_{A_i} + freeBw_{A_i})$$

where i is the index of tree. Accordingly, we define a node A to be *stronger* than a node B in tree i, if

$$Strength(A_i) > Strength(B_i)$$

The above definition suggests that a strong node is a node that does not only have a high upload bandwidth, but also has a long uptime.

The main idea in this incremental improvement is to let the stronger nodes bubble up the trees, so that eventually we end up in a layout, in which node distances from the root of trees are in the order of their decreasing strength. More exactly, stronger nodes are placed closer to the media source, and nodes with petty upload bandwidth, e.g. free riders, or nodes that join and leave very quickly, e.g. nodes who just switch the channels to see whats on and then leave, are placed at the edges of the overlay.

The improvement, which is presented in Algorithm 9 and Algorithm 10, has two steps: (i) *promotion*, and (ii) *reconfiguration*. The first algorithm handles the *promotion requests*, while the second reconfigures the subtrees below the nodes who are *promoted* or *demoted*.

Nodes periodically acknowledge their strength to their children. Whenever a child finds out that it is stronger than its parent, it sends a promotion request to the central server. Upon receipt of this request, the central server goes through the following steps (Algorithm 9):

- Calculate the strength of the parent, the requesting child and its sibling, and choose the strongest node of all, p. Note that, checking the siblings will prevent several consecutive promotion requests for the same position in the trees.
- Instructs the grandparent of the p to switch its child from the p's parent to p.
- Instruct p to start sending data to its old parent.
- Send promoted and demoted messages to p and it's old parent, respectively.

Following the above steps, the position of the two nodes are swapped, but certain considerations should also be taken into account. Figure 3.1 shows a simple example. Suppose node r is stronger than q, so it is beneficial for the system if they change their position. If this replacement happens immediately, then r would receive its next segment from p instead of q and would benefit, because it gets closer to the source. On the other hand q is pushed down and would not receive any new segment for a while, because its new parent, r, does not have any new data to send to it yet. So q would experience a delay before receiving its next segment. This delay depends on how much the head of r is behind the head of q. For example in Figure 3.1, assume each segment number is the data required for one second playback of the media. So q has to wait for almost 11 seconds to receive its next required segment from r, while its head-to-play distance is only 5. Hence it either would be disrupted for 6 seconds, or even worse would miss that stripe for that point on. The former happens when q does not receive any other stripes, while in latter it can continue the playback by receiving other stripes. In this case, although some bandwidth is wasted in the network to deliver that stripe to q, but that stripe misses the deadline.

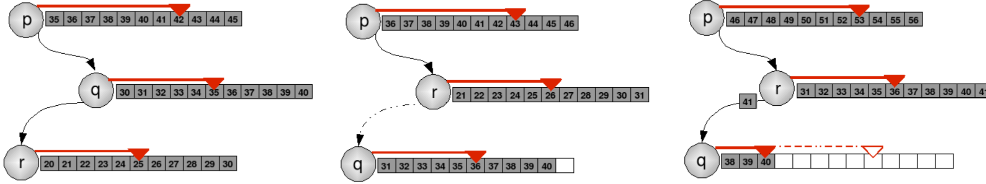

Figure 3.1: Buffers are affected when nodes change their position

A better solution is depicted in Figure 3.2. The central server asks q to continue forwarding data segments to r up to the segment it has received from p, but it instructs p to switch from its old child, q, to the new child, r, and starts sending data to r from the segment that it was about to send to q, and on. This means for a short period of time, r would receive data segments for the same stripe from two suppliers. The moment r receives the first segment from p, it can forward it to q. Hence q can receive its next required segment with a negligible delay, no matter how much r's head is behind. The only influence on q is a small decrease in its head-to-play distance.

Algorithm 9 shows how the central server reacts to a promotion request. Note that the central server changes the status of the strong child and weak parent to promoted and demoted respectively, in order to avoid several concurrent repositioning. It also informs the peers of their state to prevent them from sending new requests, when they are about to acquire new positions. What is more, the promoted node is expected to notify the the central server for "reconfiguring" the subtrees, whenever it is ready; i.e. when it does not receive any more data segments from its old parent. In such situation, the promoted node sends a *reconfiguration request* to the central server.

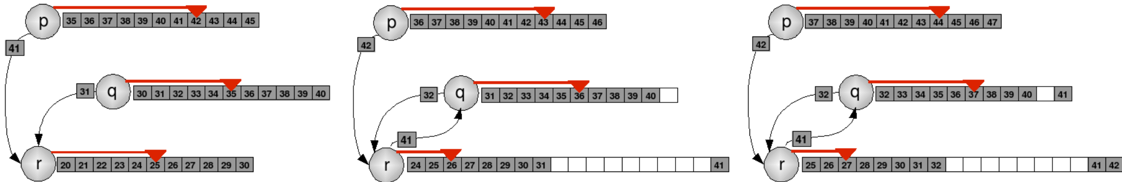

Figure 3.2: Lessen the buffer damage when nodes change their position

Algorithm 10 shows how the server handles a "reconfiguration request". The main object of reconfiguration is to rearrange the subtrees below the promoted child and the demoted parent to achieve a more balanced tree. Figure 3.3 shows an illustrating example. Suppose q and r in the left side picture are going to change their positions. In the first phase, e.i. promotion parents of r and q change to p and r,

respectively. In the reconfiguration phase, first of all, r adopts its old siblings, a and b, as its children. Since this node has a higher upload bandwidth than q, it is able to accept more children. So among its old children, it keeps as many children as it can support, preferably the strongest ones. The rest of its children are adopted by the old parent, q, who was pushed down the tree.

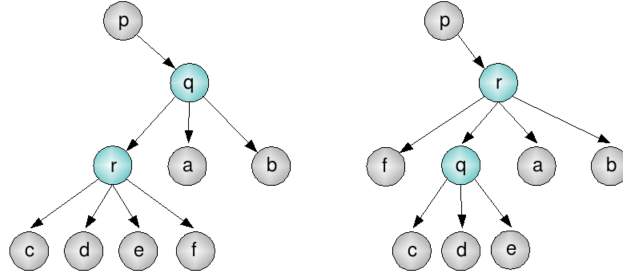

Figure 3.3: Arrange the subtrees when nodes change their position

In Chapter 5 we will investigate the influence of this improvement on the structure of trees.

Algorithm 9 Incremental Improvement - Request Promotion

1: **upon event** ⟨REQUEST-PROMOTION | *node*, *tree*⟩
2: *parent* ← *node*.getParent(*tree*)
3: **if** *parent*.promoted ≠ true **and** *parent*.demoted ≠ true **then**
4: *grandParent* ← *parent*.getParent(*tree*)
5: *child* ← *parent*.getStrongestChild(*tree*)
6: **if** *child*.promoted ≠ true **and** *child*.demoted ≠ true **then**
7: *segment* ← *parent*.lastSegment
8: **send** ⟨SWAP-CHILD | *tree*, *segment*, *parent*, *segment* + 1, *child*⟩ **to** *grandParent*
9: **send** ⟨STOP-SEND-TO | *tree*, *segment*, *child*⟩ **to** *parent*
10: **send** ⟨START-SEND-TO | *tree*, *segment* + 1, *parent*⟩ **to** *child*
11: *child*.promoted ← true
12: *parent*.demoted ← true
13: **send** ⟨PROMOTED | *tree*, *true*⟩ **to** *child*
14: **send** ⟨DEMOTED | *tree*, *true*⟩ **to** *parent*
15: **end if**
16: **end if**
17: **end event**

Algorithm 10 Incremental Improvement - Request Reconfiguration

```
1: upon event ⟨REQUEST-RECONFIGURATION | node, tree⟩
2:     demotedParent ← node.getDemotedParent(tree)
3:     child.promoted ← false
4:     parent.demoted ← false
5:     send ⟨PROMOTED | tree, false⟩ to child
6:     send ⟨DEMOTED | tree, false⟩ to parent
7:     siblingList ← node.getSiblingsFromDemotedParent()
8:     for all sibling in siblingList do
9:         if node is open then
10:            segment ← sibling.lastSegment
11:            send ⟨STOP-SEND-TO | tree, segment, sibling⟩ to demotedParent
12:            send ⟨START-SEND-TO | tree, segment + 1, sibling⟩ to node
13:            remove peer from siblingList
14:        end if
15:    end for
16:    if siblingList is not empty then
17:        for all siblings in siblingList do
18:            child ← node.getWeakestChild()
19:            childSegment ← child.lastSegment
20:            siblingSegment ← sibling.lastSegment
21:            send ⟨SWAP-CHILD | tree, childSegment, child, siblingSegment +
   1, sibling⟩ to node
22:            send ⟨SWAP-CHILD | tree, siblingSegment, sibling, childSegment +
   1, child⟩ to demotedParent
23:        end for
24:    end if
25: end event
```

3.3 Remarks

The central solution comes with many advantages. First of all the overhead of control messages is very low. That is because nodes do not have to disseminate any control messages to their neighbors and they do not make any effort to find the data. They just wait to be told what to do. Only in case of a parent failure, its children send a message to the central server and ask for a new parent. Upon receipt of such request, the server will repair the tree by assigning new parents to the orphaned nodes.

The central server can also reconfigure the trees periodically to better optimize the overall performance and nodes would follow server's orders to change their position and parents. What is more, this method not only makes things simpler and more efficient, but also nodes can join the system much faster and they will be placed in the best possible position. In other words the best decision is made in the

least possible time. Regarding the fact that most of the existing solutions suffer from the remarkably high start-up delay, the benefits of this approach are of great interest.

If the feasibility of this solution is an issue or it is susceptible to be a single point of failure then this server can be replaced by distributed servers, which can form a Distributed Hash Table, e.g. Distributed k-ary System (DKS) [17]. The advantage of this approach is that servers can be removed and added without the responsibility of trees changing too much.

Chapter 4

Simulator

Computer simulation is a prevalent method for evaluation of many types of systems. Due to the increasing complexity of systems in today's world, the simulation methods and simulators have gained remarkable research interest. Specially computer networks simulation brings about many challenging research problems due to the complexity, size, and heterogeneity of typical networks.

In the next section we explain some of these challenges and requirements in more details. Then we present our own solution, *SICSSIM-B*, and describe how it works. Finally we review some of the existing peer-to-peer simulators.

4.1 Challenges and Approaches

Discrete-event simulation is a widespread technique for computer network simulation. In this approach, operation of a system is represented as a chronological sequence of events. Each event occurs at an instant in time and marks a change of the state in the system. Such simulators have a queue known as *Future Event List (FEL)*, which includes the events that are to be processed in the order of their start time.

A traditional model to simulate network traffic is *packet-level* simulation which employs packet by packet modeling of network activities [25, 66]. In this model for each packet departure or arrival one event will be generated (Figure 4.1). In fact, packet-level simulation tries to be as close as possible to a real network. Considering the effect of any single packet makes this approach accurate, but heavy weighted. If the size of network grows, huge number of events will be generated, which results in costly processing time and significant complexity. Therefore, packet-level simulation can not scale well. That implies for peer-to-peer systems, which are generally aimed to be scalable, packet-level simulation is not a good choice.

Another model, which simplifies simulating network traffic is, *flow-level* model,

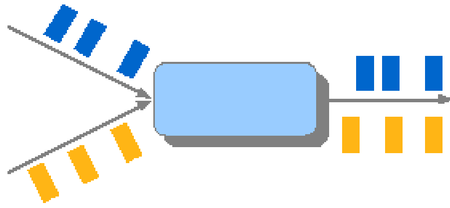

Figure 4.1: Packet level simulation

also known as *fluid-based* model [28]. In flow-level modeling the underlying layers of network are abstracted away and the events are generated only when the rate of flows change (Figure 4.2). This abstraction, enables simulations at large scale, but at the cost of losing accuracy. This is because the effects of underlying layers are ignored.

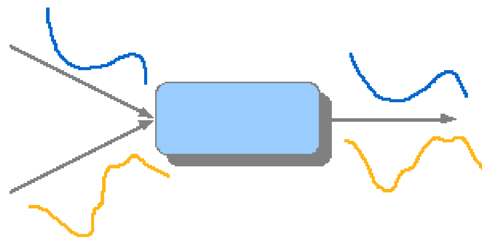

Figure 4.2: Fluid based simulation

In a static network or when flow rate changes occasionally, flow-level model would achieve a high performance. In other word, compared to packet-level simulation, the largest performance gains are achieved with small networks and cases where the number of packets represented is much larger than the number of rate changes. For larger networks, a property described as the *ripple effect* [28, 29] can reduce the performance advantage in the flow-level simulator. The ripple effect describes the situation where the propagation of rate changes leads to rate changes in other flows which then need to be propagated again [25]. Hence, this kind of modeling sometimes results in overestimating the performance of some of the algorithms. Nevertheless, [14] argues that flow-level modeling is a reasonable approach and shows that the achieved results are not dramatically different.

While these models come with their advantages and disadvantages, making an appropriate choice is important for producing valid evaluation results. This choice highly depends on the characteristics of the application which is being evaluated and the objectives of evaluating it. That would require a systematic analysis to identify the critical variables and to incorporate them in the model, while eliminating all

irrelevant distractors.

We desire to analyze the performance of a peer-to-peer system for live media streaming. Such an application involves large scale data transfer and it is very susceptible to link capacities. Also it inherently has stiff timing constraints which makes it sensitive to link latencies and packet drops in physical network. So our simulator should be able to accommodate these characteristics. It is also important to notice that connections in overlay network do not necessarily conform to the connections in the physical network. This mismatch may unexpectedly influence the performance of peer-to-peer systems. For example suppose a number of nodes are connected together as shown in figure 4.3 (a). At the same time the physical connection between them is shown in 4.3 (b). Apparently any traffic between A and B will go through C and D, use their download and upload bandwidth and loads the link connecting them; but this can not be observed in figure 4.3 (a).

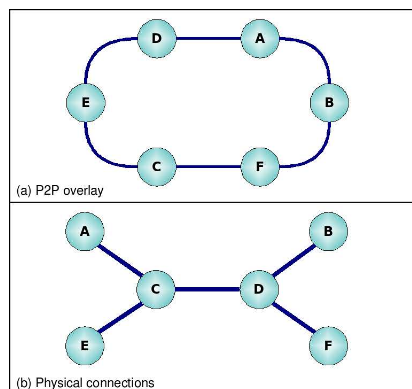

Figure 4.3: Nodes connection in a peer-to-peer overlay and in physical network

4.2 SICSSIM-B

Since our application involves high bandwidth data transfer, we choose a flow-level model to be able to conduct experiments at large scale. To deal with inaccuracy we improve this model by incorporating some of the effects of the underlying network, which may influence the performance of our algorithm. In other words, although we abstract away the functionality of the underlying layers, we do not ignore them completely and will try to reflect some of their relevant effects on our measurements. For example if a message passing scenario is simulated by putting a send and a receive event in the future event list, then for simulating packet loss it is enough to randomly ignore some of the receive events in the list. Apparently to have a realistic model, an appropriate random pattern should be applied.

We assume topology consists of *peer*, *link*, and a *core network*. As depicted in Figure 4.4, each peer connects to the core network through a link with a specified capacity and latency.

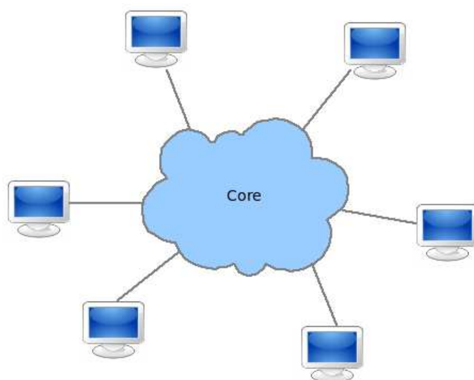

Figure 4.4: SICSSIM-B overall structure

To model bandwidth we draw two random values, from any desirable distribution, for incoming and outgoing bandwidth of each node. We then use a matrix, namely *bandwidth matrix*, to monitor data transfer rate between nodes. Each row in the bandwidth matrix is associated to one peer in the system and shows the upload rate of that peer to the other peers at an instant of time. Apparently, sum of the cells in each row should not be greater than the total outgoing bandwidth of that peer. Likewise, each column represents the downloading status of peers. Each cell shows the download rate from another peer and similarly sum of the values in a column should not be greater than the total incoming bandwidth of the associated peer. Figure 4.5 shows a sample snapshot of the bandwidth matrix. For instance node A is uploading $56Kb$ of data whiles downloading a total amount of $384Kb$ per second. It is worth mentioning that, we only constrain sum of the upload/download rates of each node. What is missing is the fact that maximum upload/download rate between two individual nodes is not only constrained to the endpoints of a connection, but also to the other links on the route from one to the other. More exactly the minimum link capacity on this route define the maximum transfer rate between two nodes.

To model link latency, we calculate the latency between each pair of nodes as follows. We add the latency of the two links, which connect the peers to the core, to the latency of the core. The core latency itself consists of two parts: a value, which is uniquely generated for each pair of nodes, and a random value, which represents the fluctuation in the network every time a data transfer takes place. Note that for each of these four elements of calculated latency, an appropriate distribution should be selected.

	A	B	C	D	Total Upload
A		56	0	0	56
B	128		1000	56	2000
C	256	1500		0	5000
D	0	0	56		56
Total Download	512	2000	10000	56	

Figure 4.5: Bandwidth matrix shows the download/uploads status of peers in Kbps

In SICSSIM-B we differentiate between the *control messages* and *data messages*. Data messages are the actual that data and can be of any type, e.g. video, audio, text, images, etc. For the sake of simplicity, we do not transfer real data in the simulator. We just assume there is a flow of data from node to node with a rate, specified in the bandwidth matrix, and a latency, calculated for those nodes. Although there is no real data, any change in the bandwidth matrices represents a change in a data flow. On the other hand, the control messages are considered to have a very small size which would use zero bandwidth; but they will be queued in a list, i.e. *Future Event List* or *FEL*, which contains all the events of the system. Then an *event scheduler* let the simulator handle the control messages in the order of their increasing start time. The simulation loop proceeds by selecting the next event from FEL, executing it and inserting newly generated events in the queue.

As mentioned before, in flow-level modeling we ignore some of the properties of the underlying layers of network. One of these properties is *link congestion*. Whenever a node joins the system, it is assigned a total incoming/outgoing bandwidth. This incoming/outgoing bandwidth may change during the simulation, because a node may also use its bandwidth for other tasks, which are irrelevant to our overlay, so the total download/upload bandwidth of nodes are subject to change, and these changes may affect the data transfer rate between peers.

We model congestion at nodes' links as well as the network on the route from one node to the others. We use a strategy similar to *slow-start* [2] congestion control, which is used by TCP. A data transmission between two nodes starts with a predefined rate. On the receiver side, the node continuously checks the download rate while considers its available incoming bandwidth. If it has enough free incoming bandwidth, it sends a message to sender and asks for a higher upload rate. On the other hand, if sum of the download rates becomes greater than its incoming bandwidth, it sends a congestion message to at least one of its uploaders. Upon receiving such message, the sender decreases the upload rate. To model network congestion, the core network (Figure 4.4) randomly sends congestion messages to an uploaders

nodes in system and asks a decrease in the upload rate.

Note that, congestion modeling in SICSSIM-B is a configuration parameter, which can be enabled or disabled.

4.3 Existing Solutions

In this section, we first look at some criteria to assess peer-to-peer simulators and then we review nine existing simulators.

4.3.1 Criteria

Stephen Naicken et. al. [35, 36] suggest that a simulator can be assessed by a number of criteria, which are as follows.

- **Simulator Architecture:** This is a key issue, which has to do design and functionality of the simulator, and the features it provides, e.g. whether it is a discrete event simulator; whether it is a flow-level simulator or packet-level; whether it supports structured or unstructured overlay simulation, or both; and how these are implemented. Also these criteria include aspects of how node's behaviour is simulated, for example whether churn can be specified or what levels of churn is simulated.
- **Underlying Network Simulation:** There are different approaches to model the underlying network. The point here is that which properties of the underlying layers are simulated by the simulator. For example whether *cross-traffic*[1] is supported by simulator; or whether it can simulate link latency or bandwidth, and how realistic the underlying topology is.
- **Scalability:** One the main capabilities expected from peer-to-peer simulators is scalability. So it is important to consider how scalable a simulator is, i.e. how many nodes it can support, and how it performs for the large network size.
- **Statistics:** Outputs and results of simulators is the next issue to be considered. To have a good statistical analysis, the results should be expressive, easy to manipulate, and not to mention reproducible.
- **Usability:** This criterion shows how easy the simulator is to learn and to use. For example, whether the simulator has a clean API that helps users to understand the protocols and use them easily; whether it supports any script language to define scenarios, or if any manual or technical document is provided along with the simulator.

[1] Other traffic on network, which is not related to the overlay.

4.3.2 Simulators Review

It this section we review some of the existing simulators.

- **DHTSim:** *DHTSim* [13] is a discrete event simulator for structured overlay networks. Since it is intended for teaching DHT protocols, it has a clear and straightforward API, however it is not well documented. This simulator does not support distributed simulation and node failure. Also its does not have the capability of extracting statistics. DHTSim is implemented in Java, and scenarios are specified by using script files.

- **P2PSim:** *P2PSim* [41] is a discrete event simulator for structured overlays, which works in packet level. It is packaged with implementations of six DHT protocols: Chord [54], Accordion [27], Koorde [24], Kelips [20], Tapestry [69] and Kademlia [33]. It supports a wide range of underlying network topologies, but does not support distributed simulation, cross-traffic, and massive fluctuations of bandwidth. It provides a standard tool to help generating graphs easily, and to summarize all log files into a single statistics file. P2PSim is implemented in C++ and has a poor documentation. Its scalability has been tested for up to $3,000$ nodes.

- **Overlay Weaver:** *Overlay Weaver* [40] is a discrete event simulator for structured overlays only. It contains implementation of Chord, Kademlia, Tapestry, Koorde, and Pastry [49]. It does not simulate underlying network, but supports distributed simulation. Overlay weaver provides users with a good interface with a few command line tools. Users can define scenarios as a simple script file. However it does not have any facility to gather statistics. It is implemented in Java with clean API and easy to read source code, but its interface is not well documented. Moreover it supports up to $4,000$ nodes.

- **PlanetSim:** *PlanetSim* [47, 16] is a discrete event overlay network simulator that supports structured and unstructured overlays. It contains the implementation of Chord and Symphony [32]. The underlying network layer can be modelled as simple random or circular networks, but it does not consider latency, cross traffic and bandwidth. PlanetSim uses *Common API (CAPI)* [12], which helps the users to develop their own protocols in a layered model, where services and applications can be built one on top of another. This feature provides user with the capability to simulate an application layer service over different overlay networks. PlanetSim does not have any mechanism to gather statistics, but has a visualizer. It is implemented in Java and a large amount of documentation is available at its website [47]. The scalability of PlanetSim has been tested for up to $100,000$ nodes.

- **PeerSim:** *PeerSim* [45] is an event based simulator for peer-to-peer overlay networks, which supports both structured and unstructured networks. It is designed for epidemic protocols, with high scalability and dynamism. Hence, it

avoids the underlying network parameters, such as latency. PeerSim supports two models of simulation: *cycle-based* and *event-based* models. In the former, the simulator chooses all the nodes at random and invokes each node protocol in turn at each cycle. In the latter, a set of messages or events are scheduled in time and the simulator invokes each node protocol according to the message delivery time order. PeerSim does not support distributed simulation. It does not have any debugging facilities, and extra components should be implemented to gather statistics. It is written in Java and users can configure it using a plain text file. It has a good documentation for its cycle-based simulation, but not for the event-based model. As [36] claims, PeerSim supports up to 10^6 nodes in the cycle-based model.

- **GPS:** *GPS* [66] is a discrete event simulator that supports both structured and unstructured overlays. It contains an implementation of BitTorrent protocol [10]. GPS partially models the underlying network topology. It does not model each packet, but simulates bandwidth and delay for each link and provides different flow level models. It is implemented in Java and its API is poorly documented. The scalability of GPS is not measured, but it has already been used for up to 1054 nodes.

- **Neurogrid:** *Neurogrid* [39] is a discrete event simulator that can simulate both structured or unstructured overlays. This simulator was originally designed for comparing Neurogrid protocol, Freenet [9] and Gnutella [56] protocols. It does not simulate the underlying network. It has facility to gather statistics for pre-determined variables, but for adding new variables, the code should be modified. Neurogrid is implemented in Java and has an extensive documentation on its web. [36] claims that Neurogrid supports up to $300,000$ nodes.

- **Query-Cycle Simulator:** The *Query-Cycle Simulator* [51], is a peer-to-peer file sharing network simulator for unstructured overlay networks. The simulation process consists of a number of query-cycles. At each cycle, if a peer is seeking a file, it sends a query to other peers. Whenever a peer receives a request for a file it poses, it responds to the requesting peer. As the peers receive the response from other peers, they choose one peer and download a file from it. The cycle completes when all querying peers download a file. This simulator is implemented in Java and its API document is available.

- **Narses:** *Narses* [38] is a discrete event, flow level network simulator. It allows to model the network at different levels of accuracy and speed. It simulates the underlying network in four different model, from the least accurate and fastest to the most accurate and slowest. It computes the available bandwidth between two peers based on end node bandwidth and does not consider the network and protocol effect. To gather statistics the source code should be

modified. Narses is written in Java, and no document is available for it. It has been tested for up to 600 nodes.

Figure 4.6 shows a summarized comparison of surveyed simulators with respect to the defined criteria and Figure 4.7 shows some complementary information about them.

Simulator	Architecture	Usability	Scalability	Statistics	Underlying Network
DHTSim	Discrete-event for structured overlay networks	Simple API, but lack of documentations	---	Not possible	---
P2PSim	Discrete-event and packet level for structured overlay networks	Poor documentation	3,000 nodes	Limited tool to generate graphs and summarize log files	Supports a wide range of underlying networks
Overlay Weaver	Discrete-event for structured overlay networks	Good API documentation, but its tools are not well documented.	4,000 nodes	Not possible to gather statistics	Not modelled
PlanetSim	Discrete-event for both structured and unstructured overlay networks	A large amount of documentation is availble for its API and design.	100,000 nodes	No mechanism to gather statistics, but visualiser is available	Limited simulation of underlying network
PeerSim	Query-Cycle and Discrete-event for structured and unstructured networks	Only Query-Cycle simulator is documented	10^6 nodes (Query-Cycle)	Components can be implemented to gather statistical data	Not modelled
GPS	Discrete-event for both structured and unstructured overlay networks	Poor documentation for its API	1054 nodes	---	Partially models the underlaying network.
Neurogrid	Discrete-event for both structured and unstructured overlay networks	Good documentation	300,000 nodes claimed	It has for pre-determined variables, but for others the code shold be modified	Not modelled
Query-Cycle Sim.	Query-Cycle for unstructured networks	Only API documentation is available.	---	---	Not modelled
Narses	Discrete-event and flow-based	No documentation	600 nodes	Yes, but requires modification of source	A number of underlying topologies, balancing execution speed and accuracy
SicsSim-B	Discrete-event for structured and unstructured networks	In progress	Tested for up to 15,000 nodes for media streaming	Yes, but requires source code modification	Congestion, latency and bandwidth are modeled.

Figure 4.6: Properties of surveyed simulators

Simulator	Language	Status	Licence	URL
DHTSim	Java	Active	GPL	Not available
P2PSim	C++	Active	GPL	http://pdos.csail.mit.edu/p2psim/
Overlay Weaver	Java	Active	Apache	http://overlayweaver.sourceforge.net/
PlanetSim	Java	Active	LGPL	http://planet.urv.es/planetsim/
PeerSim	Java	Active	LGPL	http://peersim.sourceforge.net/
GPS	Java	Inactive	---	Not available
Neurogrid	Java	Inactive	GPL	http://www.neurogrid.net/
Query-Cycle	Java	Inactive	Apache	http://p2p.stanford.edu/
Narses	Java	Inactive	GPL-Like	http://sourceforge.net/projects/narses
SicsSim-B	Java	Active	---	http://www.sics.se

Figure 4.7: Complementary information about surveyed simulators

Chapter 5

Evaluation

In this chapter we investigate the performance of ForestCast under various scenarios and with different configurations. For this purpose, we carried out a simulation-based study over SicsSim-B.

Simulation results are in three parts. Firstly, we evaluate the effects of different policies for the heuristics we introduced in Section 3.1. Secondly, we select the best identified policies and investigate the properties of the overlay when the system scales. Finally, we examine the impacts of adding an incremental improvement to the base solution, as we described in Section 3.2. The metrics to be measured are:

- *Bandwidth utilization*: The ratio of the total utilized upload bandwidth to the total demanded download bandwidth.
- *Time to join*: The time it takes for a peer to receive the first segment after it sends its join request.
- *Startup delay*: The time it takes for a peer to start playing the media after it sends its join request.
- *Playback latency*: The difference between the playback point of a node and the media point. It shows the time it takes for a peer to play a segment of the media after that segment is sent out from the media server.
- *Tree depth*: The largest hop counts from the media server to the peers in a tree.
- *Quality*: The ratio of the number of received stripes to the number of demanded stripes.
- *Disruption*: The duration in which a peer does not have any segment of media to play and so the media is paused.

5.1 Experimental Setting

In our simulations, we assume one single server for maintaining the overlay and one media server to distribute the media. We also assume the stream is split into four stripes using MDC, so a peer is able to play the media even with a single stripe. However, the playback quality increases by receiving more stripes. The bit rate of each stripe is considered to be $128Kbps$.

Upload bandwidth of nodes follow the measurements in Sripanidkulchai et al. [53], which is a feasibility study of supporting large-scale live streaming applications with dynamic application end-points. The authors have used traces from a large content distribution network and characterized the behavior of users. Figure 5.1 shows the bandwidth distribution measured in [53]. Note that, in this measurment the encoding bit rate is assumed to be $250Kbps$ and the outbound degree is normalized by this value. As a result, almost half of the peers are *free riders*, that are the nodes who do not contribute any upload bandwidth. However, since we are using a lower bit rate per stripe, $128Kbps$, we enable some of the nodes, which are free-riders in [53], to upload one stripe. On the other hand, we set an absolute maximum bound of $5Mbps$ for the upload bandwidth of peers, such that no peer in our simulations can exceed this limit, even if they have more resources to contribute. For the host with unknown measurement in Figure 5.1, the authors have proposed three algorithms to estimate the upload bandwidth of nodes. We have chosen the *distribution* algorithm, which assigns a random value, drawn from the same distribution as that of the known resources, to each host.

Type	Degree bound	Number of hosts	Upload bandwidth (Kbps)
Free riders	0	58646 (49.3%)	0 - 249
Contributors	1	22264 (18.7%)	250 - 499
Contributors	2	10033 (8.4%)	500 - 749
Contributors	3 - 19	6128 (5.2%)	750 - 4999
Contributors	20	8115 (6.8%)	5000
Unknown	-	13735 (11.6%)	-
Total	-	118921 (100%)	-

Figure 5.1: Bandwidth distribution measured in real scenarios

The upload bandwidth of media server is supposed to be $10Mbps$. Moreover, we assume peers have enough download bandwidth to download all the stripes simultaneously. *Resource Index (RI)* is defined as the ratio of the available upload bandwidth of all nodes in the overlay (including the media source) to the total bandwidth demanded by nodes.

As mentioned in Chapter 4, the connection between two peers consists of three parts: a core network and two links that connect those nodes to the core network. Hence, link delay between two communicating peers is calculated as the sum of the delay in these three parts. Link delays is either 0 or $10ms$, with a uniform distribution. The core network's delay is generated as a random number, which follows a normal distribution with $\mu = 75ms$ and $\sigma = 12ms$. This value is always the same for a pair of nodes, but it differs from pair to pair; because we believe in Internet the delay between two nodes in a connection is usually constant, sometimes with a small deviation. To mimic this deviation, we also add a forth value, which is either 0 or $10ms$, to the total delay calculated for two nodes, every time a packet is transferred from one to the other. As a result, 95% of delays between two communicating peers has a value between $20ms$ and $160ms$ with a normal distribution.

Three types of event are defined: *join*, *leave* and *fail*. Events are modeled as a lottery event [63], with a poisson distribution with $\lambda = 100$, which means the expected interval between two consequent events is $100ms$. In other words, averagely 10 events happen per second.

Moreover, we define three different scenarios:

1. *Join only*: All the events are join, which means nodes join and stay in the system.
2. *Low rate departure*: 20% of the joined nodes leave or fail.
3. *High rate departure*: 50% of the joined nodes leave or fail.

A node may depart at any moment after it joins. More exactly, there is no assumption on the order of event types happening in the scenarios including leaves and failures. Note that, for a low rate departure scenario to end up in a system of certain size, more events than that of join only scenario should happen. Likewise, more events than that of the first two scenarios are required for the third scenario to end up in a network of the same size. Hence, hereafter we use the terms *low churn* instead of low rate departure, and *high churn* instead of high rate departure.

All the presented results are the average of at least 10 runs and the error bars, where shown, denote that we are confident that 95% of the these values are in the shown interval.

5.2 Impact of Different Heuristics

In this section, we introduce a number of heuristics according to what we discussed in Section 3.1. We then apply them to build up on overlay and investigate the properties of the resulting overlay. These heuristics are about:

- How to collect the nodes,

- How to select appropriate parents,
- What the startup segment is, and
- How long the buffering delay is.

In the following experiments, we conduct the simulation for low churn and high churn scenarios. The number of nodes, remaining in the system, when the scenario is over, is 2048.

5.2.1 Node Collection

Here, we observe how the overlay is influenced by the policy the server follows to select the open nodes according to Algorithm 1. We compare two policies. In the first policy, called *BFS policy*, the central server always selects the peers in *Breath First Order*. In the second policy, called *BFS-DFS policy*, the central server considers the upload bandwidth of the joining peer, and uses *Depth First Order* for free riders and BFS for other peers.

In BFS policy we start from top of the trees, where the media server is placed, and traverse them in BFS order. Hence, the peers, regardless of their properties, are placed as close as possible to the media server. In the BFS-DFS policy, we do not insist on placing the free riders close to the source. In contrast we try to put them somewhere further, not only by using a DFS traversal to collect the open nodes, but also by selecting the worst possible position for them.

The experiments show that we have shallower trees when we use BFS-DFS policy. This is because we have a remarkable number of free riders in the system, who can fill the upper layers of the trees and leave us with some narrow deep paths, if we apply BFS policy. On the other hand, if we use BFS-DFS policy, we lessen the probability of free riders cut off the subtrees at the layers close to the root.

Figure 5.2 shows the *Cumulative Distribution Function (CDF)* [11] of playback latency for these two policies in low churn and high churn scenarios. In this figure, the Y-axis shows the fraction of nodes in the system, and the X-axis shows the playback latency. We can see that the playback latency of peers when we are using BFS-DFS policy is lower than playback latency of peers with BFS policy. For example, in low churn scenario with BFS-DFS policy, 95% of peers have playback latency less than $2000ms$; while in the same scenario with BFS policy, the same fraction of peers have playback latnecy less than $3000ms$. In high churn scenario, although BFS policy shows a better result in the beginning, finally BFS-DFS policy outperforms it, such that 95th percentile happens at $3900ms$ and $5200ms$ for BFS-DFS and BFS policies, respectively.

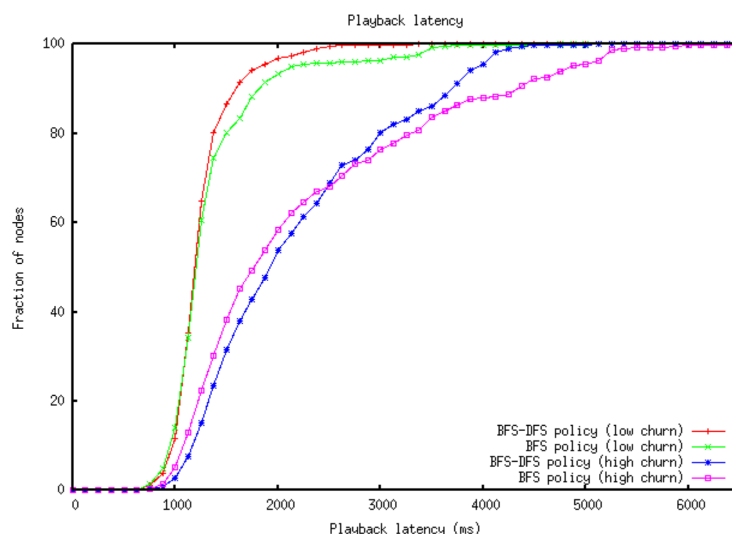

Figure 5.2: Playback latency of system for 2048 nodes for two cases of selecting open nodes by BFS-DFS policy and BFS policy (low churn and high churn)

Likewise, the quality of peers is improved by using BFS-DFS policy. Figure 5.3 is the CDF of quality received by peers with the two policies in low churn and high churn scenarios. In this figure, the Y-axis shows the fraction of nodes and the X-axis shows the percentage of received quality. We can see that in low churn scenario 78% of peers in BFS-DFS policy receive the stream with average quality more then 98% during their lifetime; while in the same scenario with BFS policy this is achieved in 72th percentile. Similarly, in the high churn scenario, BFS-DFS policy is more desirable than BFS policy and we have the minimum quality of 80% at 75th percentile in BFS-DFS policy and 65th percentile of peers in BFS policy.

5.2.2 Parent Selection

In this section, we study how ForestCast selects the supplying peers. After finding a set of open nodes as candidates, we should select one peer as a provider for each stripe. As mentioned in Section 3.1.1 we use a *priority function* to assign a value to each peer according to its properties.

One important heuristic on selecting parents is the *fair distribution* of stripes between parents. Suppose we have two stripes in the system, $stripe1$ and $stripe2$ and P can provide both of them, and also consider P is the only open node who posses $stripe2$. In this example if P uses all its upload bandwidth to transfer $stripe1$ then $stripe2$ will become unavailable in the system.

To see the result of fair distribution of stripes on the overlay we define two policies. In the first one, called *Rarest Stripe policy*, we define the priority function by

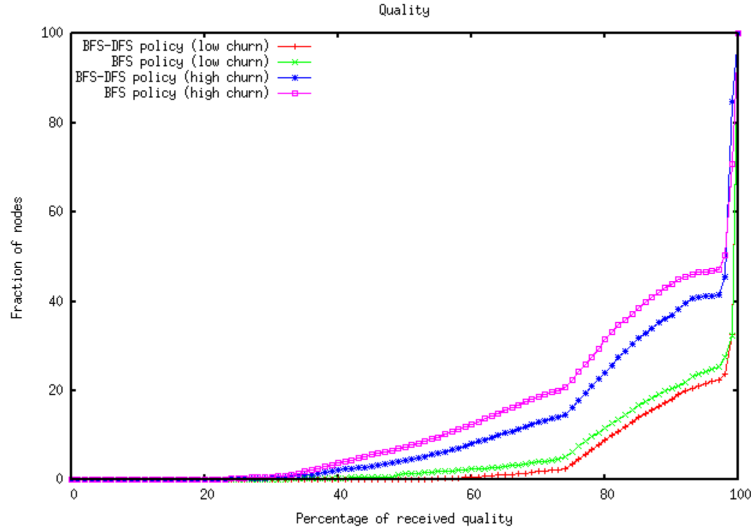

Figure 5.3: Quality of received media for 2048 nodes for two cases of selecting open nodes by BFS-DFS policy and BFS policy (low churn and high churn)

considering fanout of peers in each stripe tree as $\frac{b^\alpha}{f^\beta \cdot l^\gamma}$. The fanout, f, in each tree shows the number of peers that a peer is sending stripe to them. Considering fanout in priority function means that a peer who sends fewer number of one stripe, gets a higher value for forwarding that stripe. In the second policy, we ignore fanout of peers and define the priority queue as $\frac{b^\alpha}{l^\gamma}$. We call this policy, *Any Stripe policy*. We compare these two policies for 2048 peers in low churn and high churn scenarios. The results show that in 30% of experiments, by using Any Stripe policy, at least one of the stripe trees are saturated before all peers can join them, while we do not have this problem in case of using Rarest Stripe policy. By saturating each stripe tree, a group of peers can not join it and respectively can not receive that tree's stripe, and it causes the quality of received media decreases. So for the rest of evaluations we use Rarest Stripe policy.

Another important heuristic to find parents is *distinctness* between providers. We define two policies: In the first policy, called *Non-Distinct Parents policy*, each candidate peer who has higher priority has a more chance to be selected as a provider. In the second policy, in addition to peers' priorities, selecting distinct providers is also important. In this policy, called *Distinct Parents policy*, we avoid selecting repetitive parents in different stripe trees, if it is possible. For example, suppose for a node P, we are trying to find two providers for two stripes $stripe1$ and $stripe2$. If peer Q is selected as the provider for $stripe1$, due to its high priority, then we try to find another provider, like peer R, for $stripe2$, even if Q still has higher priority than R. Only when there is no other open node, which can provide $stripe2$, we choose Q once more.

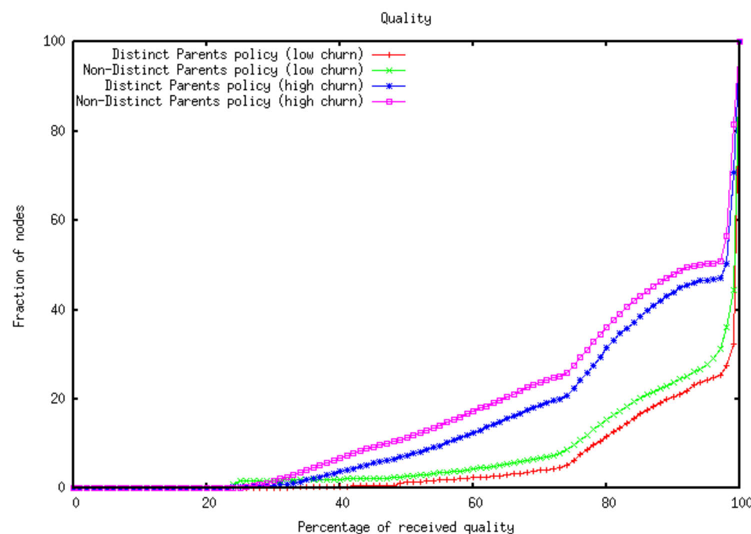

Figure 5.4: Quality of received media for 2048 nodes in cases of having Distinct Parents and Non-Distinct Parents policies (low churn and high churn)

The results show that the number of disrupted nodes with Distinct Parents policy is fewer than Non-Distinct Parents policy. By using the latter one, it is possible that one peer is selected as the provider for all the stripes of another peer. In such case if that provider fails, the peer fails to receive all its stripes. However, since we have selected different providers for different stripes in Distinct Parents policy, the probability of missing all the providers at the same time is less. In low churn scenario with Non-Distinct Parents policy, averagely 9 peers out of 2048 are disrupted, while it is less than 2 peers with Distinct Parents policy. In high churn these numbers are about 160 peers and 80 peers, respectively.

With a similar argument, we expect to have a higher quality in the second policy, which is confirmed in Figure 5.4. This figure shows CDF of quality received by 2048 peers in low churn and high churn scenarios. As we can see, in low churn scenario with Distinct Parents policy, about 75% of peers have a minimum quality of 98%, while with Non-Distinct Parents policy, about 70% of peers have the same quality. In high churn scenario with Distinct Parents policy, about 80% of peers have a minimum quality of 75% but this is achieved in about 70% of peers in Non-Distinct Parents policy.

5.2.3 Startup Segment

Startup segment determines a specific point of the media that a peer can start to play. This segment number is the first segment providers send to the peer. Whenever the providers of a peer are selected, they should agree on startup segment number.

In this heuristic, we study the effect of selecting different startup segments on the quality of stream.

We consider two policies to select startup segment. In the first policy, called *Head-Segment policy*, the minimum of head of buffers in providers are considered as the startup segment (Figure 5.5). In the other policy (Figure 5.6), we first find the range of common segments in the buffer of all providers, and then choose the segment in the middle of this range. We call this policy *Mid-Segment policy*.

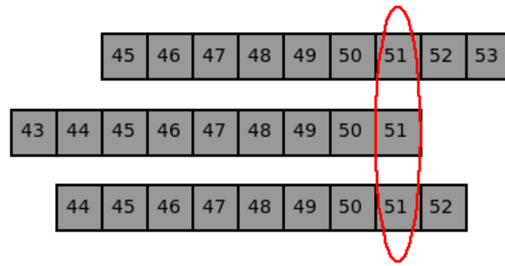

Figure 5.5: Minimum of head of buffers is selected as startup segment

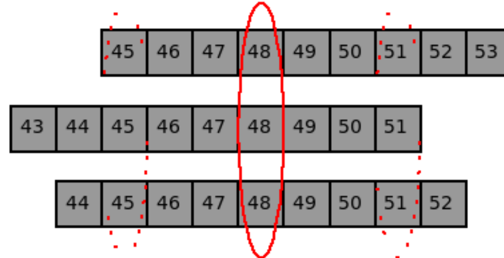

Figure 5.6: average of minimum of head of buffers and maximum of tail of them is selected as startup segment

By using Head-Segment policy, the playback latency, which is the difference between playback point of the media server and a peer, is less. It is because the selected segment number is the closest one to the most recent segment delivered by the media server. This is confirmed in Figure 5.7, which shows the CDF of playback latency for 2048 nodes for two policies. For example in low churn, about 95% of peers have playback latency less than $1900ms$ by Head-Segment policy, but 50% of peers have a higher playback latency in Mid-Segment policy. Likewise, in high churn scenario, only 10% of peers have playback latency more than $2100ms$ by using Head-Segment policy, whereas 50% of peers with Mid-Segment policy have a playback latency more than $2100ms$.

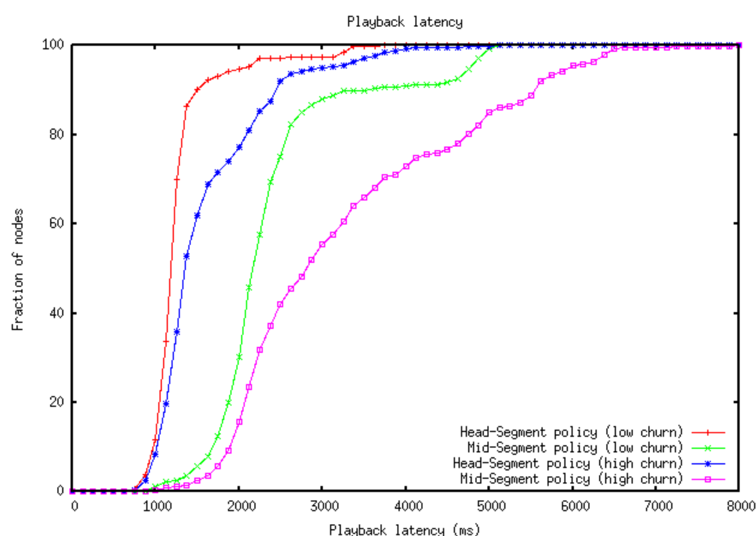

Figure 5.7: Playback latency of 2048 nodes for Head-Segment and Mid-Segment policies (low churn and high churn)

We also expected that by using Mid-Segment policy, the peers become more tolerant to the failure. Suppose P is providing $stripe1$ for Q and Q provides that stripe for R. If we use Head-Segment policy then if P fails, Q does not have any other segment of $stripe1$ in its buffer to send to R, whereas by using Mid-Segment, Q still has some segments to forward to R. Hence, a better quality is expected by using Mid-Segment policy. Surprisingly, the simulation shows that this argument is not valid. Figure 5.8 shows the CDF of quality for the two policies. In low churn scenario, with Head-Segment policy, about 80% of peers receive a minimum quality 98%, while 70% of peers get the same quality, in Mid-Segment policy. Similarly in high churn scenario, about 70% of peers receive 98% of quality and more by using Head-segment policy, but only 40% of peers receive the same quality. In ForestCast, the central server selects the startup segment or the common segment on providers by using its local information. Peers update this local information periodically. Meanwhile the peers update their information, the central server predicts the state of peers by their last information, and because of this, it is possible that it estimates the common segment wrong. For example, in Mid-Segment policy, it is possible that the selected segment number is smaller than the segment received by providers of the peer. In this case when the providers receive the server instruction to send that segment, they find out that their earliest segment in their buffer is bigger than the requested segment, so they start to send from the smallest segment number in their buffer. So by happening this scenario, the providers send different startup segment instead of a common segment, and it reduces the overall quality. The probability of happening this situation is lower in case of using Head-Segment policy.

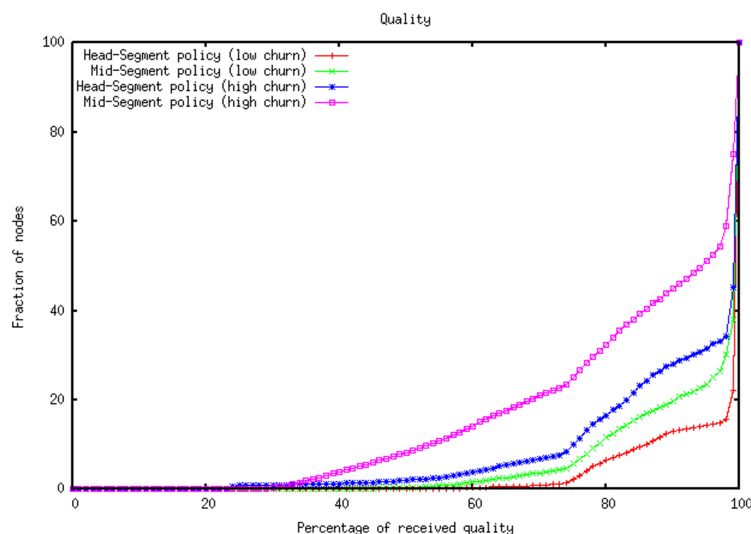

Figure 5.8: Quality of received media for 2048 nodes for Head-Segment and Mid-Segment policies (low churn and high churn)

5.2.4 Buffering Delay

Although each peer can start to play the media as soon as it receives its startup segment. It buffers the media for a while before playing it. The duration of buffering, called *buffering delay* has an important effect on the quality of the playback. In this study we change the time of buffering from $200ms$ to $3200ms$, and measure the change in quality and playback latency.

Figure 5.9 shows the CDF of playback latency and Figure 5.10 shows the CDF of quality received by peers. As we can see in Figure 5.9, the playback latency of peers increases by spending more time on buffering the stream before playback, but as Figure 5.10 shows, this will result in a better quality. For example for $T = 200ms$, the playback latency of 80% of peers is less than $1000ms$, while this time is about $4800ms$ for $T = 3200ms$. In Figure 5.10 however, we can see that more than 95% of peers in $T = 3200ms$ have a minimum quality 98%, but for $T = 200ms$, only 60% of peers have the same quality. That is because peers have more time to find substitute providers, is case of failure of any providers. To conclude, deciding about buffering delay is a trade of between quality and palyback latency of media stream.

5.3 Measurements at Scale

In this section, we demonstrate the behavior of ForestCast when the number of participating nodes increases. We start from 128 nodes and at each step double the size of the nodes in the overlay. In the following simulations we use the following heuristics and settings:

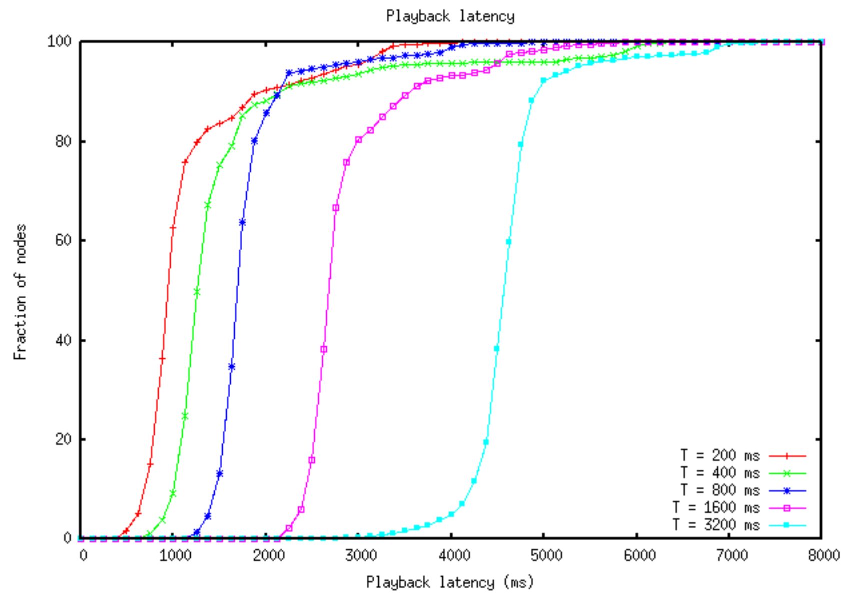

Figure 5.9: Playback latency of peers for 2048 nodes for different buffering delay (join only)

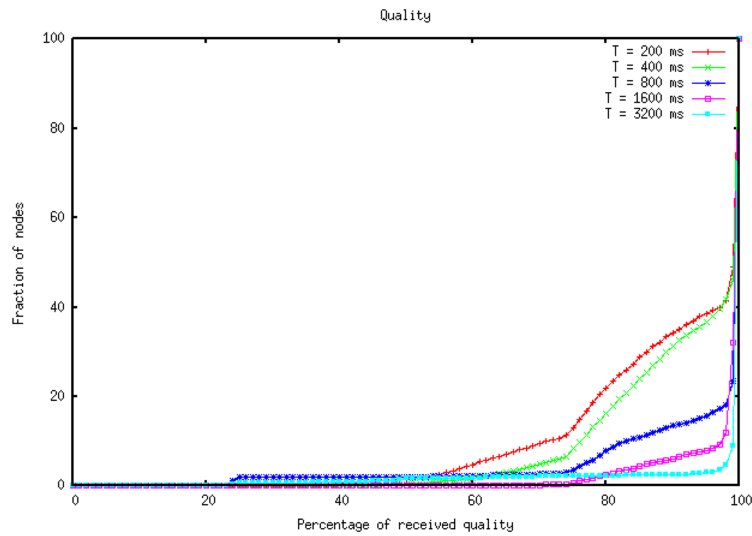

Figure 5.10: Quality of received media stream for 2048 nodes for different buffering delay (join only)

- We use BFS-DFS policy, for node collection;
- The value assigned to candidate providers is defined as $\frac{b^2}{f^2 \cdot l^4}$ (Algorithm 5). Note that a node may get different values in different trees;

- To select parents we use Distinct Parents and Rarest Strip policies;
- The central server uses the Head-Segment policy to find the startup segment;
- The buffering delay is $400ms$.

5.3.1 Bandwidth Utilization

Bandwidth utilization of the system is defined as the ratio of the total utilized upload bandwidth to the total demanded download bandwidth. Measurements show that bandwidth utilization of ForestCat is nearly 1 for all three scenarios, and for varying number of nodes. Note that a media streaming application is a bandwidth intensive application, therefore it is of great importance to utilize the bandwidth of peers as much as possible. If this value is less than 1, that would mean although there are peers who have free upload capacity, there are some other peers who can not receive some part of the media they request for. This might be either because the free peers cannot be identified, or they might not posses the requested data. The former is easy to be solved by a server who knows about all the peers in the overlay, but the latter is not that trivial. In ForestCast the server tries to distribute distinct stripes evenly in the system in order to prevent the latter problem. That is why the fanout of peers in different trees is taken into account and each peer prioritize forwarding its rarest stripe to a new child. Moreover, fragmenting data at the first place and using multiple trees are also important to achieve this high bandwidth utilization, because it makes it possible for peers with petty upload bandwidth to contribute some resource to the system and be helpful.

5.3.2 Tree Depth

The depth of trees is another metric that is important to us for several reasons. Firstly, we are interested in shallow trees because generally nodes in such trees are more resilient to damages in the structure i.e. failure of an interior node. Secondly, in shallow trees the average value for the playback latency is expected to be less than that of deep trees, because in shallow trees larger number of nodes are positioned closer to the media server. On the other hand, upload bandwidth of nodes and the order in which they enter the system, highly affects the depth of trees. For example if a number of free riders join the overlay at the beginning, then it is more probable for the overlay to have deep narrow trees. It is the responsibility of the central server to mitigate such problems and avoid building deep narrow trees, if possible.

In order to investigate the efficiency of our solution with this regard, it is important to see how the trees' depth grows when the system scales to the larger number of participating nodes. Figure 5.11 shows how the average depth of trees change when system size grows, for our three different scenarios, i.e. join only, low churn

and high churn. Note that, the Y-axis is presented in logarithmic scale.

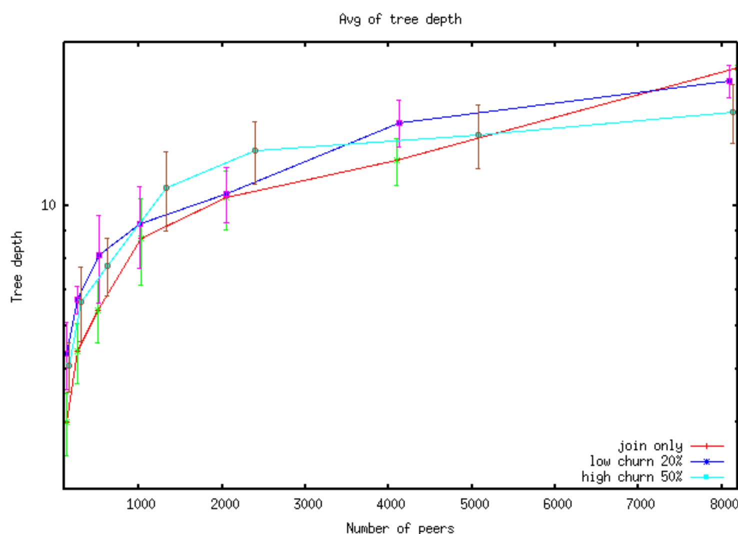

Figure 5.11: Average tree depth in ForestCast increases in a logarithmic scale when the network size grows

An interesting thing in this figure is that in high churn scenario the average tree depth is lower than the other two. Likewise, the low churn scenario has a lower growth rate than join only scenario for greater network sizes. We believe this is because peers, whose parents fail or leave, rejoin the overlay and may find a better position than their initial one. In other words the server is given a chance to revise its previous decisions and as a result trees evolve to a more desirable structure. This result motivated us to work on some kind of gradual modifications, when nodes are given a chance to bubble up the trees and find a better position in the overlay. As a result we came up with Section 3.2 on incremental improvements and Section 5.4 on its corresponding evaluation.

5.3.3 Startup Delay

Startup delay is the time it takes for a node to start playing the media after it sends its join request. This time consists of two parts. First is the time it takes for the node to recieve the first segment of media in at least one stripe, called *join delay* . It is good to recall that we assume MDC, which enables nodes to play the media even with a single stripe. Second is the time the node waits and buffer the media before playing it, called *buffering delay*. We assume that all nodes have the same buffering delay, that is $400ms$ in the following simulations. So we only need to measure the join delay. Since we are using a central server to handle the join requests, we expect this time to be independent of the size and shape of the overlay. This is confirmed by the result of the simulation, which is shown in Figure 5.12. We can see that for

large number of peers, this time is almost a constant in all three scenarios. Indeed, this time is sum of (i) the time required for the new node to contact the server, (ii) the time it takes for the server instructions to be received by selected parents of the new node, and (iii) the time it takes for the first segment to be transferred to the node.

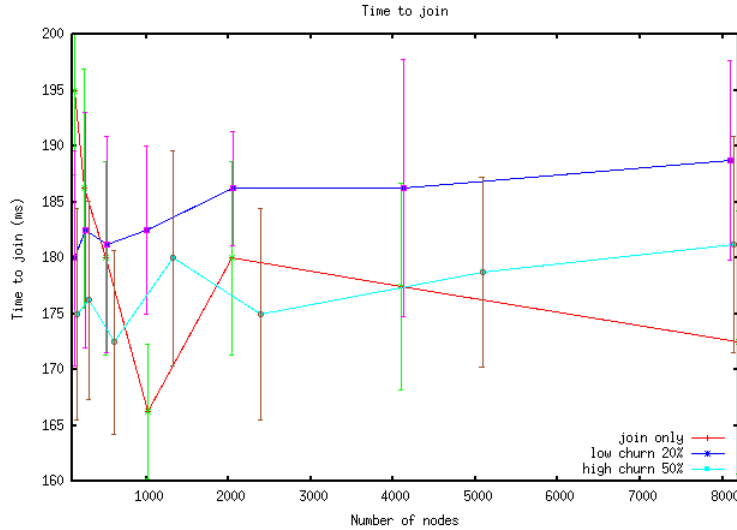

Figure 5.12: Average time it takes for a peer to join (join only, low churn and high churn scenarios)

It is worth mentioning that, the total startup delay is almost always a value less than $600ms$, which is insignificant compared to the measured startup delay in some of the successfully deployed solutions. For example, this value is reported to be averagely between 20 and 30 seconds for popular channels and up to two minutes for less popular channels in PPLive [22] and between 10 and 20 seconds in Coolstreaming [65]. As peers spend more time to buffer the data, they will be more resilient to failure of their parents. This is because in case a parent fails, the peer still has some data in its buffer to play. Hence, our next results for quality and disruption metrics would be comparable to the existing solutions only if this parameter is configured similarly. However, we do not change the buffering delay in our next simulations.

5.3.4 Playback Latency

We desire that nodes play nearly the same point of the media with negligible latencies to the media source. There are two things that affect the playback latency: (i) the buffering delay, which is the time each peer waits to buffer the data before playing it, and (ii) the position of the peer in trees. We talked about the buffering delay in Section 5.3.3 and mentioned that it influences the startup delay as well.

Here again, we assume the buffering delay is $400ms$. The difference between startup delay and playback latency is that, in the former we measure the time required for the node to play its first segment of the media, no matter what segment it is; whereas in the latter we are interested to see what point of the media is being played at a node and how much behind it is from the point of the media, which is currently broadcast by the media server. In other words, here we want to measure how long it takes for a node to play a segment of the media after that segment is sent out from the media server. Apparently this depends on how far the node is from the media server, that is the position of the node in the overlay.

In the following simulations we not only measure the average playback latency of nodes in the overlay, but also observe the fraction of nodes experiencing different playback latencies.

Figure 5.13 shows the average playback latency of nodes for join only, low churn and high churn scenarios. Apparently when the number of peers in system increases, the average playback latency of peers also increases. That is because when the number of peer grows, depth of the trees expands and so the number of nodes that are positioned further from the media server increases. What is more, we can see in Figure 5.13 that the best value is achieved in low churn scenario and the value in join only and high churn scenarios change very closely. Interestingly, for larger number of nodes the value in the high churn scenarios tends to be even better than that of join only scenario. We believe the reason is that in the join only scenario, shape of the overlay is highly affected by the order in which nodes come into the system, which in turn influences the average playback latency. In this scenario the playback latency constantly increases as we go down the trees. But in the presence of churn nodes have to move up or down the trees, when their parent leaves or fails. So there is a chance for nodes to improve their position. Even if they have to move down a tree, then that tree would not have an increasing playback latency as you go to the deeper layers. This would be beneficial for new arriving nodes, because they may have a better playback latency than that of what is expected for the layer in which they are placed. Moreover, if the churn rate increases, then some of the nodes may face disruption which would increase their playback latency. This result is interesting, because it persuades incorporating an incremental improvement in the solution, which gradually modifies the structure of the trees.

Figures 5.14, 5.15, and 5.16 show the CDF of peers experiencing different levels of playback latency in join only, low churn and high churn scenarios, respectively. As we can see in the join only scenario, for 128 nodes, 97th percentile happens at $100ms$, which is reasonable; but as the network size grows, this value has a significant increase, such that 97th percentile for 8192 nodes is nearly $4500ms$. Even the 90th percentile happens at $4000ms$.

Figure 5.15 demonstrate a more desirable growth in playback latency when the

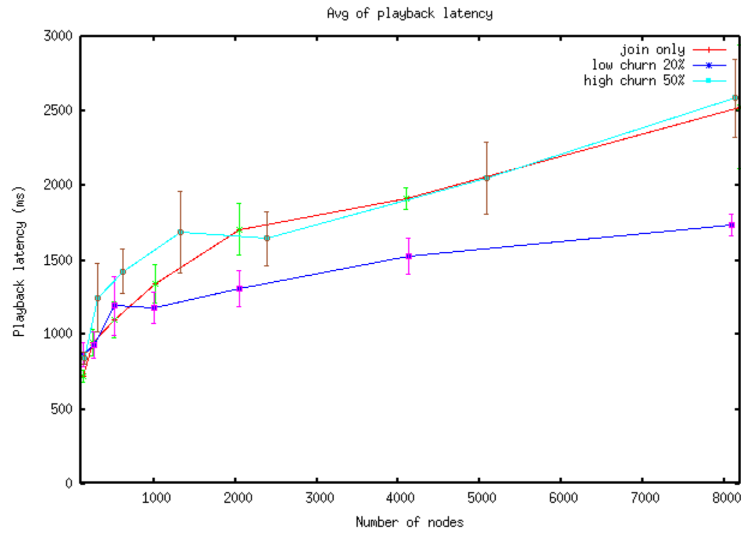

Figure 5.13: Average playback latency (join only, low churn and high churn scenarios)

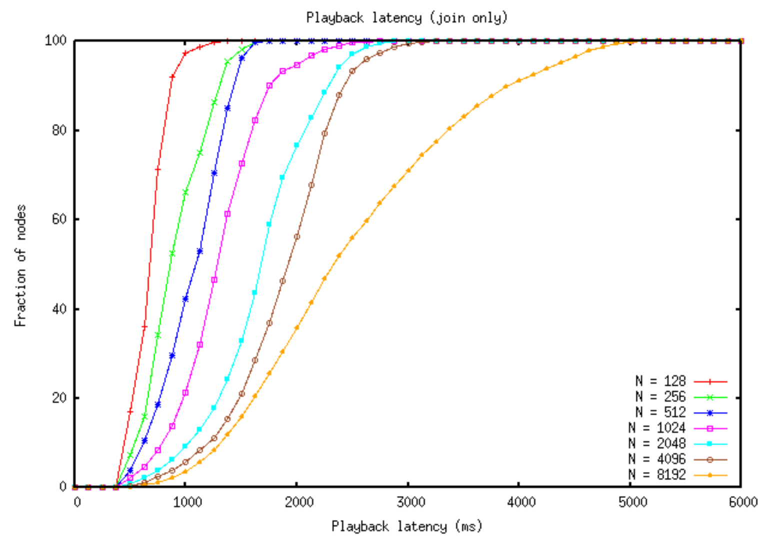

Figure 5.14: Fraction of peers experiencing different playback latencies (join only scenario)

number of nodes increases in the presence of a low rate churn. For example for 8192 nodes, 90th and 97th percentile happens at $2200ms$ and $3200ms$, respectively. Note that, these levels of playback latency for the same network size are achieved at 45th and 75th percentiles in join only scenario.

This comparison becomes more interesting as we come to Figure 5.16, which

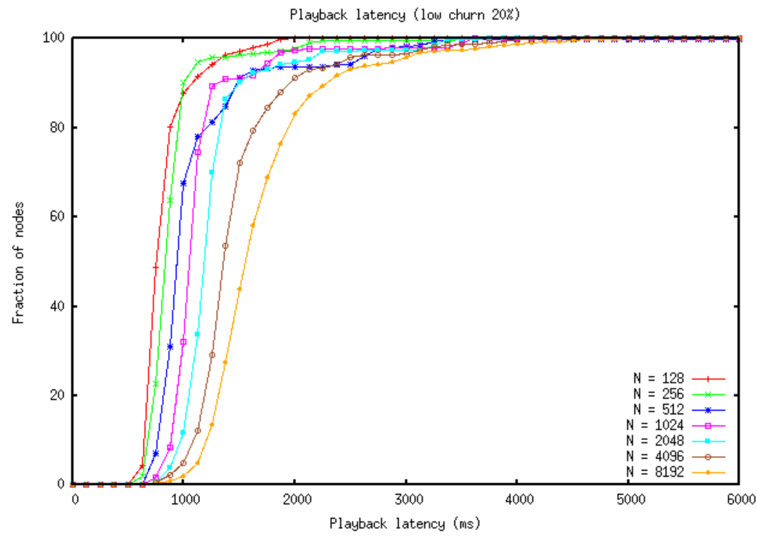

Figure 5.15: Fraction of peers experiencing different playback latencies (low churn scenario)

shows the same measurement in the high churn scenario. We can see that for 8192 nodes, 90th and 97th percentile happens at $3800ms$ and $5200ms$, respectively. Note that although the curve has a longer tail compared to join only scenario, it rises more sharply than that of join only scenario. Hence the average value for playback latency is better than that of join only scenario when the number of nodes grows.

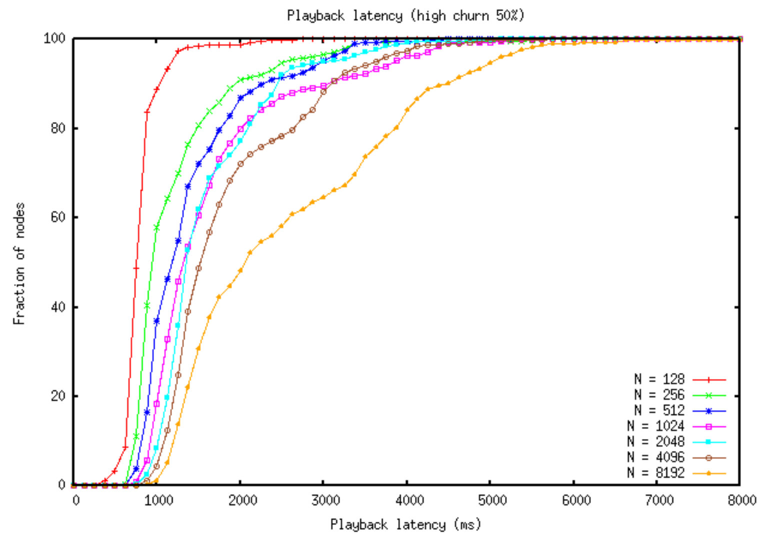

Figure 5.16: Fraction of peers experiencing different playback latencies (high churn scenario)

5.3.5 Received Quality

In this section, we measure the *quality* of media received by peers. Quality is defined as the ratio of the number of received stripes to the number of demanded stripes. Note that, the quality may change over time, due to the failure of parents, which may cause the peer not to receive one or more stripes for a while. Hence, we measure it segment by segment during the lifetime of peers in the overlay. It is worth mentioning that we have prioritized “not having disruption” over “having a low quality”. That is, for a node to proceed to playback, having a single stripe is enough, even though the quality degrades. But if for a certain point of media, there is no segment available from any of the stripes, then we would have a *disruption*. We will shortly be back to this latter metric and for now, we only focus on the quality metric, which does not include the disruption intervals.

In order to evaluate quality, we not only measure the average *quality* received by nodes, but also demonstrate fraction of nodes receiving different levels of quality. Figure 5.17 shows the average quality received by peers in our three scenarios. Not surprisingly, in join only scenario, all peers receive all stripes they have asked for. But as the failures start to happen, e.g. in the low churn scenario, there are nodes that miss some of the stripes and so their quality decreases. As the number of failures increases, e.g. in the high churn scenario, the average quality lessens even more. An interesting point here is that in all the scenarios, increasing the number of peers does not dramatically change the average quality. Even when the network size exceeds 8000 nodes in the high churn scenario, the average quality never falls below %85.

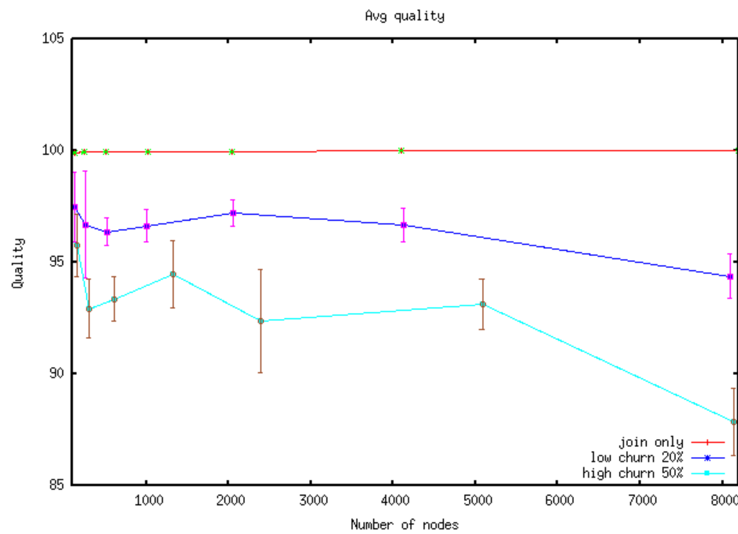

Figure 5.17: Average quality received by nodes (join only, low churn and high churn scenarios)

Figures 5.18 and 5.19 show the CDF of peers receiving different quality for low churn and high churn scenarios, respectively. Note that in the join only scenario, which is not shown here almost all the peers have quality more than 98% and this quality does not change by increasing the number of peers in the system.

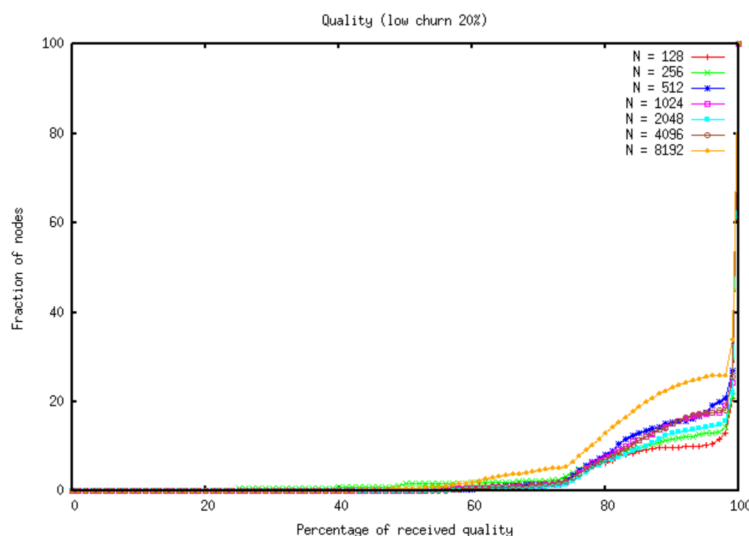

Figure 5.18: Fraction of nodes receiving varying levels of quality (low churn scenario)

As Figure 5.18 shows, more than 80% of node receive a nearly full quality of the media during their lifetime regardless of the network size. The other 20 percentage of nodes have a quality between 75% and 100%. Since we have four stripes in the system and almost all nodes have enough download capacity to receive all four stripes, this result means that all the nodes receive at least three out of four stripes.

Figure 5.19 shows a similar trend. In the high churn scenario, nearly 70% of nodes receive all four stripes, and the rest receive three out of four, which results in a quality between 75% and 100%; except for the network size 8192. Even in this last case, half of the nodes receive all four stripes and the other half receive at least two out of four stripes.

5.3.6 Disrupted Nodes

Disruption is the duration a peer does not have any segment of media to play. A disruption ends as soon as the next segment becomes available at the peer. Figure 5.20 shows the number of disrupted nodes in all three scenarios. As we can see, in join only scenario we do not have any disrupted node and in low churn scenario for 8000 nodes less than 0.1% of nodes are disrupted. We have the most number of disrupted nodes in high churn scenario and with 8000 nodes. In this case we have

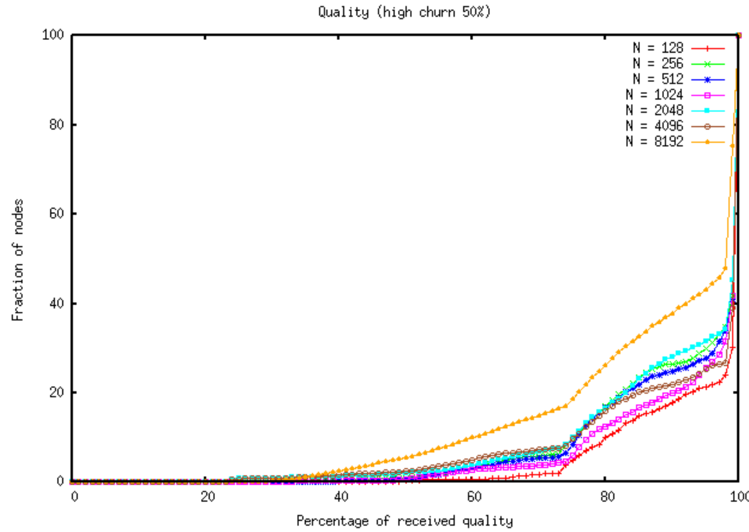

Figure 5.19: Fraction of nodes receiving varying levels of quality (high churn scenario)

about 200 disrupted nodes, which is about 2.5% of nodes in system.

Disruptions may have different severity levels. We believe a long disruption is worse than several short disruptions. In order to take into account how sever a disruption is, we use a counter, which counts the time units the media is paused. When the playback is resumed, we raise the measured time to the power of two and add it to a total variable for disruption severity, and reset the counter. This new metric, called *disruption severity*, shows the harshness of disruptions at a peer during its membership in the overlay. Disruption severity is more sensitive to long disruptions, because they are magnified by power of two. For example, suppose the media stream has 1000 segments. If a peer misses 100 non-consecutive of the whole media, its disruption severity is 100. A peer, which misses 10 consecutive segments has the same disruption severity.

Our measurement for disruption severity, which is not presented in the document, shows that very few nodes suffer from noticeable disruptions. In the worse case, the disruption severity is less than 100 for a media stream with 10000 segments, which is equal to missing 10 consecutive segments or 100 non-consecutive segments.

5.4 Impact of Incremental Improvement

Up to here, we have not considered any improvement in the overlay. We repeated some of the above simulations while incorporating the incremental improvements, which we described in Section 3.2. Moreover, in order to choose a node to leave or

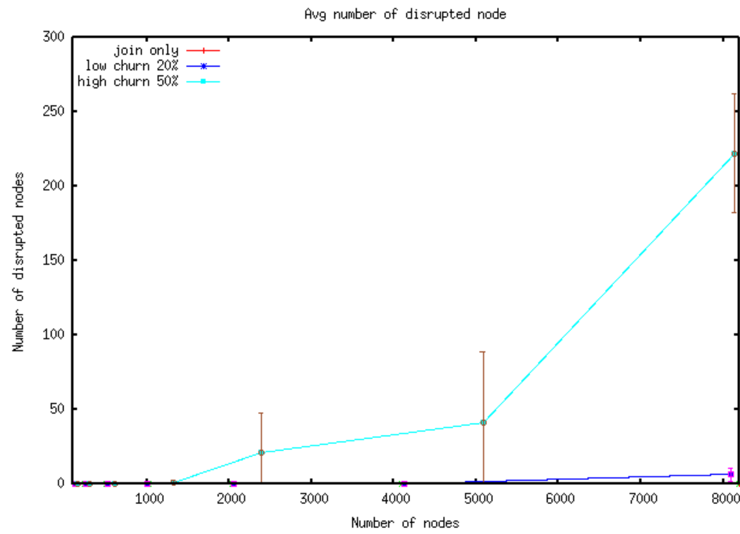

Figure 5.20: Average number of disrupted nodes (join only, low churn and high churn scenarios)

fail, we use a pareto distribution [44] with parameter values 1 for location and 0.5 for shape. As a result, the simulator chooses an older node with less probability than a younger node, for the purpose of the leave or fail.

Results show that in all three scenarios (join only, low churn and high churn) the average tree depth is nearly less than half compared to when we had no improvements. As described before, tree depth is an important factor. Having shallower trees not only lessens the vulnerability of nodes to churn in the trees, which in turn increases the quality and decreases the probability of having disruption, but also influences the average playback latency of nodes. It is worth mentioning that, this improvement may not be in favor of some of the nodes, i.e. the nodes that are pushed down a tree. These nodes may experience some quality degradation or even disruptions; but our measurements show that the average value for received quality improves.

Chapter 6

Future Work

We believe that the idea in ForestCast can be used in either a centralized or a decentralized model. In the centralized model, as we discussed, a central server is responsible for building and maintaining the data overlay. In decentralized model nodes are given a partial view of the network and they actively cooperate with their neighbors to send and receive different stripes. In this chapter we introduce our idea to extend ForestCast to a decentralized solution, which we believe can be further improved.

In the decentralized approach we use similar heuristics to build up the trees and maintain the overlay, but without the assumption of a global knowledge. Instead, nodes maintain local views of the network and they participate in making decision to send or receive data. We assume that media is fragmented into data segments, which are numbered by the media server sequentially. In this algorithm, we use hierarchical-like model to find supplying peers and use push-pull model to deliver data.

Finding Supplying Peers

To find supplying peer, we need to explain the meaning of *local state (LS)* of a node and its *global state (GS)*. The local state of each node i is defined as follow:

$$LS(i) = \Phi_i(b_i, l_i, f_i)$$

Φ_i is a function of node i's available upload bandwidth, b_i, its latency to the media source, l_i, and its fanout, f_i. We define the global state as follow:

$$GS(i) = \Theta_i(LS(i), GS(ch_i))$$

Θ_i is a function of local state of node i and global states of its children. Each node periodically sends its global state to its parent. When a node receives its children's global state, it calculates its own value and sends it upward till it hits the root.

Whenever a new node comes into system at first, it contacts the *rendezvous point* to get a list of nodes as its neighbor set, and then it sends its join request to the media server. If the media server has enough free bandwidth, it accepts the new node as its own child. Otherwise it forwards the join request to its *best child* to adopt the new node. The best child is the node who has the highest global state value among others. The node that receives a join request will do the same until someone accepts the joining node. Through the join procedure, the new node updates its neighbor set, considering the neighbor set of nodes on the path. Experiments show that the size of this neighbor set does not need to be very large. In [55] it is shown that having a neighbor set of size 8 would suffice, but the authors suggest a neighbor set of size 20. In our model, we consider the size 10 for neighbor set of each node.

Data Delivery
We believe that the best mechanism in data delivery is a push-pull model. For this to work, we assume the media server fragments the media into data segments. Data delivery has two phases: in the first phase, the data segments are pushed down the tree, and in the second phase the nodes pull the missing segments from their neighbor set. Each node periodically sends its data availability information to its neighbor set. We use tit-for-tat strategy to motivate the peers to cooperate in data delivery.

Optimization
The idea of optimization is the same as what we propose for centralized model. We desire nodes with higher bandwidth to be placed closer to the source, but this can be risk if the node is not that reliable. On the other hand we know that a node which has long stayed in the overlay is less probable to leave the overlay. like Section 3.2, we define the *strength* of each node as a function of node's age, its free upload bandwidth, and the its fanout:

$$Strength(A_i) = age_A \cdot (fanout_{A_i} + freeBw_{A_i})$$

A node periodically sends its strength value to its parent. If a node finds out its strength is smaller than that of one of its children, it changes its position with that child. The strong child then accepts its old parent and its old siblings as its children. If it does not have enough free upload bandwidth to keep all of its old children, it releases the weaker one. Orphan children will be either adopted by the demoted parent or rejoin the system.

As a result eventually we will have more reliable nodes with more bandwidth closer to the source. Note that when a node jumps up the tree, its latency to the media source will decrease, but it will not have any effect on the playback latency. The only effect is that the node would have more time to buffer the data. This would not only result in a better quality of the media the node is receiving but also it would be more tolerant to the failure of its parent or ancestors.

Chapter 7

Conclusion

In this document we worked on peer-to-peer live streaming systems, reviewed different solutions in the field and proposed ours. Mainly this thesis, we focused on three topics: (i) designing an algorithm for peer-to-peer media streaming, called *ForestCast*, (ii) implementing a stochastic discrete event peer-to-peer simulator, which also models bandwidth and link latencies, called *SICSSIM-B*, and finally (iii) investigating how different heuristics will affect the quality of service experienced by clients.

ForestCast
There are different methods in peer-to-peer systems to construct overlay network and deliver data to peers. ForestCast uses a centralized directory to construct and maintain the overlay, and multiple trees for data delivery. Our proposed solution, differs from the existing solutions in that we solve the problem using a heuristic technique. ForestCast provides a framework, which can be configured with different constraints and objective functions. In other words, we choose a performance centric approach that easily adopts to various design goals. So this solution can serve a wide range of media providers, who have their own requirements, priorities and objectives.

SICSSIM-B
As mentioned, currently there very few peer-to-peer simulators available. These simulators either have to consider all the details of packet transmissions in network, which makes them barely scalable for simulating huge number of data packets, as it is required in media streaming, or they have to abstract all the underlying layers of network, which leads to inaccurate measurements. Hence, to evaluate our algorithm, we developed our simulator, called *SICSSIM-B*. Since our application involves high bandwidth data transfer, we choose a flow-level model to be able to conduct experiments at large scale. To deal with inaccuracy we improve this model by incorporating some of the effects of the underlying network, e.g. latency, bandwidth and congestion.

Investigation different heuristics

We investigate different heuristics for constructing overlay network. What is desired in all the existing algorithms, is to construct and maintain an efficient data delivery overlay, but there is no common definition for being "efficient". This is why the algorithms are not fairly comparable to one another. If we use a central server with global knowledge, which can quickly make the best possible decisions with respect to certain criteria and objectives, we will end up in an optimum overlay for the respective configuration.

Our experiments show that for positioning nodes in the trees, it is very important to consider the properties of the joining nodes. For example it is a wise decision to put nodes with petty bandwidth as far as possible from the media source, by using a Depth First Search policy. Even during the lifetime of an overlay, incorporating an incremental improvement, which pulls the stronger nodes up the trees and pushes the weaker nodes down, will result in a more efficient overlay. Note to be taken, that a strong node is not only a node with a high bandwidth, but the one who has also stayed long in the system.

We also observe that to maximize bandwidth utilization, it is of crucial importance to uniformly distribute different parts of the media, e.g. substreams, in the overlay. For example, nodes are better to prioritize forwarding a substream, which they have forwarded less. A fair distribution of data in the overlay, prevents a situation in which some part of the data becomes rare, and even though there is a free capacity in the network for data delivery, data is not accessible and the bandwidth can not be utilized.

What is more, we show that selecting distinct parents for a node, makes it more fault tolerant and will decrease the number and severity of disruptions nodes may ever experience.

Another interesting result we come up with, is that a greedy approach to choose the startup segment for a joining node, would result in not only a better playback latency, but also, surprisingly, a better quality. More exactly, although one may think starting with an earlier point of the media would make a node more fault tolerant, we conclude that it is always better to provide the nodes with the most recent data available, that is the largest segment all the parents have in common.

Bibliography

[1] GONG An, DING Gui-guang, DAI Qiong-hai, and LIN Chuang. Bulktree: an overlay network architecture for live media streaming, 2006.

[2] Fast Retransmit And. Network working group w. stevens request for comments: 2001 noao category: Standards track january 1997 tcp slow start, congestion avoidance,.

[3] Suman Banerjee, Bobby Bhattacharjee, and Christopher Kommareddy. Scalable application layer multicast. In *SIGCOMM '02: Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 205–217, New York, NY, USA, 2002. ACM.

[4] A.R. Bharambe, S.G. Rao, V.N. Padmanabhan, S. Seshan, and H. Zhang. The impact of heterogeneous bandwidth constraints on DHT-based multicast protocols. *Proc. of IPTPS*, 2005.

[5] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[6] Dejan Kosti C, Adolfo Rodriguez, Jeannie Albrecht, and Amin Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh, Sep 2003.

[7] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. Splitstream: high-bandwidth multicast in cooperative environments. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 298–313, New York, NY, USA, 2003. ACM Press.

[8] J. Chakareski, S. Han, and B. Girod. Layered coding vs. multiple descriptions for video streaming over multiple paths. In *MULTIMEDIA '03: Proceedings of the eleventh ACM international conference on Multimedia*, pages 422–431, New York, NY, USA, 2003. ACM.

[9] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: a distributed anonymous information storage and retrieval system. In *International workshop on Designing privacy enhancing technologies*, pages 46–66, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[10] Bram Cohen. Incentives build robustness in bittorrent, 2003.

[11] "Cumulative Distribution Function". Accessed Jan-2008, Available: $http : //en.wikipedia.org/wiki/cumulative_{d}istribution_{f}unction.$

[12] F. Dabek, B. Zhao, P. Druschel, and I. Stoica. Towards a common api for structured peer-to-peer overlays, 2003.

[13] "DHTSim". Accessed May-2006, Not available: $http : //www.informatics.sussex.ac.uk/users/ianw/teach/dist - sys.$

[14] Kolja Eger, Tobias Hobfeld, Andreas Binzenhofer, and Gerald Kunzmann. Efficient simulation of large-scale p2p networks: packet-level vs. flow-level simulations. In *UPGRADE '07: Proceedings of the second workshop on Use of P2P, GRID and agents for the development of content networks*, pages 9–16, New York, NY, USA, 2007. ACM Press.

[15] Ayalvadi J. Ganesh, Anne-Marie Kermarrec, and Laurent Massoulié. Peer-to-peer membership management for gossip-based protocols. *IEEE Trans. Comput.*, 52(2):139–149, 2003.

[16] Pedro GarciÂa, Carles Pairot, Ruben Mondejar, Jordi Pujol, Helio Tejedor, and Robert Rallo. *PlanetSim: A New Overlay Network Simulation Framework.* 2005.

[17] Ali Ghodsi. *Distributed k-ary System: Algorithms for Distributed Hash Tables.* PhD dissertation, KTH—Royal Institute of Technology, Stockholm, Sweden, October 2006.

[18] Vivek K. Goyal. Multiple description coding: Compression meets the network. *IEEE Signal Processing Magazine*, 18(5):74–93, September 2001.

[19] Yang Guo, Kyoungwon Suh, Jim Kurose, and Don Towsley. Directstream: A directory-based peer-to-peer video streaming service, 2006.

[20] Indranil Gupta, Ken Birman, Prakash Linga, Al Demers, and Robbert van Renesse. Kelips: Building an efficient and stable P2P DHT through increased memory and background overhead. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, 2003.

[21] M. Hefeeda, A. Habib, D. Xu, B. Bhargava, and B. Botev. Collectcast: A peer-to-peer service for media streaming, 2003.

[22] X. Hei, C. Liang, J. Liang, Y. Liu, and KW Ross. Insights into pplive: A measurement study of a large-scale p2p iptv system. *Proc. of IPTV Workshop, International World Wide Web Conference*, 2006.

[23] Xuxian Jiang, Yu Dong, Dongyan Xu, and Bharat Bhargava. Gnustream: A p2p media streaming system prototype, 2003.

[24] M. Frans Kaashoek and David R. Karger. Koorde: A simple degree-optimal distributed hash table. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, 2003.

[25] Cameron Kiddle, Rob Simmonds, Carey Williamson, and Brian Unger. Hybrid packet/fluid flow network simulation. In *PADS '03: Proceedings of the seventeenth workshop on Parallel and distributed simulation*, page 143, Washington, DC, USA, 2003. IEEE Computer Society.

[26] D. Kostic, A. Rodriguez, J. Albrecht, A. Bhirud, and A. Vahdat. Using random subsets to build scalable network services. In *Proceedings of USITS*, 2003.

[27] Jinyang Li, Jeremy Stribling, Robert Morris, and M. Frans Kaashoek. Bandwidth-efficient management of DHT routing tables. In *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI '05)*, Boston, Massachusetts, May 2005.

[28] B. Liu, Y. Guo, J. Kurose, D. Towsley, and W. Gong. Fluid simulation of large scale network: Issues and tradeoffs TITLE2:. Technical Report UM-CS-1999-038, , 1999.

[29] Benyuan Liu, Daniel R. Figueiredo, Yang Guo, James F. Kurose, and Donald F. Towsley. A study of networks simulation efficiency: Fluid simulation vs. packet-level simulation. In *INFOCOM*, pages 1244–1253, 2001.

[30] Thomas Locher, Remo Meier, Stefan Schmid, and Roger Wattenhofer. Push-to-Pull Peer-to-Peer Live Streaming. In *21st International Symposium on Distributed Computing (DISC), Lemesos, Cyprus, Springer LNCS 4731*, September 2007.

[31] Nazanin Magharei and Reza Rejaie. Prime: Peer-to-peer receiver-driven mesh-based streaming. In *INFOCOM*, pages 1415–1423, 2007.

[32] G. Manku, M. Bawa, and P. Raghavan. Symphony: Distributed hashing in a small world, 2003.

[33] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the xor metric, 2002.

[34] 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*, pages 275–282, Washington, DC, USA, 2006. IEEE Computer Society.

[35] S. Naicken, A. Basu, B. Livingston, and S. Rodhetbhai. A Survey of Peer-to-Peer Network Simulators. In *Proceedings of The Seventh Annual Postgraduate Symposium, Liverpool, UK*, 2006.

[36] S. Naicken, B. Livingston, A. Basu, S. Rodhetbhai, I. Wakeman, and D. Chalmers. The state of peer-to-peer simulators and simulations. *SIGCOMM Comput. Commun. Rev.*, 37(2):95–98, April 2007.

[37] Animesh Nandi, Aditya Ganjam, Peter Druschel, T. S. Eugene Ng, Ion Stoica, Hui Zhang, and Bobby Bhattacharjee. Saar: A shared control plane for overlay multicast. In *NSDI*. USENIX, 2007.

[38] "Narses Network Simulator". Accessed Jan-2008, Available: *http : //sourceforge.net/projects/narses*.

[39] "NeuroGrid". Accessed Jan-2008, Available: *http : //www.neurogrid.net*.

[40] "Overlay Weaver: An Overlay Construction Toolkit". Accessed Jan-2008, Available: *http : //overlayweaver.sourceforge.net*.

[41] "P2Psim: A Simulator for Peer-to-Peer (P2P) Protocols". Accessed Jan-2008, Available: *http : //pdos.csail.mit.edu/p2psim*.

[42] Venkata N. Padmanabhan, Helen J. Wang, Philip A. Chou, and Kunwadee 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*, pages 177–186, New York, NY, USA, 2002. ACM.

[43] V. Pai, K. Kumar, K. Tamilmani, V. Sambamurthy, and A. Mohr. Chainsaw: Eliminating trees from overlay multicast, 2005.

[44] "Pareto Distribution". Accessed Jan-2008, Available: *http : //en.wikipedia.org/wiki/Pareto*$_d$*istribution*.

[45] "PeerSim: P2P Simulator". Accessed Jan-2008, Available: *http : //peersim.sourceforge.net*.

[46] Fabio Pianese, Joaquin Keller, and Ernst W Biersack. PULSE, a flexible P2P live streaming system. In *9th IEEE Global Internet Symposium 2006 in conjunction with IEEE Infocom 2006, 28-29 April 2006, Barcelona, Spain*, Apr 2006.

[47] "PlanetSim: An Overlay Network Simulation Framework". Accessed Jan-2008, Available: *http : //planet.urv.es/planetsim*.

[48] "PPLive". Accessed Jan-2008, Available: *http : //www.pplive.com*.

[49] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218:329–??, 2001.

[50] Antony I. T. Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. Scribe: The design of a large-scale event notification infrastructure. In *NGC '01: Proceedings of the Third International COST264 Workshop on Networked Group Communication*, pages 30–43, London, UK, 2001. Springer-Verlag.

[51] Mario T. Schlosser, Tyson E. Condie, and Sepandar D. Kamvar. Simulating a p2p file-sharing network, 2002.

[52] Subhabrata Sen and Jia Wang. Analyzing peer-to-peer traffic across large networks. In *Second Annual ACM Internet Measurement Workshop*, November 2002.

[53] K. Sripanidkulchai, A. Ganjam, B. Maggs, and H. Zhang. The feasibility of supporting large-scale live streaming applications with dynamic application end-points. *Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 107–120, 2004.

[54] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.

[55] S. Tewari and L. Kleinrock. Analytical model for bittorrent- based live video streaming. In *3rd IEEE International Workshop on Networking Issues (NIME'07)*, Las Vegas, NV, January 2007.

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

[57] D. Tran, K. Hua, and T. Do. Zigzag: An efficient peer-to-peer scheme for media streaming, 2003.

[58] Eveline Veloso, Virgílio Almeida, Wagner Meira, Azer Bestavros, and Shudong Jin. A hierarchical characterization of a live streaming media workload. In *IMW '02: Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, pages 117–130, New York, NY, USA, 2002. ACM.

[59] Vidhyashankar Venkataraman, Kaouru Yoshida, and Paul 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*, pages 2–11, Washington, DC, USA, 2006. IEEE Computer Society.

[60] Vivek Vishnumurthy and Paul Francis. On heterogeneous overlay construction and random node selection in unstructured p2p networks. In *INFOCOM*, 2006.

[61] L. Vu, I. Gupta, J. Liang, and K. Nahrstedt. Mapping the pplive network: Studying the impacts of media streaming on p2p overlays. *Department of Computer Science, University of Illinois at Urbana-Champaign, Tech. Rep. UIUCDCS-R-2006-275, Aug*, 2006.

[62] Long Vu, Indranil Gupta, Jin Liang, and Klara Nahrstedt. Insights into PPLive: A Measurement Study of a Large-Scale P2P IPTV System. Technical report, 2005.

[63] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Operating Systems Design and Implementation*, pages 1–11, 1994.

[64] Feng Wang, Yongqiang Xiong, and Jiangchuan 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*, page 49, Washington, DC, USA, 2007. IEEE Computer Society.

[65] S. Xie, G.Y. Keung, and B. Li. A Measurement of a large-scale Peer-to-Peer Live Video Streaming System. *Parallel Processing Workshops, 2007. ICPPW 2007. International Conference on*, pages 57–57, 2007.

[66] Weishuai Yang and Nael Abu-Ghazaleh. Gps: A general peer-to-peer simulator and its use for modeling bittorrent. In *MASCOTS '05: Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 425–434, Washington, DC, USA, 2005. IEEE Computer Society.

[67] W. P. Ken Yiu, Xing Jin, and S. H. Gary Chan. Challenges and approaches in large-scale p2p media streaming. *IEEE MultiMedia*, 14(2):50–59, 2007.

[68] Xinyan Zhang, Jiangchuan Liu, Bo Li, and Tak-Shing Peter Yum. Coolstreaming/donet: A data-driven overlay network for peer-to-peer live media streaming, 2005.

[69] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, April 2001.

Index

A

Any Stripe policy, 62
Application Level Multicast, 1, 19

B

Bandwidth matrix, 50
Bandwidth utilization, 31, 57, 68
BFS policy, 60
BFS-DFS policy, 60
Bloom Filter, 27
Buffering delay, 37, 66
Bulk Tree, 9
Bullet, 26

C

Centralized method, 8, 12
Chainsaw, 22
Chunk driven, 14, 21, 22, 24
ChunkySpread, 20
CollectCast, 14
Controlled flooding method, 11, 12
CoolStreaming, 11, 14, 21
CoopNet, 8

D

Description, *see* stripe
DHT based method, 9, 12
DHTSim, 53
DirectStream, 8, 17
Discrete event simulation, 47
Disruption, 57, 73, 76
  severity, 76
Distinct parent, 62
Distinct Parent policy, 62
Distinct parents, 35

F

FEL, 47, 51
Flow level simulation, 47
Fluid based simulation, *see* flow level simulation
ForestCast, 2, 31, 32
Free rider, 58, 60
Future Event List, *see* FEL

G

GnuStream, 11, 14, 23
Gnutella, 11, 23
Gossip based method, 11, 12
GPS, 54

H

Head of buffer, 32
Head-Segment policy, 64
Hierarchical method, 9, 12
High churn, 59
High rate departure, 59

I

IPTV, 24

J

Join only, 59

L

Lottery event, 59
Low churn, 59
Low rate departure, 59

M

MDC, 13
Media point, 32

Media streaming, 1
Mid-Segment policy, 64
mTreeBone, 26
Multiple Description Coding, *see* MDC

N

Narses, 54
Neurogrid, 54
Nice, 9
Node strength, 40
Non-Distinct Parents policy, 62
Normal distribution, 59

O

Open node, 33
Orchard, 19
Overlay Weaver, 53

P

P2PSim, 53
Packet level simulation, 47
Pastry, 17, 18, 27
Peer profile, 33
PeerSim, 53
PlanetSim, 53
Playback latency, 31, 33, 57, 71
Playback point, 32
Poisson distribution, 59
PPLive, 24
Prime, 15, 25
Priority function, 35
Pull method, 12, 14
Pulsar, 10, 15, 24
PULSE, 11, 14, 22
Push method, 12, 13
Push-Pull method, 12, 15

Q

Quality, 31, 57, 63, 73
Query-Cycle, 54

R

RanSub Protocol, 26
Rarest Stripe policy, 62
Receiver driven, 14, 23
Resource index, 58
Ripple effect, 48

S

SAAR, 27
SCAM, 21
SCAMP, 26
Scribe, 17, 18, 27
SICSSIM-B, 2, 3, 49, 57
Slow-start congestion control, 51
SplitStream, 10, 13, 18
Startup delay, 31, 57, 69
Startup segment, 32, 36, 64
Stripe, 13, 32
SwapLinks, 20

T

Tail of buffer, 32
Time to join, 57
Tree depth, 57, 68, 77

U

Uniform distribution, 59

Z

ZigZag, 9, 16