University of Alberta

Design and Implementation of an Incentive-Aware Peer-to-Peer System for Live Streaming

by

Tianhao Qiu ©

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

Edmonton, Alberta
Spring 2007

Your file Votre référence
ISBN: 978-0-494-30006-0
Our file Notre référence
ISBN: 978-0-494-30006-0

Abstract

In this work, we present a peer-to-peer (P2P) multicast system for broadcasting high bandwidth streams to large numbers of heterogeneous and transient users. A scalable swarm-based P2P scheme is introduced, which does not maintain a rigid logical topology. Instead peers self-organize into an unstructured overlay in an ad-hoc fashion. A credit-based incentive mechanism is proposed to encourage peers to contribute their upload capacity. The proposed scheme is evaluated through simulations in a dynamic and heterogeneous environment. We find that it is able to operate under resource-constrained conditions where traditional tree-based approaches typically fail. At the end, we demonstrate the feasibility of our design by implementing an operational P2P prototype system for live streaming.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Introduction

Live media streaming over the Internet has gained significant popularity in recent years due to the continuous increase in network access speed of the end-users. Most today's Internet broadcast systems are based on the traditional client-server model, which leads to limitations on the achievable performance. The streaming server becomes the bottleneck since the bandwidth required for serving many clients at once is huge and very costly for the broadcasting entity.

Technically speaking, IP multicast [2] is the most efficient approach to support scalable live media streaming. However, the deployment of IP multicast has been slow for a variety of reasons [3]. Application Level Multicast (ALM) [4] has been proposed as an alternative to IP multicast. ALM builds an overlay network via unicast connections between end hosts. Along this direction, many P2P live streaming systems have been developed in recent studies [5, 6, 7].

The end hosts in a P2P streaming system contribute their upload bandwidth to spread streaming media to other peers. The bandwidth requirements are now being shared by the source and participating peers, providing a potentially scalable solution. However, there arises a few practical challenges when we shift the multicast functionality from highly available and dedicated routers to autonomous and vulnerable end-hosts.

- Live streaming usually requires high bandwidth and has rigorous continuity demands. Dynamic changes in network conditions ((available bandwidth and latency), which tend to be more frequent in a P2P environment, can greatly affect the playback quality.

- Extreme peer transience: The measurement study performed in [8, 9] found that a significant number of live streaming users have very short sessions. The average lifetime is only in the order of minutes. Such observation implies that a successful P2P streaming system has to be able to handle a very high rate of failures and dynamics of peer participation (churn).

- Heterogeneity in upload capacity: In P2P streaming, the bottleneck resources can easily be identified as the overall upload capacity in the system. However peers are heterogeneous in how much bandwidth they can contribute. In the current Internet environment, a large percentage of peers rely on asymmetric connections with high receiving capacity and low forwarding capacity. They are likely to consume more and contribute less. Such inherent unfairness requires incentives for other more resourceful peers to donate more bandwidth.

We argue that to address these issues, the design of a successful P2P live streaming system needs the following characteristics: (a) it should maintain a flexible structure to withstand peer transiency; and (b) instead of relying on altruism it should provide incentives to encourage capable peers to contribute their resources.

1.2 Contributions

Our major contributions in this work are three-fold. First, we identify two inherent problems of the traditional tree-based P2P overlay multicast, namely resource underutilization and premature saturation. We argue that premature saturation undermines the sustainability of a tree under constrained bandwidth conditions. To this end we propose a P2P live streaming system based on an unstructured swarm overlay, where each node is connected to a random subset of peers. To fully utilize upload capacities smaller than the stream bitrate, the stream data are divided into small data unit fragments. Peers transact on the basis of concurrent uploads and downloads of unit fragments. In addition, an "informed" push-based scheduling strategy is introduced which helps propagate data efficiently while reducing duplicate transfers.

Second, we propose a credit-based incentive mechanism that is designed to accommodate a certain degree of "unfairness" naturally arising in a heterogeneous environment. Rather than ensuring everyone contributes the same amount of bandwidth, and turning away resource-constrained users, it encourages resource-rich nodes to donate more uplink bandwidth to subsidize resource-constrained peers. Simulation results show that our incentive scheme can motivate capable nodes to upload more because it provides them with better stream quality as well as a higher probability of being placed closer to the source.

Third, to demonstrate that our swarm-based scheme is a feasible architecture for P2P live streaming, a substantial part of our work is devoted to design, implement, and deploy a prototype system based on this architecture. We believe our experience offers a good starting point for others to design and deploy future P2P live streaming systems.

1.3 Organization

The rest of the dissertation is organized as follows: In Chapter 2 we present the background of our research, followed by a brief history of peer-to-peer live streaming techniques. The most relevant systems are categorized based on the architectural models they use. We compare the characteristics of various proposals, focusing on their strengths and weaknesses. In Chapter 3, we concentrate our analysis on the traditional tree-based overlay multicast systems, and further explain its two inherent problems. We then detail the design of a swarm-based P2P live streaming system in Chapter 4. We describe our experimental setup and evaluate the simulation results in Chapter 5. In Chapter 6, we demonstrate the feasibility of our design by implementing an operational prototype system. We describe the aspects that were not addressed in the design phase, and present the modifications to make the system work in a real world environment. Finally, in Chapter 7 we sum up the open problems not covered by this work.

Chapter 2

Related Work

2.1 Why not Wireless?

The seminal work by Gupta and Kumar [1] has seriously put into question the ability of mobile ad-hoc networks to scale to a level applicable for "large" deployment. In simple terms, what [1] demonstrated is that the potential of nodes to communicate with far away nodes cancels performance dividends stemming from spatial reuse. Certain "data diversity" approaches [40, 41] have been proposed as the means to avoid the aforementioned lack of scalability by injecting data traffic only within a restricted number of hops. However, exploiting data diversity requires the development of particular protocols that incite the cooperation of nodes.

In our work presented in [39], we re-examine [41] from the viewpoint of more realistic assumptions. In [41], a single data item is of interest to all the nodes in the system. Instead we assume different per-node interest and a large population of data items (files) with a popularity that follows a Zipf-like distribution. The particular shape of the popularity distribution has been identified in wired P2P networks [42]. To capture the shared nature of local wireless channel, we apply a simplified max-min fair bandwidth allocation algorithm. We also consider dynamic environments where, over time, new nodes join the network and some old ones depart.

Our approach exposes additional issues not originally studied, such as the throughput under different node interests in terms of the files available, or the need for nodes to possess a "healthy" fraction of the total population of files in order for the throughput to be maintained at high levels. As a second step,

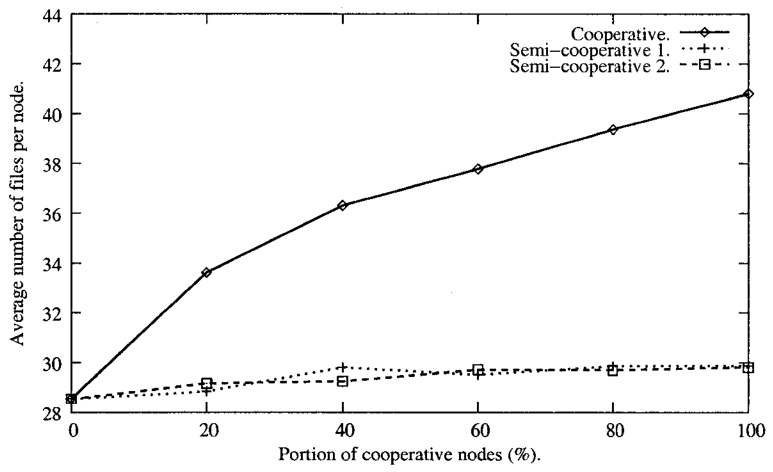


Figure 2.1: Performance of different policies.

we revisit the lessons from [40] and observe the fact that in an environment with restricted levels of mobility, the time horizon over which reception of a desired file can take place may be unreasonably long. We therefore propose how to leverage the design rule (limit the distance in terms of number of hops between communicating/exchanging nodes) but not interpreted at its extreme (exchanges only between adjacent nodes) that [41] assumes. We thus introduce a tradeoff whereby less throughput (due to paths limited to a few hops "congesting" locally the network) comes at the benefit of better delay for obtaining a file. However, before one can reasonably expect nodes to implement such limited-length paths, we need to answer the question of what is the incentive for a node that is not the endpoint of an exchange to participate in such exchange. We attempt to find whether a simple policy can provide the incentive necessary so that nodes participate as intermediate hops.

In the end, three different cooperative policies are studied. A node with the cooperative policy is willing to contribute its resources for any file it doesn't have. A node acting semi-cooperative is only willing to route traffic carrying one of its desired files. The difference between the two semi-cooperative policies is in the way of forwarding the file exchange query.

As shown in Figure 2.1, we find out semi-cooperative nodes do not help at all. During the simulation, most traffic is still direct transactions via one-hop path. When a semi-cooperative node tries to serve as an intermediate node

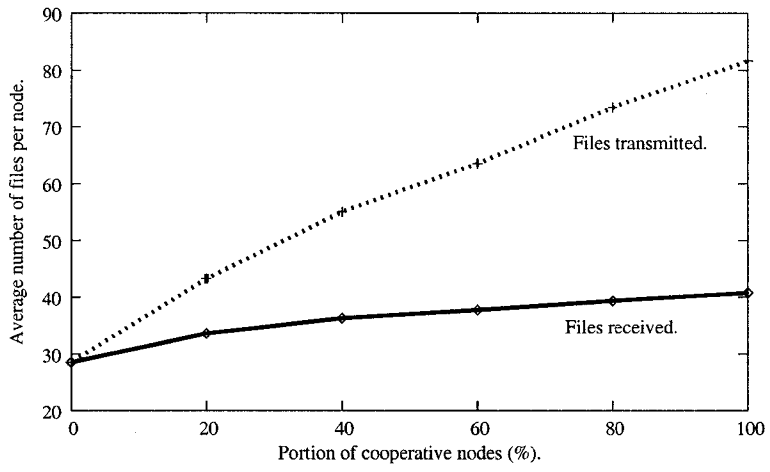


Figure 2.2: Cooperative policy overhead.

by relaying a file exchange request, it will modify the information included in the request according to its own target file list resulting in an even more rigid query. Thus few next-hop nodes can fulfill the more restricted query and initiate a multi-hop file exchange. Since the semi-cooperative schemes are under-performing in a two hop maximum case, they will be performing even worse in multiple hop cases since the query becomes more and more specific to few (and possibly even down to zero) items. Certainly, if the set of files requested in common by all intermediate nodes is the empty set, the query is abandoned (dropped), and no exchange takes place. In environments with immense file populations, and few desired files for each node (compared to the total population of files), the semi-cooperative policies simply make no sense.

Although Figure 2.1 shows promising performance for the cooperative policy, further study shows that this performance increase comes at a price of higher power usage. In Figure 2.2, the number of desired files acquired per node is compared to the number of files transmitted per node. The gap between the two curves represents the overhead introduced by cooperative node behavior. When all nodes in the network are cooperative, the number of files transmitted by a node is almost twice as the number of its desired files. This is not particularly surprising because in addition to one-hop exchanges (that are guaranteed to be relevant to the desired files of the communicating nodes), the nodes are also forwarding traffic for nodes two hops apart. If we

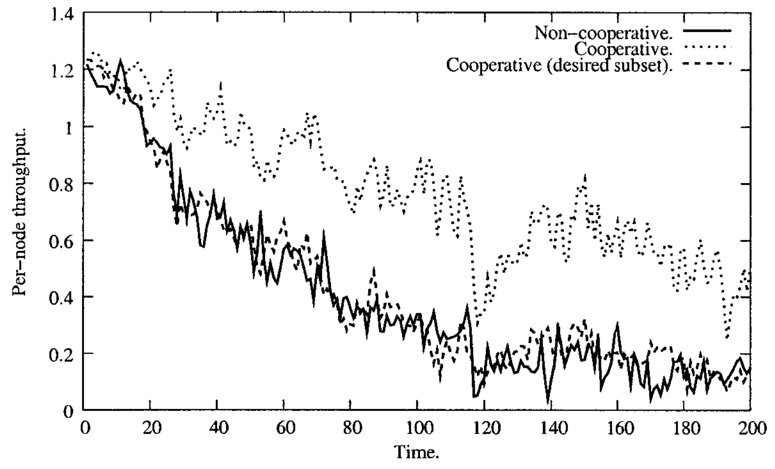


Figure 2.3: Cooperation does not pay off.

allow for longer paths, more of the node's capacity is devoted to forwarding than to data items it desires. Therefore, the gap evidenced in Figure 2.2 could increase even further if longer paths are introduced.

To clarify the last point we will look in more detail into the throughput of a single node, instead of the throughput of the system as a whole. We track the average per-node throughput during the simulation to see if cooperative nodes can enjoy better performance than non-cooperative nodes. 50 cooperative nodes and 50 non-cooperative nodes are placed into the network. Per-node throughput is plotted against time in Figure 2.3. The per-node throughput is expressed as average throughput (counted in units of file blocks) over a period of five successive time units. As shown in Figure 2.3, cooperative nodes have a higher per-node throughput than non-cooperative nodes. But if we only count the part of the throughput that carries desired files, the per-node throughput is indistinguishable to that of the non-cooperative nodes. Hence, the cooperative nodes receive the same "essential" throughput as non-cooperative ones but at the cost of having expended more energy forwarding traffic on behalf of others. This is the disadvantage of cooperation, and raises the question of whether we can seriously use *any* form of multi-hop routing in mobile environments without coming up with a better incentive scheme for nodes to participate in the exchanges of other nodes.

We note that our work in [39] essentially defines some form of peer-to-peer protocols between adjacent nodes in wireless environments . The rather pessimistic results of our experiments motivate us to look back into wired networks instead. In a wired system, a data item can be "discovered" and obtained however far away and however few nodes possess it. Hence, the performance of peer-to-peer protocols in wired environments is no longer so much dependent on user population and data diversity. The simplest case is to assume a single data item is of interest to all the nodes in the system (extending naturally to multiple items of interest to all nodes), where the criteria of success becomes how fast the data are being distributed, namely the system throughput and transmission delay. In the end peer-to-peer live streaming systems stand out as a perfect example for our study since users in live streaming systems naturally share common interest in the data being broadcasted, and the most important measurements in those systems are the average throughput and delay.

In the rest of this chapter we will give a brief discussion of the most relevant peer-to-peer based techniques that have been proposed and deployed in the area of live media streaming.

2.2 Peer-to-Peer Networks

The term "peer-to-peer" (P2P) refers to a class of systems and applications that employ distributed resources to perform a function in a decentralized manner. With the pervasive deployment of computers, P2P is increasingly receiving attention in research, product development, and investment circles. Some of the benefits of a P2P approach include: improving scalability by avoiding dependency on centralized points; eliminating the need for costly infrastructure by enabling direct communication among clients; and enabling resource aggregation [10]. The first popular P2P system was Napster [11] which allowed users to share MP3-files with each other. Since then, the main application for P2P networks has been file sharing, in which users make some files available on their computers and others can download these files.

Although there have been significant research efforts in peer-to-peer systems during the last years, one category of P2P systems has received less attention until recently: the P2P media streaming system [12]. These systems are different in the *data sharing* mode among peers, because media streaming systems use the "play-while-downloading" mode. In file-sharing systems, a client first downloads the entire file before using it. There are no

Table 2.1: P2P file sharing vs. streaming

	P2P File Sharing	P2P Streaming
Mode	Open-after-downloading	Play-while-downloading
Download Order	Our of order	In order
Download Speed Requirement	Only average download speed matters	Require relatively steady download speed
Current Status	Widely deployed and accepted	Not widely accepted yet

timing constraints on downloading the fragments of the file; rather the total download time is more important. However, in media streaming systems, a client overlaps downloading with the consumption of the file. It uses one part while downloading another to be used in the immediate future. Timing constraints are crucial, since a packet arriving after its scheduled play back time is useless and considered lost. Some key diffences between P2P file sharing and media streaming are listed in Table 2.1.

2.2.1 P2P Live Streaming

Live streaming refers to the synchronized distribution of streaming media content to one or more clients. The content itself may either be truly live or pre-recorded. A *live* stream has the important property of being *history-agnostic*: the group-member is only interested in the stream from the instant of its subscription onwards [13].

The first attempts to apply P2P systems for distributing live media date back to 2000 with the proposal of the Overcast [14] and Scattercast [15] architectures. They both employ a two-tiered infrastructure (the P2P part of the network is just the core, end-users don't take an active role in the content distribution) to spread the load from a single server to a large pool of supporting nodes.

In the following years (2001 to the present), the research started focusing on completely distributed systems for media broadcast. Many P2P live streaming systems have been proposed. They can be classified in three main categories based on their overlay architecture:

- Tree-based overlay approaches. In these systems peers are hierarchically organized in a single-tree or multiple-tree overlay, and data are forwarded from the source to peers along the tree(s).

- Unstructured overlay approaches. The content is divided into pieces that are spread in the network by the source without following a predefined structure. Peers establish relationships among themselves to retrieve missing pieces.

- Others. Some systems [18] use a hybrid approach that exploits both previous techniques.

In the following sections we are going to describe these three categories by presenting one or two examples for each of them .

2.3 Tree-Based Systems

In these systems peers are organized in a fairly static structure upon which the stream is spread. This structure is a hierarchical (single or multiple) tree, with the source as root, where every node receives the whole data content from its parents and transmit it to its children.

2.3.1 Single-Tree Approaches

The single-tree model is by far the most common approach to build a P2P live streaming system due to its simplicity. It reproduces the IP multicast structure as peers are organized into a source-rooted spanning tree across unicast connections between them. Many P2P live streaming systems (SpreadIt [5], ESM [4]) have been proposed to create and maintain an efficient tree overlay. Differences between these systems mainly concern the way peers are organized and the algorithms used to create, to maintain and to repair the tree structure.

SpreadIt [5] is among the earliest attempts for streaming live media over a P2P network. It consists of a lightweight *peering* layer that runs between the application and transport layers on each peer, which maintains the connectivity to the rest of the network under arbitrary joins, leaves, and failures of peers in the network. A simple *redirect* mechanism is used to provide hints to a requesting node and guide it to an unsaturated peer in the network. Each join request starts by contacting the source, and then goes down (redirection after redirection) along the tree until a node offers to accept the newcomer. Redirections are done following deterministic policies or at random. When an internal node wants to leave the tree, all its descendants have to be reattached in a new position. Prior to leaving, the node sends redirect messages to its children to gracefully hand them over to other nodes known to have free resources. In case of node failures, the recovery phase is more difficult: first the failure has to be detected (through heartbeat mechanisms, like periodic pings), then all the descendant nodes of the failed parent have to restart a join attempt.

2.3.2 Multiple-Tree Approaches

One of the main issues of the single-tree approach is that the load of distributing the content is supported by a relatively small number of interior nodes while most nodes are leaves and do not share their upload bandwidth. This unfairness issue also leads to the error propagation problem that data loss near the root affects a large population of downstream nodes.

The multiple-tree architecture has been proposed to address these issues by building N different trees, sharing the same source, among peers. One example of the multiple-tree approaches is the SplitStream system [6] from Microsoft Research. In SplitStream, the stream is divided into N disjoint sections called *stripes*, each being didtributed by one tree. To receive the full stream, a node must join every tree. To ensure fairness, the trees are built such that every node is an interior node in precisely one tree. Since all peers are involved in the data distribution, the load is now spread among all nodes. Also a node failure only causes the loss of one stripe, which improves robustness.

A drawback of these systems is the higher control overhead with respect to the single-tree approach, due to the fact that there are now N trees to build and maintain. Another problem of multiple-tree systems is that they are usually tightly coupled with advanced coding techniques like Multiple Description Coding (MDC) [16], which are not widely available yet.

2.4 Unstructured Systems

In these systems peers are no longer organized in a hierarchical structure where the stream is forwarded as a continuous flow of data. Instead the source splits the stream into a series of pieces (often called chunks). There is not a predefined data path since every chunk follows a different route to arrive at peers. The connections established among nodes are "data driven" in the sense that a peer connects to potential providers in order to obtain missing pieces. As a consequence, it is not possible establish a precise time bound for packet reception. It is thus necessary to adjust the stream play out time according to the download rate. This is not a trivial task since the download rate can fluctuate during time. On the other hand, unstructured systems offer good performance in terms of robustness since peers naturally adjust their position in the overlay according to the network changes.

One example of unstructured P2P live streaming systems is CoolStreaming/DONet [17]. In DONet, a gossip membership protocol is used to distribute the knowledge of other nodes in the network. When a node receives a membership message, it will update/create the entry in its membership list. Then it forwards the message to another randomly selected peer to spread the message. A node maintains connections to M partners selected from the peers present in its membership list. Data are exchanged among partners only. Information about what data chunks a peer has is spread among partners using a bitmap. DONet assumes a cooperative environment where nodes upload willingly. Thus no incentives are provided to justify contributions.

2.5 Other Systems

There are some P2P live streaming systems that can not be easily classified in the previously discussed categories because they employ hybrid approaches. These system try to exploit the positive features of both structured and unstructured approaches while mitigating their drawbacks. For example, Bullet [18] combines a standard single-tree structure with an overlay mesh layered on on top of the tree. Bullet nodes begin by self-organizing into an overlay tree, which is used to convey control messages and most data packets. A highly variable mesh structure is then constructed among peers via random connections that are orthogonal to the tree. The mesh overlay enables nodes to quickly locate multiple peers and retrieve missing data items from them in parallel. Thus Bullet is able to avoid bandwidth bottlenecks in the tree by allowing leaf nodes to forward data as well. However since it still relies on the tree to disseminate most data, Bullet does not completely address the inherent issues in the single-tree structure. Namely a node's position

Table 2.2: General comparison of various P2P live streaming systems

	Transmission Delay	Robustness / Adaptiveness	Fairness	Control Overhead
Structured	Low	Bad	Bad - Fair	Low - Medium
Unstructured	Variable	Good	Good	Medium - High
Hybrid	Medium	Fair	Bad	High

in the tree still determines the bandwidth it is likely to give back and the performances it can obtain. Also there are no incentives for Bullet nodes to contribute their resources.

2.6 Conclusions

In the previous sections we briefly discussed the most relevant results in the area of peer-to-peer live media streaming. It is not easy to make a comparison among them since the evaluation results would depend a lot on the assumptions, which include (but not limited to): peer behavioral model, network topology, availability of upload bandwidth, nature of the media stream, and the application environment (i.e. small-scale, intra-company, large-scale, etc.).

In Table 2.2 we make a rough comparison between those different approaches. Structured systems are able to achieve optimal performance with respect to the transmission delay, but they suffer badly in the presence of peer transiency. Unstructured systems show a better resilience to peer transience. However the transmission delay is unpredictable due to the dynamic overlay. Hybrid systems seem like an overall balanced solution. However they require higher control overhead and higher management complexity since they have to maintain both a tree structure and a mesh overlay.

There is not a "best" approach. Every solution has its advantages and its drawbacks. An approach may thus be suitable or not depending on the goals the system needs to achieve and the environment where it will be deployed.

Chapter 3

The Case Against Tree-Based Overlay Multicast Systems

We have seen that a tree-based application level overlay is by far the most popular, obvious and intuitive structure in P2P live streaming systems. In this chapter, we will analyze the tree structure in detail and identify two inherent problems of the tree-based overlay multicast systems, namely resource underutilization and premature saturation.

As a rigid structure, a tree is inherently vulnerable to network dynamics. Unlike IP multicast, the non-leaf nodes in the tree are autonomous end hosts, which can crash or leave at will. When a non-leaf node leaves the multicast tree, its descendants, possibly a significant number of nodes, will suffer stream discontinuity until they find new parents. In addition, the bandwidth available to any host is limited by the bandwidth available from that node's parent in the tree. As a result, any data loss or bandwidth fluctuation at a node near the root may affect a large population of downstream nodes.

Another difficulty with trees is that only the interior nodes are responsible for forwarding traffic, while the leaf nodes do not upload at all. This unfair sharing of load can cause a lack of incentives for interior nodes to contribute. Moreover, if a peer's upload capacity is below the stream bitrate, it can only join the tree as a leaf node. For high bandwidth streams where only a small percentage of peers can forward the stream at full rate, the tree may easily become saturated.

3.1 Resource Index

Here we introduce a metric called *resource index* that measures the service capacity of the system. The resource index RI is defined in [19] as the ratio of the aggregate supply of upload bandwidth (from the source and peers) to the total demand for bandwidth in the system, which is the number of peers times the stream bitrate. RI captures the theoretically available resources the system can offer to participating peers. However in a single-tree structure, the amount of bandwidth that can be utilized not only depends on the capacity of hosts, but also is related to the stream bitrate. For example, if the stream is encoded at $512kbps$, a host with a upload capacity lower than $512kbps$ cannot contribute any resources. On the other hand, if a host has an upload capacity of $768kbps$, this host can support at most one child, which leaves a residual capacity of $256kbps$ unusable. To reflect this resource underutilization of tree structure, we define another resource index RI_{tree} by only considering the upload bandwidth usable to construct a tree for a particular stream bitrate. Evidently, RI_{tree} is always smaller than RI.

To calculate the value of RI_{tree}, we need to model the distribution of upload capacity of Internet end hosts. To simplify our analysis without loss of generality, in this work we divide hosts into two categories based on their upload capacity: *resource-rich* and *resource-poor*. *Resource-rich* hosts have a upload capacity exceeding the stream bitrate, representing users in academic institutions and large corporate entities. *Resource-poor* hosts have a upload capacity lower than the stream bitrate, mostly home users using ADSL or cable modems. Under this bimodal distribution of upload capacity, if each resource-rich host, including the source, is able to support d children on average, then a tree's resource index $RI_{tree} = d \times \alpha$, where α is the percentage of resource-rich hosts.

RI_{tree} plays an important part in determining a tree's scalability and stability. When RI_{tree} is well above 1, there are plenty of idle resources available in the system to support newcoming participants. Furthermore, after an internal node leaves the tree, its descendants can quickly find new parents and recover from disruption. As RI_{tree} drops closer to 1, the system becomes more resource-constrained. As a result, the tree has to grow deeper to accommodate the same number of nodes. The average recovery time for a disconnected node also increases. If RI_{tree} is smaller than 1, there are not enough resources in the system to support the current number of participants. New join requests will be blocked and the descendants of a failed node may

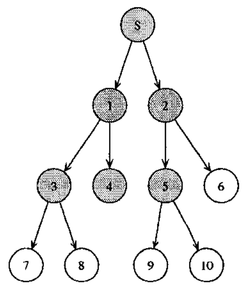


Figure 3.1: A well-balanced tree

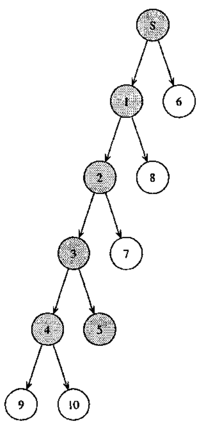


Figure 3.2: An unbalanced tree

not be able to reconnect to the already saturated tree.

3.2 The Impact of Resource Imbalance

A favourable tree-overlay should be balanced to minimize the path length from the root to leaf nodes because a longer path is more prone to failure and congestion. Another benefit of a balanced structure is that each node departure affects a smaller number of descendants on average. However it is not easy to maintain an efficient tree structure due to the existence of resource-poor hosts.

Figure 3.1 shows a well-balanced tree structure, where shaded circles represent resource-rich hosts (note $d = 2$ and $RI_{tree} > 1$). Figure 3.2 shows an unbalanced tree structure with the same set of hosts. We notice the tree becomes unbalanced in Figure 3.2 because there are a few resource-poor hosts (node 6 and 8) connected at higher levels of the tree. As a result, other resource-rich hosts near the root (node 1 and 2) become critical points in the tree as their descendants consist of most of the population. For example, when node 1 leaves, 8 nodes will be disconnected in Figure 3.2, while only 4 affected in Figure 3.1.

Resource-poor peers near the root also cause a tendency for the tree to become prematurely saturated in a dynamic environment. A tree is saturated when every node in the tree is unable to accept more children, leaving some other nodes unable to join the tree. Ideally a tree should only become

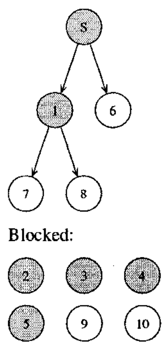


Figure 3.3: Premature saturation

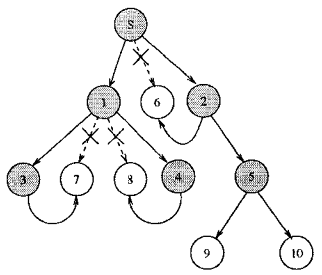


Figure 3.4: Preemption avoids premature saturation

saturated when $RI_{tree} < 1$; when $RI_{tree} > 1$ all participants should be able to join the tree. However due to dynamics of peer participation, it is possible for a resourceful tree ($RI_{tree} > 1$) to become saturated prematurely, causing a large portion of nodes including some resource-rich hosts disconnected. Premature saturation happens when some critical resource-rich host near the source fails and its position is taken over by another resource-poor host. An example of premature saturation is shown in Figure 3.3, which can be formed following a link failure between node 1 and node 2 in Figure 3.2. Premature saturation causes a damaging effect to the tree-based system. It happens more frequently in a resource-constrained environment where there are a lot of resource-poor hosts in the system and the tree tends to be more unbalanced.

3.3 What about Preemption?

An apparent solution to avoid premature saturation is preemption. Preemption allows disconnecting resource-poor hosts from the tree and replacing them by incoming resource-rich hosts [19]. An example of preemption is shown in Figure 3.4. With preemption, resource-poor peers will eventually be pushed further away from the source, resulting in a more balanced tree. Preemption can also serve as an incentive scheme to reward contributing hosts [20].

However, a practical design of preemption has to be able to prevent cheating when identifying contributing hosts because users tend to deliberately misreport information if there is an incentive to do so [21]. If a node declares it has more resources than it has and attempts to accept more children than its capacity allows, it affects not only its own performance, but also its children's performance. We may try to measure and verify reported information from peers, but it requires expensive techniques to automatically estimate the outgoing bandwidth of hosts. Another question is how to ensure that a peer actually contributes its promised donations. A peer can cheat by accepting children but sending little data to them. We may catch such cheaters if children are allowed to audit their parent's behaviors and complain when they believe they are being "mistreated". But this leaves the possibility of fake complaints and collusion.

In this chapter, we have identified two key issues in a tree-based overlay. One is whether there are enough resources to sustain a scalable tree in a heterogeneous environment. Another issue is whether it is feasible to maintain a stable and connected tree in the presence of peer transience. We find that a tree-based overlay is susceptible to premature saturation in a resource-constrained environment. This observation poses serious questions on whether a tree-based scheme is feasible to support high bandwidth streams in the current Internet environment. To this end we seek to design a P2P live streaming system that does not rely on a rigid structure like a tree.

Chapter 4

Design of A Swarm-Based P2P Live Streaming System

4.1 P2P Swarming

Recently, an unique peer-to-peer content distribution mechanism has become very popular, which is sometimes called P2P swarming or file swarming. The technology is most commonly implemented in the BitTorrent protocol [22], though other variations have also been proposed [23]. Unlike many peer-to-peer systems, a P2P swarming system does not maintain a structured overlay. Instead, nodes self-organize into an unstructured overlay in an ad-hoc fashion whereby each node is connected to a random subset of neighbors, as shown in Figure 4.1. A swarm topology resembles a random graph, and thus is robust against partition even in the presence of high rate of churn [24, 25].

P2P swarming is commonly used for distribution of large static files. The file to be distributed is broken into small units, usually $256KB$ to $1MB$ in size. Initial copies of these copies are distributed by the source among randomly selected peers. At the same time each peer periodically exchanges unit availability information with its neighbors, and transacts with them to upload and download data units in parallel. After all the units are downloaded, a peer can re-construct the original file.

It has been shown, both in the literature [26] and in real-world applications [27], that P2P swarming content distribution is more efficient than traditional client-server and CDN approaches with respect to the utilization of upstream capacity. And we argue that some unique features of swarming

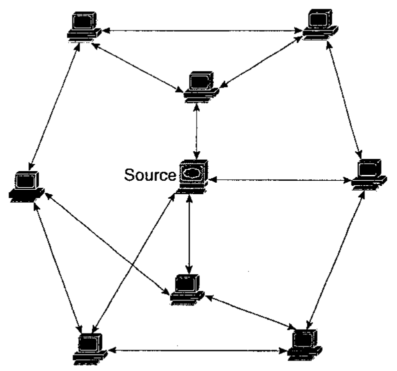


Figure 4.1: Swarming: Peers form a random overlay

could make it a better live streaming solution than a tree-based one in situations where many hosts have asymmetrical bandwidth and short sessions:

- Splitting content into small units helps to fully utilize small granules of available service capacity. Resource-poor hosts that cannot support children in a tree are now able to act as suppliers in a swarm and contribute their upload bandwidth. In circumstances where there are barely sufficient resources, especially at the point when $RI > 1 > RI_{tree}$, a swarm is able to sustain the demands of all participants when a tree cannot.

- In a tree a peer receives data from one single supplier - its parent. In comparison, a peer in a swarm downloads from multiple suppliers in parallel, which enables it to better cope with bandwidth fluctuations and recover from loss of supplier(s) with less quality degradation.

- The bidirectional property of data transfers in a swarm makes it feasible to implement a low-cost and robust incentive scheme, as shown by the effectiveness of the tit-for-tat variant policy used in BitTorrent [22].

However P2P swarming is originally designed for file downloading only. It is not trivial to extend it to support live streaming because live streaming has some distinct characteristics that put more stress on the system.

- One of the reasons that swarming achieves high throughput is that data units are distributed out of order. The most commonly applied algorithm is the rarest-first policy [22], where the least duplicated data unit among neighbors is uploaded first in order to improve data diversity . However, live streaming requires data to be accessed sequentially. If every peer tries to acquire data in strict streaming order, the overall performance of the swarm will deteriorate due to overlapping data sets among peers.

- Compared to file downloading, live streaming is less tolerant toward an insufficiency of resources. A lower resource index only causes a longer waiting time in file downloading, but it seriously degrades the playback quality in streaming. If peers find the stream not watchable, they may decide to leave.
- Only the average download speed matters in file downloading. But for streaming, it is also important to receive the data at a relatively constant rate because the playback quality is subject to oscillations in download speed. Also to ensure the "liveliness" of the stream playout (i.e., to keep it as close to the "live" transmission instants at the source) the delay to disseminate a unit to all the peers must be kept relatively small.
- In file swarming, after the source has distributed a full copy of the file, it functions as a normal peer and may leave without damaging the system. For live streaming, the source remains as the only "seed" in the system throughout the broadcast session. If the source fails to distribute enough copies of some data units, those units may not be able to traverse the whole overlay in time.

We now start to present the design of our swarm-based P2P live streaming system in details. Note that some operations of the protocol are common in generic file swarming so we describe them concisely.

4.2 Membership Management

Like most P2P swarming systems, a peer joins by connecting to a number of peers that are currently in the swarm. We assume that there is a third-party membership service that provides random knowledge about current participants. It helps peers to form a random-graph togology that is robust against partition even in the presence of high rate of churn [24, 25]. There are some existing techniques to implement such a membership service, like the central tracker used in BitTorrent, or by a distributed mechanism, either flood-based (like Gnutella) or gossip-based [28]. Upon being queried by a joining peer, the membership service returns a random subset of m peers that are currently in the system.

With the help of the membership service, each peer, including the source, tries to maintain connected to at least m neighbors. The neighbor list is continuously updated during the session to accommodate membership dynamics. When a peer leaves the system, it will notify the membership service and its neighbors. When the number of neighbors connected to a peer becomes less than m due to departures, the peer will query the membership service again to discover some new neighbors.

4.3 Buffering

The live stream content is broken into fixed size data units at the source. Each data unit is assigned a zero-based index number according to its position in the stream. We observe that peers are more interested in what is being broadcasted "now", and therefore are loosely synchronized with the source in a broadcast session. At any time, a peer is only interested in a small continuous window of the stream depending on the latency between itself and the source. Thus only a small size buffer is needed to store the data units that are inside a peer's current window of interest. As shown in Figure 4.2, a peer's window of interest scrolls at the same rate of the stream as the oldest data unit is being removed and later consumed by the media player. If a data unit is still missing upon its removal, it will cause a discontinuity at playback.

The size of the buffer determines how long a lag there is between a peer's playback and the broadcast time at the source. A smaller buffer size produces a more "lively" stream but at the risk of causing higher data loss rate. On the other hand, a bigger buffer size improves the chances of obtaining data in time thus produces smoother playback. In this paper, we assume that each peer can tolerate an one-minute lag behind the broadcast time and as such allocates a corresponding size of buffer.

The availability of data units in the buffer of a peer is represented by a bitfield along with the index number of the oldest unit in the buffer. When two peers establish a connection, they exchange their bitfields during the

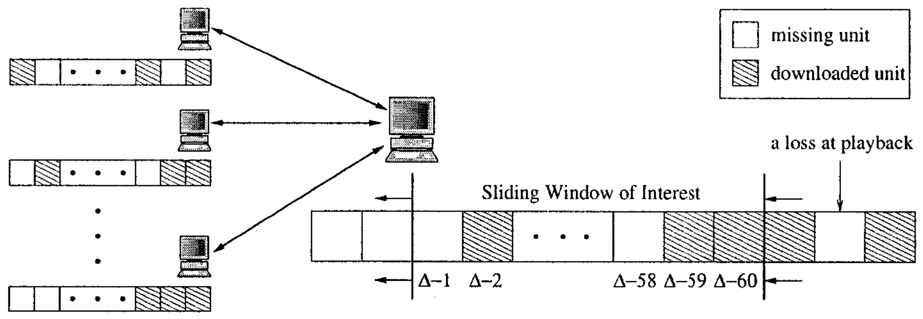


Figure 4.2: A peer's buffer is controlled by a sliding window. Δ is the index number of the latest data unit at the source.

handshaking. Each time a peer obtains a new data unit, it announces the availability to its neighbors by sending a *have* message, which includes the index number of the newly downloaded unit. After receiving a *have* message from a neighbor, a peer updates that neighbor's bitfield accordingly. It should be noted that the overhead of exchanging *have* messages is very small. Assume each data unit contains one second of media data, which implies that a peer generates *have* messages at an average rate of one message per second. Suppose the size of a *have* message is 40 bytes, the upload/download bandwidth consumed by *have* messages is only about $16kbps$ for a peer connected to 50 neighbors. Such a small overhead is negligible compared to the stream bitrate.

The buffer is empty when first allocated. Therefore it is wise for a newly connected peer to defer playback until its buffer is filled up with contiguous data units to the average level of its neighbors. We define startup delay as the delay between the time a peer connects and the time it actually begins playback.

4.4 Credit-Based Incentive Scheme

Like many peer-to-peer applications, we allow users to configure how much upload bandwidth they are willing to contribute. Sometimes limiting the amount of upload bandwidth to contribute is necessary to avoid impeding other ongoing activities sharing the same connection. We also assume that users will not specify values exceeding the upload capability of their network connections. Note that it is not in a peer's own interest to set its upload bandwidth to the physical capacity because doing so will likely lead to congestion on the upstream link, which in turn hurts its download speed.

Realizing that peers are heterogeneous in their upload capacity, we do not require everyone to contribute a minimum amount of bandwidth to be admissible into the system. Instead, we aim to design an incentive scheme that encourages resource-rich peers to contribute more upload bandwidth and subsidize resource-poor peers. Ideally, resource-poor peers are to be served normally as long as there are sufficient resources available. But once resources become critical, the incentive scheme should allow peers that contribute more upload bandwidth to receive better playback quality in order to encourage them to continue their contribution.

A candidate solution is apparently the tit-for-tat upload algorithm described in [22], which has been proven to work pretty well in real-world networks as shown by the success of BitTorrent. The algorithm works in a fully distributed way by letting each peer work independently to maximize its own download rate. A peer selects a fixed number of neighbors to upload while "choking" others. It decides which neighbors to upload strictly based on the current download rates it receives. As a result, peers that upload more tend to have higher download rates as well. Once a peer has finished downloading the whole file, depending on its level of altruism it either leaves or becomes a "seed" to continue uploading.

The above incentive scheme has the important property of being *history-independent*. When a peer determines which neighbors to reciprocate, it does not consider history transactions or long-term transfer rate. Instead only the instantaneous transfer rate is taken into account in order to maximize the download throughput. However, in live streaming the average download rate a peer can achieve is restricted by the stream bitrate. Once a peer has obtained all the data units currently available in the system, it stops downloading until it slides its window of interest to accommodate a new data unit generated at the source. With a history-independent incentive scheme, an adversarial move during such transient periods is to stop uploading as well because there is no direct reciprocity to continue uploading. Such selfish moves could cause underutilization of resources and fluctuations in download rates.

In order to encourage "seeding" behaviors from (potentially adversarial[1]) peers, we introduce a credit-based incentive that is designed to be more

[1]"Adversarial" in this context means a peer with selfish behavior tries to maximize

history-dependent. A peer assigns a credit (CR) to each neighbor to guide future reciprocation decisions. A neighbor's credit is continuously updated solely based on the net amount of data the peer has received from that neighbor. When a peer receives a data unit from neighbor i successfully (before playback), it increments neighbor i's credit accordingly: $CR_i = CR_i + 1$.

With a history-dependent credit in effect, an adversarial peer cannot expect immediate rewards from short-term cooperative behaviours. Therefore it is encouraged to continue uploading to accumulate its credits regardless of the progress of its downloading. However, an ever increasing credit could bring another problem to the incentive scheme. Having uploaded a certain amount of data to a neighbor, an adversarial peer may decide not to upload anymore data to that neighbor because it has accumulated enough credits to secure reciprocations from that neighbor. Also in a dynamic environment, newly connected peers will find themselves in an adverse position to compete with peers that have been around for a while. Thus we introduce an aging factor β into credit computation to reduce old credits in the process. Each peer periodically (default is once every second) updates its neighbor's credit by: $CR_i = \beta \times CR_i, 0 < \beta < 1$.

Each newly connected neighbor receives an initial credit $\epsilon \geq 0$. The value of ϵ determines the competitiveness of a new neighbor. Note that a big ϵ can undermine the effectiveness of the incentive scheme. In particular, if ϵ is higher than its current credit, an adversarial peer is tempted to *whitewash* itself [29] by acquiring a new identity to avoid the penalty on free-riding. In this paper, new neighbors receive an initial credit $\epsilon = 0$ to impose a penalty on newcomers and discourage whitewashing.

In summary, the algorithm to calculate neighbor i's credit can be described as following:

- $CR_i = 0$ (initially);
- $CR_i = CR_i + 1$, when successfully receives a data unit from i;
- $CR_i = \beta \times CR_i, 0 < \beta < 1$ (periodically).

Note that our incentive scheme is immune to cheating and collusion as peers make decisions strictly by observing the behaviors of their neighbors. As a result, malicious peers can not benefit from colluding because the incentive scheme only relies on local states.

its gains while still playing according to the rules of the P2P protocol, i.e., by exploiting weaknesses of the protocol.

4.5 Informed Push-Based Scheduling Assisted by Feedback

A common feature of many multicast systems is that data delivery is scheduled in a push-based fashion: Data are forwarded without explicit requests from the receivers in order to reduce overhead and accelerate the propagation of delay-sensitive data. Therefore a push-based scheduling is more desirable for broadcasting live stream to a large set of dispersed nodes in a short time limit.

However, there are a few disadvantages to a purely push-based scheduling. It is difficult to recover from data loss if the receiver is unable to ask for retransmission of lost data. The reason we would prefer retransimission in a live streaming environment is due to the one-minute buffer size, which gives us enough time to benefit from retransimission. Another disadvantage of push-based scheduling happens when there are multiple suppliers for a given receiver. Uninformed push may create duplicate transfers, which is particularly undesirable in a resource-constrained environment.

We devise an informed push-based scheduling strategy assisted by feedback from receivers, which works as follows. A peer always uploads to a fixed number of neighbors depending on its upload capacity. [2] When it finishes uploading a complete data unit, it selects the next recipient based on its neighbors' credit rating: the neighbor having the highest credit will be chosen. Once a peer chooses a neighbor to reciprocate, it compares its own bitfield with the neighbor's bitfield to determine a subset of data units it is able to offer. The particular unit to be uploaded is then selected from the subset in a random fashion. Note that we do not employ a rarest-first selection policy because a rarest first policy tends to postpone transferring units imminent to playback since those units are usually more common in the system. This can cause unnecessary playback discontinuity. On the other hand, always selecting units imminent to playback might yield better playback continuity, but doing so can undermine data diversity in the system, resulting in a lower throughput because the performance of a swarm-based system is directly subject to data diversity. We find that simply selecting

[2]We assume that the underlying transport protocol is able to saturate the upload capacity and control congestion as the same time. In practice, a variant of TCP Friendly Rate Control (TFRC) protocol [30] may be used instead of TCP to avoid the abrupt changes of the sending rate, which are unfavorable in a streaming application.

unit at random is a good compromise between data diversity and playback continuity.

With informed push, recovering from data loss caused by node failures is an automatic process. However there is still a possibility that a peer receives duplicate data from multiple suppliers, especially when only a few data units are still missing from its buffer. When a duplicate transfer happens, the receiver intervenes by sending back a *feedback* message, instructing the supplier to stop and instead send another data unit suggested in the feedback.

Because the source receives no upload, all its neighbors have a zero credit. To ensure fairness the source uploads to its neighbors in a round-robin fashion. Also the source always pushes out the latest data units with higher priority. This is done to facilitate in-time data delivery and avoid late data loss.

Chapter 5

Evaluation

5.1 Experimental Setup

We developed a flow-level, discrete-event simulator to simulate the P2P live streaming network. The simulator models the pairwise latency between peers and it does not model packet losses and cross traffic for simplification. Without considering a physical network topology, we assume all nodes reside at the edge of the network and no bottleneck links exist in the core of the network. This simplification implies that congestion only happens at the access links to the network [31]. In a P2P streaming system, the bottleneck resources are indeed the limited upload capacity of end hosts. Therefore our simulator only models congestion at those outbound links, which is done by fairly divide available bandwidth between concurrent flows leaving a node.

5.1.1 The Network

The simulated network consists of 1740 nodes, with a pairwise latency matrix derived from measuring the inter-node latencies of 1740 DNS servers using the King method [32]. For simulations involving larger networks we assign each node a random pair of coordinates on a two-dimensional Euclidean space and derive the network delay between a pair of nodes from their corresponding Euclidean distance. The average and maximum round-trip delay between node pairs in both the King data set and the Euclidean plane is about 180 ms and 800 ms respectively.

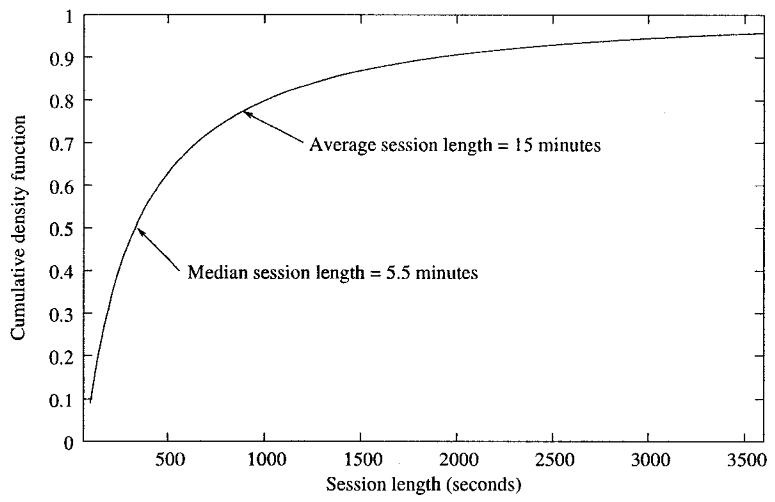


Figure 5.1: Cumulative distribution of session length

5.1.2 Peer Membership Dynamics

Based on the measurement study of live streaming workloads in [8, 9], we model the interarrivals of newly joined peers by an exponential distribution. The average arrival rate is represented by λ. After a peer joins the broadcast, it stays in the system for a random period of time before it leaves. Such period of time is defined as the session length, which follows a log normal distribution shown in Figure 5.1. Note there are a significant number of very short sessions and the median length is only about 5 minutes. However the mean length is around 15 minutes. This implies a heavy tail where a few peers tune into the broadcast for much longer periods of time.

5.1.3 Streaming Media Model

We assume the source is broadcasting a constant bitrate stream encoded at $512kbps$. The stream consists of a sequence of media packets. Each packet contains the compressed media data spanning across a short time period, which is assumed to be one second in simulations. The loss of a single packet causes the media of that second alone to be not decodable, thus incurs an one-second playback discontinuity. Note that the media packet is interchangeable with the data unit defined in Chapter 4.

5.1.4 Bimodal Configuration of Upload Capacity

We divide peers into two categories according to their upstream bandwidth: (a) *resource-rich* peers have an upstream bandwidth of $2048kbps$, which is four times the bitrate; (b) *resource-poor* peers have a constrained upstream bandwidth of $256kbps$, half the bitrate. Since we only consider congestion at upstream links, peers are assumed to have unlimited downstream capacity. The percentage of resource-rich peers α controls the resource index RI and RI_{tree} in the system as: $RI = 3.5\alpha + 0.5$ and $RI_{tree} = 4\alpha$.

5.1.5 A Tree-Based Protocol

To compare the performance of the swarm-based live streaming protocol with a tree-based one, we implemented a simplified version of the tree overlay protocol borrowed from [19]. Like many existing P2P live streaming approaches, it builds and maintains a single connected tree rooted at the source. When a peer needs to connect to the tree, either at its join time or at the time of a reconnection, it contacts the peer membership service to get a random subset of m peers that are currently in the system. It will then probe the m peers to see if they are currently connected to the tree and have enough capacity to support a new child. If there are multiple positive replies, the peer selects the parent with the minimum depth. This process is repeated until the peer is able to find a parent and connect to the tree. Unlike [19], we do not give higher priority to join requests from potential contributors based on the reasoning that it is very difficult to realize such a preemption scheme in real-world networks (see Section 3.3). Therefore there may be cases where the tree becomes prematurely saturated, leaving some resource-rich peers unconnected and unable to contribute their upload bandwidth. We also employ a different tree repairing method when an internal node fails: Instead of having each disconnected descendant look for a new parent independently, only the children of the failed node try to reconnect while others stay put in the subtrees. Such graft-like method generates fewer reconnection requests. It is also supposed to create a more stable tree structure.

5.2 Simulation Results

Unless otherwise noted, each run of simulation simulates a period of two hours. To allow the system to reach steady state behavior, we start to collect

Table 5.1: Default simulation parameters

Number of hosts returned by the membership service	$m = 20$
Average arrival rate	$\lambda = 1$ arrival per second
Credit aging factor	$\beta = 0.9$ per second
Initial credit	$\epsilon = 0$
Startup delay	30 seconds
Size of the sliding window	60 seconds
Source capacity	2560 kbps

statistics after a warm-up period of one hour in simulation time. Default simulation parameters are listed in Table 5.1.

5.2.1 Performance of the Tree-Based Protocol

We evaluate the performance of the protocols using two important metrics, *average stream quality* and *aggregate system throughput*. Average stream quality measures the playback continuity received at individual peers, defined as the number of data units arrived before playback over the total number of units. Aggregate system throughput measures the performance of the system as a whole, defined as the aggregate instantaneous download throughput over the total amount of demands (number of peers times the stream bitrate).

Figure 5.2 plots the average stream quality as a function of α, the percentage of resource-rich peers. It can be seen that the stream quality remains very low when there are few resource-rich peers in the system. Only after α exceeds 0.25, the stream quality begins to improve to an acceptable level. When half of the participants are resource-rich peers, the tree is able to deliver a near perfect quality stream to all participating peers. Figure 5.2 also shows that the source's capacity is very important in a tree-based protocol. A small fan-out degree at the source has a big negative impact on the performance because the tree is more likely to become unbalanced if a few resource-poor peers happen to be connected near the source.

Figure 5.3 shows the changes in the tree's aggregate throughput over time under different resource conditions. We find that lack of resources causes instability in the tree. When $\alpha = 0.4$, the system is able to quickly recover from

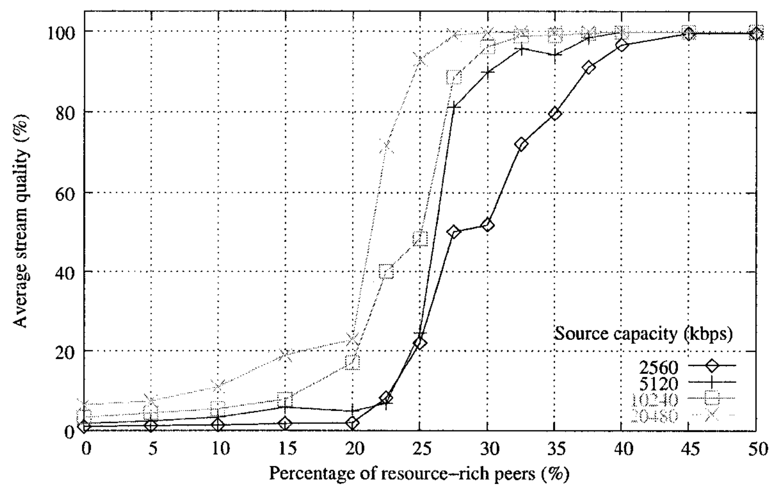


Figure 5.2: Tree's performance as a function of α under different source capacity

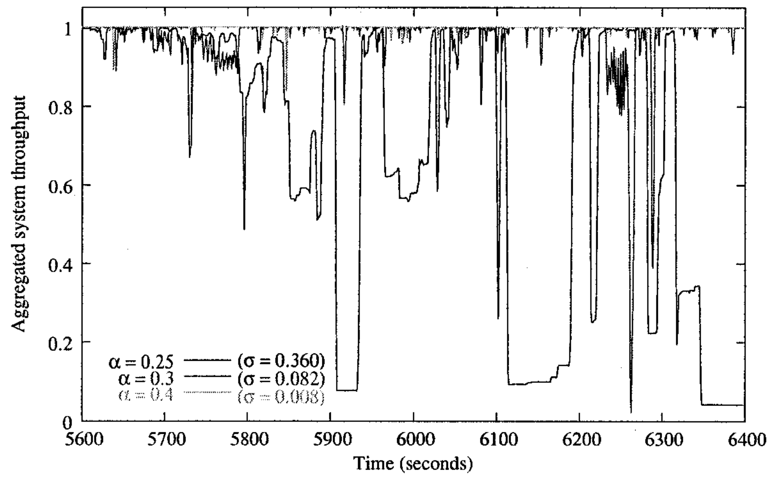


Figure 5.3: Less resources cause more instabilities in the tree

node failures. The sample standard deviation σ of the system throughput over the shown time period is negligible, at 0.008. With a smaller $\alpha = 0.3$, the system becomes more vulnerable to peer transience. Oscillations in throughput occur more frequently and last longer, resulting in a higher standard deviation of 0.08. When $\alpha = 0.25$, the tree begins to experience large-scale and continuous breakdowns. The throughput plunges when a lot of peers are disconnected due to departures occurring near the source. What is worse is that some breakdowns continue for a long period of time if the tree becomes prematurely saturated. Such highly fluctuated throughput is destructive to playback continuity by causing frequent buffer underflows.

Figure 5.4 shows the underutilization of resources in the tree at $\alpha = 0.25$ over time. Total resources (RI) are the total amount of upload bandwidth available in the system. Usable resources (RI_{tree}) are the amount of upload bandwidth that can be used to construct the tree. The difference between RI and RI_{tree} demonstrates the inherent underutilization of resources in the tree-based protocol because resource-poor peers are unable to contribute their upload bandwidth in the tree. The capacity of the tree is determined by the amount of upload bandwidth from resource-rich peers that are currently in the tree. Apparently the tree's capacity is maximized and equal to RI_{tree} when all resource-rich peers are connected. However as shown in Figure 5.4, the tree's capacity is only maximized in a few cases. Instead it often happens that many resource-rich peers are blocked from joining the prematurely saturated tree, which exacerbates the underutilization of resources.

To further illustrate the instability of the tree structure, we plot the cumulative distribution of disruption periods in Figure 5.5. A disruption period is the interval between the time a peer is disconnected from the tree and the time it finds a new parent, during which the peer suffers playback discontinuity. Figure 5.5 confirms that a tree is much more unstable in a resource-constrained environment. When $\alpha = 0.4$, most disruptions are recovered in less than 1 second; but when α drops to 0.25, more than 30 % of disruptions last longer than 10 seconds .

5.2.2 Performance of the Swarm-Based Protocol

Figure 5.6 and Figure 5.7 show the performance results of the swarm-based protocol. Comparing Figure 5.6 to Figure 5.2, we find that the swarm-based protocol is much more resilient to scarcity of resources than the tree-based protocol. Because a swarm is able to fully utilize all the available resources,

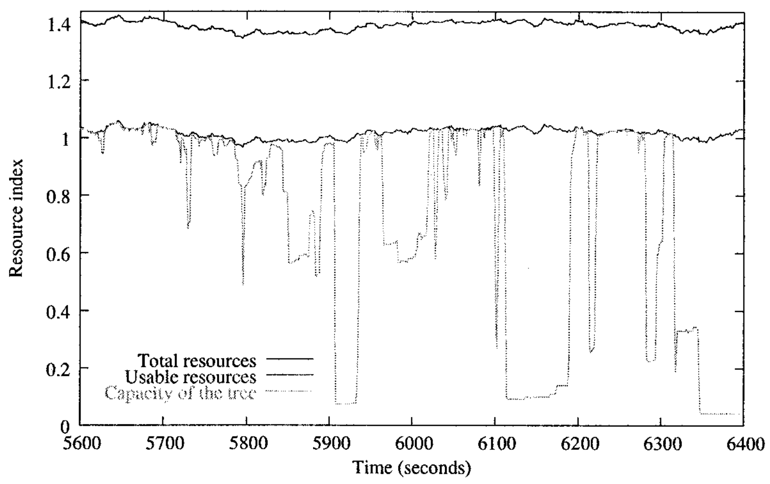


Figure 5.4: Resource underutilization in the tree ($\alpha = 0.25$). Note that the throughput curve in Figure 5.3 closely follows the tree's capacity in this figure, indicating that our tree-based protocol is efficient in locating unsaturated nodes.

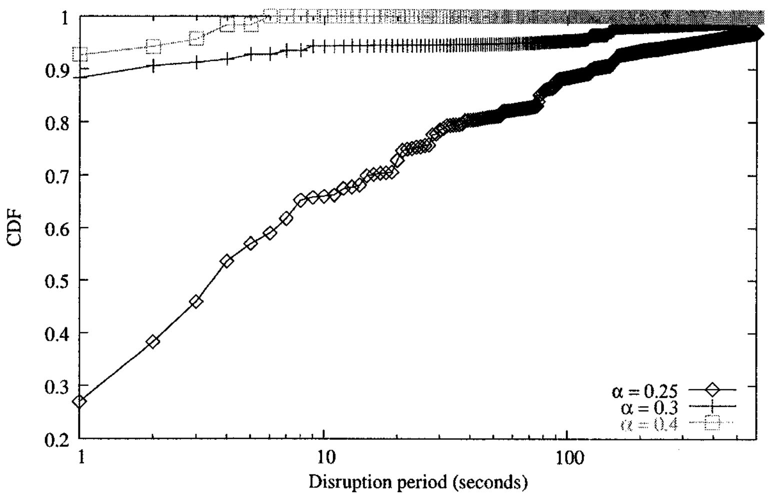


Figure 5.5: Cumulative distribution of disruption periods in the tree

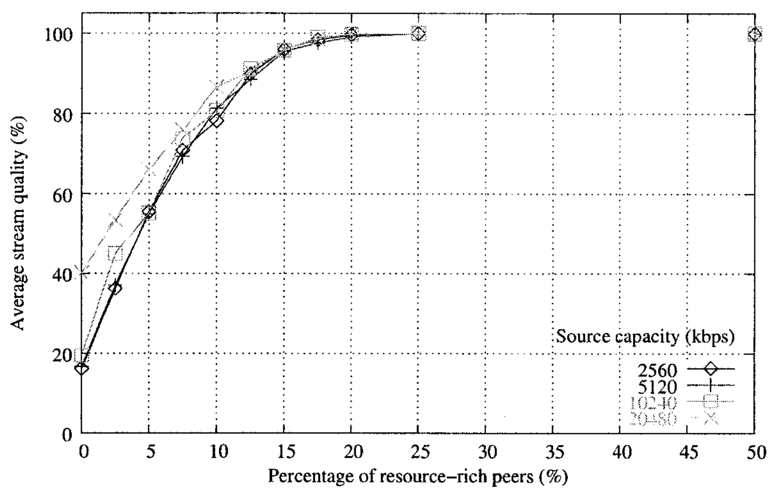


Figure 5.6: Swarm's performance as a function of α under different source capacity

the average stream quality in Figure 5.6 quickly improves as the resource index increases. At $\alpha = 0.2$, the swarm-based system is able to deliver a near-perfect stream while the stream in the tree-based system is still unwatchable (quality $\ll$ 50%). We also find that a swarm requires less source capacity to yield good performance than a tree does. The reason is that the data forwarding load in a tree is carried by a fraction of internal nodes only, while in a swarm all the peers are engaged in distributing data. Therefore the problem of premature saturation, which plagues a tree with limited fan-out degree at the root, seldom happens in a swarm.

Not only can it produce much better stream quality, a swarm is also more stable than a tree in the presence of churn. The reason is that a swarm does not need to maintain a rigid structure. Instead the propagation path of data units is constantly changing to be adaptive to network dynamics. Figure 5.7 plots the swarm's aggregate throughput over time. When $\alpha = 0.25$, the system throughput remains within a very small range below 1 and the sample standard deviation is only 0.003. This is in sharp contrast with what we see in Figure 5.3, where the throughput of $\alpha = 0.25$ fluctuates intensively in an unpredictable way, resulting in an extreme standard deviation of 0.36.

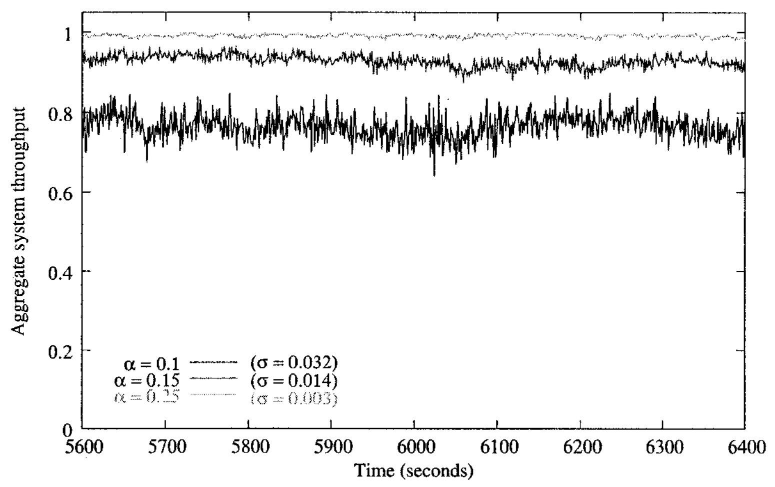


Figure 5.7: More resources bring higher throughput and better stability to the swarm

5.2.3 Scalability of the Swarm-Based Protocol

To study the scalability of the swarm-based protocol, we generate different workloads on the system by increasing the arrival rate λ. Figure 5.8 shows how different network sizes affect the average and the maximum path length to deliver data units from the source to all participating peers. We find that the path length grows very slowly as the number of nodes increases. This is expected because the propagation routes for each particular data unit follow a spanning tree rooted at the source, whose depth grows logarithmically with the size of the network. As shown in Figure 5.8, even when there are ten thousands of nodes in the swarm, the propagation tree still stays within a practical depth, incurring a maximum end-to-end latency in the order of a few seconds if we ignore the waiting at each hop.

In the same set of experiments, we also find that the average stream quality is not affected by the network size in a noticeable way, and we omit the results for brevity. Moreover, because a peer is connected to a fixed number of neighbors regardless of the total number of nodes, the overhead at each peer is also independent of the network size.

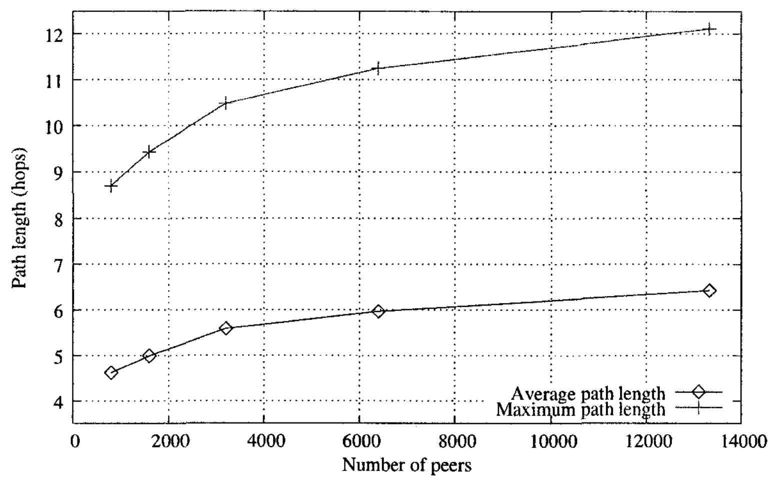


Figure 5.8: Swarm shows good scalability to network size ($\alpha = 0.25$)

5.2.4 Providing Incentives

To investigate the effect of the credit-based incentive scheme, we compare the average stream quality received by resource-rich and resource-poor peers in Figure 5.9. As we expected, the incentive scheme works in an adaptive way, responding differently to different resource conditions in the system. When the resource index is larger than 1, both resource-rich and resource-poor peers are able to receive near-perfect stream quality because there are more than enough resources to sustain the whole population. However once the resource index drops below 1, the effect of the incentive scheme begins to show up: the stream quality received by resource-poor peers plunges, while resource-rich peers are not as remarkably affected. This difference in quality of service becomes more distinguishable if the resource index continues to drop. Therefore, resource-rich peers are encouraged to contribute larger amount of uplink bandwidth, because doing so will lead to better playback quality in the presence of resource variations. Another beneficial effect of the incentive scheme is when the system is short of resources, resource-poor peers are more likely to leave due to poor stream quality, while resource-rich peers are more likely to stay since they see much less quality degradation. As a result, the system will be able to self-recover from resource shortage and restore the average stream quality.

Figure 5.10 plots the average path length of resource-rich and resource-poor peers as a function of α. It is desirable to receive data across shorter

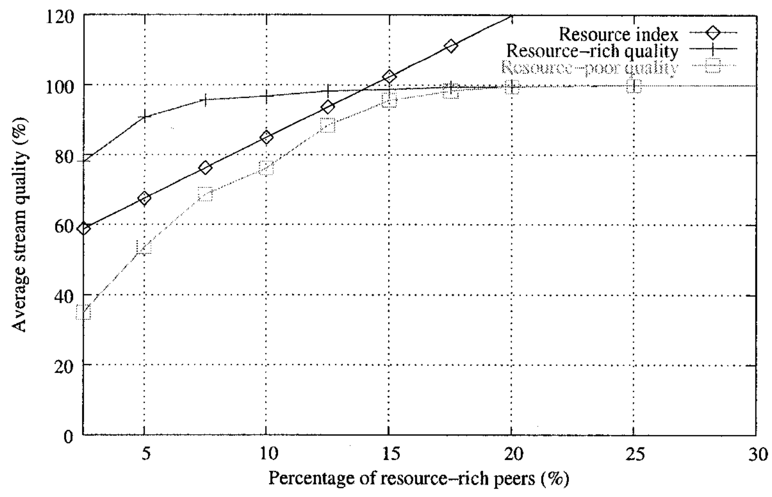


Figure 5.9: Resource-rich peers receive better stream quality under constrained bandwidth conditions

paths, since it reduces the propagation delay and the probability of service disruption. We find that there is a clear correlation between the path length of a peer and the amount of its contributions. Resource-rich peers receive data in fewer hops, which serves as another incentive for them to contribute more uplink bandwidth. This is a natural consequence of the credit-based incentive scheme that favors peers with higher credits. A resource-rich peer accumulate more credits at its neighbors by uploading more. Therefore it is able to preempt other resource-poor peers in the contention for upload slots. Since resource-rich peers have higher out-degree, placing them closer to the source also reduces the depth of the spanning tree.

5.2.5 Effect of Startup Delay

When a peer joins the system, it has an empty buffer and no credit as $\epsilon = 0$. Therefore the data loss rate is likely to be very high before the peer can obtain a few data units and accumulate some credits. Figure 5.11 shows the average data loss rate during a peer's startup time under different resource conditions. With a reasonable resource index ($\alpha = 0.25$), a newly connected peer can expect its buffer to be filled up to the average level in less than 30 seconds by its neighbors. From that point on, the peer is able to receive the stream with nearly no data loss. Therefore a 30-second startup delay is

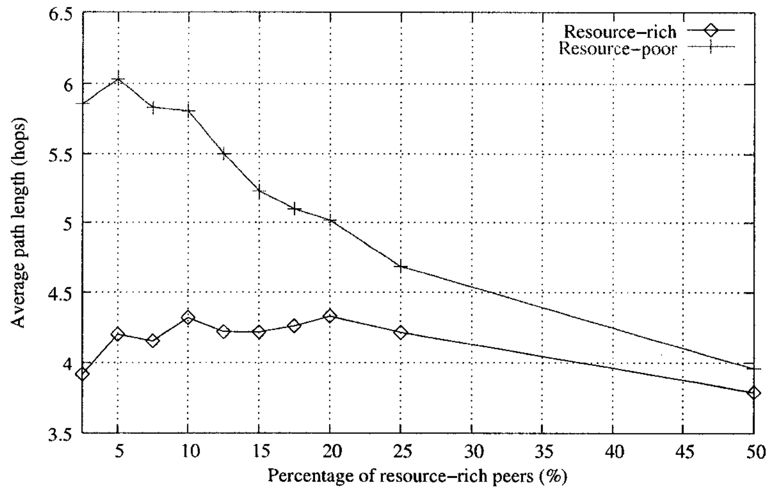


Figure 5.10: Resource-rich peers are placed closer to the source

enough for a new peer to start smooth playback. However we also find that a much longer startup delay is necessary if the resource index gets lower. Such a penalty on newcomers can be partly lifted with a higher initial credit ϵ.

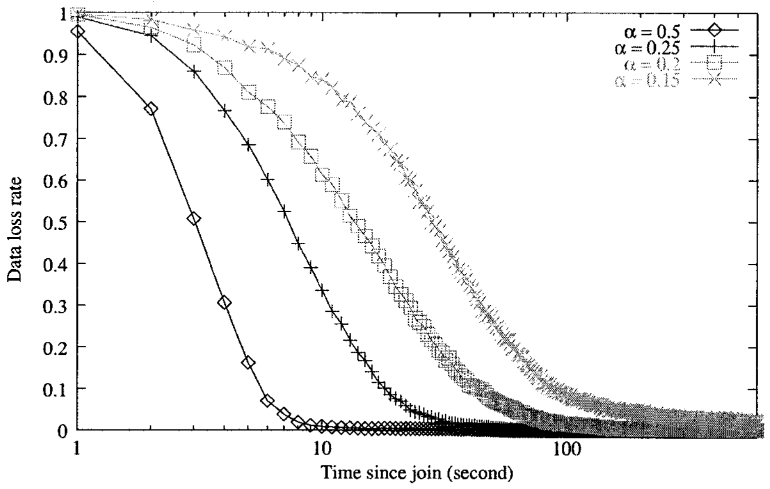


Figure 5.11: Data loss rate during startup period

Chapter 6

Implementation

In this chapter, we proceed to implement a real-world P2P live streaming system based on the proposed design in Chapter 4. Many challenges involved in developing a deployable and operational system are not considered in the design phase. For example, our system needs to support users behind network address translators (NATs) and firewalls, which consist of a large percentage of Internet users nowadays. The system also has to cope with a variety of popular media player and encoder softwares in order to import media data from encoders and provide them for playback in media players. In engineering our system, we have adopted some simple or existing solutions to accelerate the development. As a result, some compromises have been made to use suboptimal approaches with the hope that they can be revisted later. But still significant efforts have been invested to make the system robust and easy to use to meet the requirements of public release.

6.1 System Overview

Figure 6.1 gives a high-level overview of the live streaming system. The encoder converts the raw audio/video data generated by the camera into a compressed media stream, and sends it to the source peer. The source peer cuts the media stream into pieces of fixed size, typically each piece contains roughly one-second of media data. It should be noted that data integrity is not verified in the current implementation. The source peer and receiver peers run the same swarm-based P2P live streaming protocol to disseminate the stream data in the units of pieces along the overlay. Each receiver peer

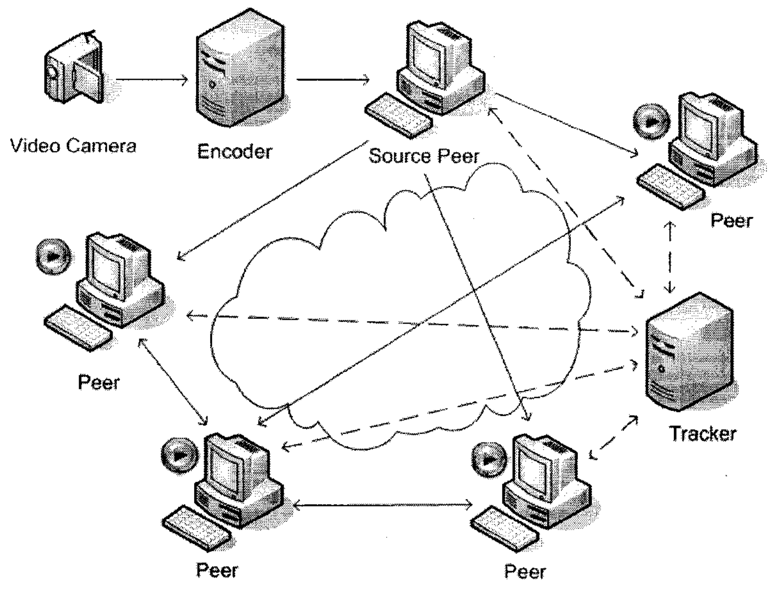


Figure 6.1: System overview

then forwards received data to the media player usually running on the same machine for playback.

The tracker is responsible to maintain a constantly updated list of peers currently in the session. A peer joins the broadcast session by contacting the tracker. The tracker responds with a list of contact information of peers randomly selected from the pool. With the help of the tracker, a random overlay among peers is maintained for each broadcast session. This technique is based on the existing BitTorrent specifications. In fact, an open-source implementation of BitTorrent tracker [33] is used as the codebase to develop our tracker software. In addition to help peers find each other, the tracker is also responsible to gather statistics from peers for both online and offline analyses. Currently, the data being collected include each peer's IP address, upload/download rate, lifetime and average playback quality.

The architecture of our P2P software is depicted in Figure 6.2. The modules inside the broken line are the function blocks of a single peer node. Other than the media import and player service modules, the source peer and receiver peers share the same design and implementation, which simplifies the development and deployment. Each of these modules will be discussed in detail in the rest of this chapter.

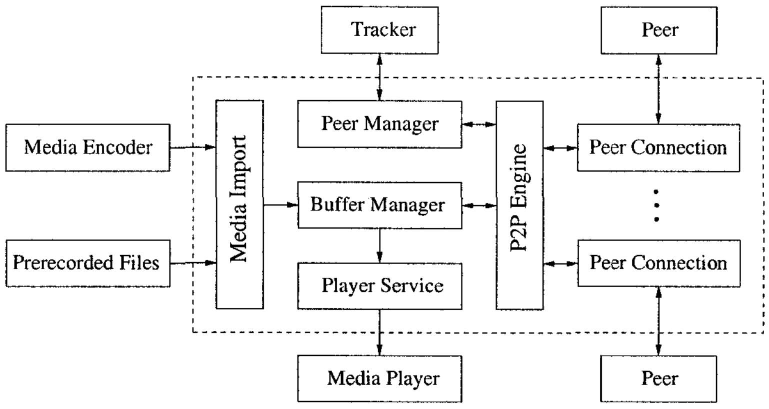
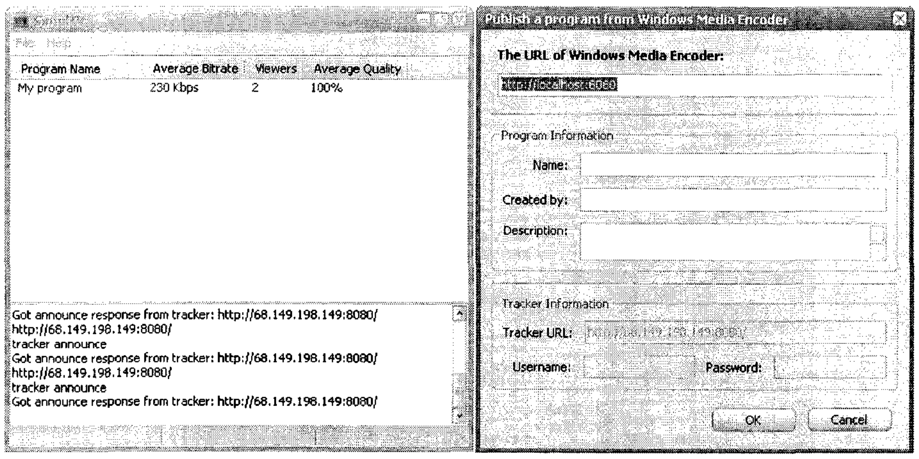


Figure 6.2: Diagram of the P2P software architecture. Arrows indicate data flow.

6.2 Publishing and Joining a Broadcast Session

To start a broadcast session, the source peer (publisher) first needs to define the content to be broadcasted. The content can either be a live event captured by Windows Media Encoder [34], or a list of prerecorded media files. Currently only the ASF file format [35] is supported. The publisher then announces the content by sending a HTTP POST request to the tracker. Upon receiving the POST request, the tracker assigns a program URL to identify the content, which includes essential information like an unique program id, average bitrate reported by the publisher, and some optional parameters like the name and description of the content. The program URL can either be distributed by the publisher to potential viewers separately, or be announced by the tracker to peers upon query.

To join the broadcast session, a user simply clicks the program URL and the installed P2P software will be invoked with appropriate configurations. The peer first sends a HTTP GET request to the tracker via its peer manager module. The parameters in the GET request include the requested program's program id, the peer's peer id, the port number that the peer is listening on for incoming connections, etc. Once receiving a random list of peers from the tracker, the P2P engine module will start to make connections to and accept connections from other peers in the same broadcast session. Note that all connections between peers can transfer in both directions, and are used to



Program Name	Average Bitrate	Viewers	Average Quality
My program	230 Kbps	2	100%

```
Got announce response from tracker: http://68.149.198.149:8080/
http://68.149.198.149:8080/
tracker announce
Got announce response from tracker: http://68.149.198.149:8080/
http://68.149.198.149:8080/
tracker announce
Got announce response from tracker: http://68.149.198.149:8080/
```

Figure 6.3: User interface including the publishing dialog

forward and receive stream data at the same time.

6.3 Peer-to-Peer Protocol

Peers exchange state information and data pieces via the peer-to-peer protoocl, implemented by the peer connection module. In order to have a working prototype as soon as possible, we select TCP as the transport protocol because it is widely available and its congestion control algorithm has been well-tested. We are aware that TCP is not an ideal protocol for real-time streaming media due to late retransmissions and excessive rate oscillations. In the future, we plan to incorporate TFRC [30] into the system, which is a UDP-based congestion control protocol optimized for streaming media.

6.4 Buffer Management and Synchronization

Due to the nature of live streaming, received data quickly become obsolete after being sent to media player while new data are being generated at the source. Therefore a ring buffer is used to store the data pieces that are inside a peer's current window of interest, which constantly moves itself by removing the oldest piece to make space for the newly generated one.

Buffer management is quite straightforward for the source peer, where the buffer manager retrieves pieces from the media import module in a sequential order, and informs the P2P engine module to announce new pieces to its connected peers. At a receiver peer, the buffer manager is also responsible to keep its buffer semi-synchronized with the source peer. This is achieved by monitoring *have* messages the peer receives, and moving its buffer accordingly once an out-of-boundary piece has been announced. If a peer is not connected to the source peer directly, the movement of its buffer tends to fall behind the source peer's schedule. This is actually desirable since a peer only needs to move its buffer when a new piece is available among its neighbors.

6.5 Media Player Interface

We use Windows Media Player as the default media player in our system because of its dominant presence on Windows platforms. For other operating systems including Linux and Mac, we support the VLC media player, available at *http://www.videolan.org/vlc/*. The media player is directed to a fixed *mms://localhost:port* URL served by the player service module, which acts as a unicast streaming media server and reconstructs the original media stream with data pulled from the buffer manager module.

6.6 NATs and Firewalls

NATs and firewalls impose fundamental restrictions on pair-wise connectivity of peers in the overlay. For two peers to setup a connection, at least one of them must be able to accept incoming TCP connections. In most cases, it is not possible for peers both behind NATs or firewalls to communicate directly with one another. To improve overlay connectivity without sacrificing the benefits of TCP, we have included support for the UPnP Internet Gateway Device (IGD) protocol [36] that makes it possible for our application to automatically configure NAT traversal. We also plan to incorporate the STUNT protocol [37] to traverse NATs and firewalls that do not support UPnP.

6.7 Deployment and Experiments

After months of design and development, we have finished a working prototype system, including the tracker and the P2P client application. It is written in C++ and released at [38].

We have conducted some preliminary tests in a lab setting, where nodes are directly connected over a switched 100Mbit network with negligible delays and more than sufficient bandwidth. The prototype system works very well in these small-scale trials with 8-10 nodes involved. Peers are able to play the stream smoothly with little start-up delay. When the upload bandwidth of the source is intentionally limited below the system throughput, peers automatically take over the missing parts and the average playback quality is not affected.

Chapter 7

Conclusions

In this work we present the design and implementation of an incentive-aware P2P live streaming system, which does not maintain a rigid structure. Instead, nodes self-organize in an ad-hoc fashion into a random-graph overlay. An informed push-based scheduling strategy is employed to propagate data efficiently without many duplicate transfers. Our simulation results show that the architecture can scale to a large number of nodes and is resilient to high rate of churn. With the help of a credit-based incentive scheme, our system is able to work in a non-cooperative environments by rewarding contributors while penalizing non-contributing nodes. The main results of this study are that the swarm scheme outperforms comparable tree-based schemes but also provides performance dividends exactly when necessary, i.e., when resources are scarce. In fact, it is under resource-scarce conditions that the incentive scheme provides resource-affluent nodes with a performance advantage that encourages their continued presence and participation.

7.1 Open Issues and Future Work

An important open issue is how to make the swarm topologically aware so that the data propagation path is efficient in terms of metrics such as link stress and end-to-end latency. It is also important for recipients to be able to verify data integrity on the fly during a P2P live streaming session. However existing protocols to enforce data integrity are either expensive or inapplicable to P2P streaming. Among the explored possibilities are reputation-based schemes, and in general schemes that do not have to rely on pre-existing trust relationships.

We intend to investigate the aforementioned issues in our future work. The work we foresee for this project also consists of improving and fine-tuning the algorithms by expanding the test environment in real-world conditions.

Bibliography

[1] P. Gupta and P. R. Kumar. The Capacity of Wireless Networks. IEEE Transactions on Information Theory, vol. IT-46(2), pages 388-404, March 2000.

[2] S. Deering. Multicast Routing in Internetworks and Extended LANs. In Proceedings of the ACM SIGCOMM, August 1988.

[3] C. Diot et al. Deployment Issues for the IP Multicast Service and Architecture. *IEEE Network*, vol. 14(1), 2000.

[4] Y. Chu, S. G. Rao, S. Seshan, and H. Zhang. A Case for End System Multicast. IEEE Journal on Selected Areas in Communication (JSAC), Special Issue on Networking Support for Multicast, Vol. 20, No. 8, 2002.

[5] H. Deshpande, M. Bawa and H. Garcia-Molina. Streaming Live Media over a Peer-to-Peer Network. Technical Report, Stanford University, April 2001.

[6] M. Castro, P. Druschel, A. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. Splitstream: High-bandwidth multicast in cooperative environments. In 19th ACM Symposium on Operating Systems Principles, 2003.

[7] P. A. Chou, V. N. Padmanabhan, and H. J. Wang. Resilient peer-to-peer streaming. Technical Report MSR-TR-2003-11, Microsoft Research, Redmond, WA, March 2003.

[8] K. Sripanidkulchai, B. Maggs, H. Zhang. An Analysis of Live Streaming Workloads on the Internet. In Proceedings of the 4th ACM SIGCOMM conference on Internet measurement, Taormina, Sicily, Italy, Oct. 2004.

[9] E. Veloso, V. Almeida, W. Meira, A. Bestavros, and S. Jin. A Hierarchical Characterization of a Live Streaming Media Workload. In Proceedings of Internet Measurement Workshop (IMW), Nov. 2002.

[10] D. S. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, B. Richard, S. Rolling, Z. Xu. Peer-to-Peer Computing. Technical Report HPL-2002-57, HP Lab, 2002.

[11] Napster Inc. The Napster homepage. http://www.napster.com/, 2001.

[12] D. Xu, M. Hefeeda, S. Hambrusch, B. Bhargava. On Peer-to-Peer Media Streaming. In Proceedings of IEEE-ICDCS'02, Vienna, Austria, July 2002.

[13] M. Bawa, H. Deshpande, and H. Garcia-Molina. Transience of peers and streaming media. ACM SIGCOMM Computer Communication Review, Volume 33, Issue 1, January 2003.

[14] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek and J. W. O'Toole. Overcast: Reliable Multicasting with an Overlay Network. In Proceedings of the Fourth Symposium on Operating Systems Design and Implementation, 2000, pp. 197-212.

[15] Y. Chawathe. Scattercast: An Architecture for Internet Broadcast Distribution as an Infrastructure Service. Ph.D Thesis, University of California, Berkeley, 2000.

[16] V. K. Goyal. Multiple description coding: Compression meets the network. IEEE Signal Processing Magazine, pages 74-93, September 2001.

[17] X. Zhang, J. Liu, B. Li, and T.-S. P. Yum. DONet/CoolStreaming: A Data-driven Overlay Network for Live Media Streaming. In Proceedings of IEEE INFOCOM'05, Miami, FL, USA, March 2005.

[18] D. Kostic, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: High Bandwidth Data Dissemination Using an Overlay Mesh. In Proceedings of the 19th ACM Symposium on Operating SystemsPrinciples (SOSP 2003), October 2003.

[19] K. Sripanidkulchai, A. Ganjam, B. Maggs, H. Zhang. The Feasibility of Supporting Large-Scale Live Streaming Applications with Dynamic

Application End-Points. In Proceedings of SIGCOMM 2004, Portland, Oregon, Aug. 2004.

[20] W. T. Ooi. Dagster: Contributor-Aware End-Host Multicast for Media Streaming in Heterogeneous Environment. In Proceedings of MMCN 2005, San Jose, California, January 2005.

[21] S. Saroiu, P. Krishna Gummadi, and S. D. Gribble. A Measurement Study of Peer-to-Peer File Sharing Systems. In Proceedings of Multimedia Computing and Networking, 2002.

[22] B. Cohen. Incentives build robustness in BitTorrent. In Proceedings of the First Workshop on the Economics of Peer-to-Peer Systems, Berkeley, CA, June 2003.

[23] R. Sherwood, R. Braud, and B. Bhattacharjee. Slurpie: A Cooperative Bulk Data Transfer Protocol. In Proceedings of IEEE INFOCOM'04, Hong Kong, March 2004.

[24] V. Vishnumurthy, P. Francis. On Random Node Selection in P2P and Overlay Networks. Technical Report, Department of Computer Science, Cornell University, 2004.

[25] G. Pandurangan, P. Raghavan, and E. Upfal. Building low-diameter p2p networks. In STOC 2001, Crete, Greece, 2001.

[26] D. Stutzbach, D. Zappala, and R. Rejaie. The Scalability of Swarming Peer-to-Peer Content Delivery. In Proceedings of 2005 IFIP Networking Conference, Waterloo, Ontario, Canada, May, 2005.

[27] BitTorrent, Inc. The BitTorrent homepage. http://www.bittorrent.com/, 2002.

[28] A. J. Ganesh, A.-M. Kermarrec, L. Massoulie. Peer-to-Peer Membership Management for Gossip-Based Protocols. IEEE Transactions on Computers, Vol. 52, No. 2, February 2003.

[29] M. Feldman and C. Papadimitriou and J. Chuang, and I. Stoica. Free-Riding and Whitewashing in Peer-to-Peer Systems. In Proceedings of ACM SIGCOMM'04 Workshop on Practice and Theory of Incentives in Networked Systems (PINS), August 2004.

[30] M. Handley and S. Floyd and J. Pahdye and J. Widmer. TCP Friendly Rate Control (TFRC): Protocol Specification. RFC 3448, Proposed Standard, January 2003.

[31] Z. Cataltepe, P. Moghe. Characterizing Nature and Location of Congestion on the Public Internet. In Proceedings of ISCC'2003, Kemer, Antalya, Turkey, Jun. 2003.

[32] K. Gummadi, S. Saroiu, S. Gribble. King: Estimating Latency between Arbitrary Internet End Hosts. In Proceedings of SIGCOMM Internet Measurement Workshop (IMW 2002), Marseille, France, Nov. 2002.

[33] T. Hogan. The BNBT homepage. http://bnbt.depthstrike.com/, 2006.

[34] Microsoft Corporation. Windows Media Encoder. http://www.microsoft.com/windows/windowsmedia/forpros/encoder/default.mspx, 2006.

[35] Microsoft Corporation. Advanced Systems Format (ASF) Specification. http://go.microsoft.com/fwlink/?LinkId=31334, 2004.

[36] UPnP Forum. Internet Gateway Device (IGD) Standardized Device Control Protocol V 1.0. http://www.upnp.org/standardizeddcps/igd.asp, 2001.

[37] S. Guha. STUNT - Simple Traversal of UDP Through NATs and TCP too. http://nutss.gforge.cis.cornell.edu/pub/draft-guha-STUNT-00.txt, December 2004.

[38] SimulTV. http://www.cs.ualberta.ca/ tianhao/simultv/, 2006.

[39] T. Qiu and I. Nikolaidis. On the Performance and Policies of Mobile Peer-to-Peer Network Protocols. In Proceedings of the 2nd Annual Communication Networks and Services Research Conference, Fredericton, New Brunswick, Canada, May 19-21 2004, pp. 208-217.

[40] M. Grossglauser and D. N. C. Tse. Mobility Increases the Capacity of Ad Hoc Wireless Networks. IEEE/ACM Transactions on Networking, Vol. 10, No. 4, August 2002.

[41] W. H. Yuen and R. D. Yates and S.-C. Mau. Exploiting data diversity and multiuser diversity in noncooperative mobile infostation networks. In Proceedings of 22nd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'03), San Francisco, CA, Mar. 2003.

[42] K. Sripanidkulchai. The popularity of Gnutella queries and its implications on scalability. In Proceedings of O'Reilly Peer-to-Peer & Web Services Conference. http://www-2.cs.cmu.edu/ kunwadee/research/p2p/gnutella.html, Washington, D.C., Sept. 2001.