Stony Brook University

Incentive Mechanisms for Peer-to-Peer Streaming

A Dissertation Presented

by

Vinay Pai

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

May 2011

Stony Brook University

The Graduate School

Vinay Pai

We, the dissertation committee for the above candidate for the

Doctor of Philosophy degree, hereby recommend

acceptance of this dissertation.

Dr. Erez Zadok — Dissertation Advisor
Associate Professor, Computer Science Department

Dr. Michael A. Bender — Chairperson of Defense
Associate Professor, Computer Science Department

Dr. Rob Johnson
Assistant Professor, Computer Science Department

Dr. Martin Farach-Colton
Professor, Computer Science Department,
Rutgers, The State University of New Jersey

This dissertation is accepted by the Graduate School

Lawrence Martin
Dean of the Graduate School

Abstract of the Dissertation

Incentive Mechanisms for Peer-to-Peer Streaming
by

Vinay Pai

Doctor of Philosophy
in
Computer Science

Stony Brook University

2011

The increasing popularity of high-bandwidth Internet connections has enabled new applications like the online delivery of high-quality audio and video content. Conventional server-client approaches place the entire burden of delivery on the content provider's server, making these services expensive to provide. A *peer-to-peer* approach allows end users to reduce the burden on the service provider by contributing bandwidth by uploading data they have downloaded to other clients. However, the success of a peer-to-peer system hinges on resources contributed by participants. Unfortunately, studies have shown that end users are often reluctant to contribute resources to the system without a concrete incentive to do so. Our thesis is that a robust incentive mechanism is necessary to encourage nodes to contribute resources to the system, and a receiver-driven architecture with a pairwise incentive mechanism allows for great flexibility, simplicity, robustness, and performance.

The popular file sharing software BitTorrent is widely used, and includes an incentive mechanism that aims to tie the quality of service a node receives to the amount of resources it contributes. Their incentive mechanism is *pairwise*, in that nodes only rely on direct first-hand observations eliminating the need for complex distributed algorithms. However, studies have shown that flaws in BitTorrent's incentive mechanism make it vulnerable to gaming. We present SWIFT, our alternative incentive mechanism for BitTorrent-like file sharing applications, and experimentally show that it is more resistant to gaming, while retaining the benefits of a pairwise mechanism.

Having validated pairwise incentive mechanisms, we turn to our main goal of live streaming. Pairwise mechanisms rely on a bi-directional flow of data between nodes so that nodes may directly penalize neighbors that do not upload data to them. Therefore, traditional tree-based live streaming systems are not amenable to pairwise incentives. We address this with Chainsaw, our peer-to-peer live streaming system based on an unstructured mesh network. Through extensive experimental evaluation we demonstrate that Chainsaw is able to support high-bandwidth streams to a large number of simultaneous receivers with low packet-loss rates over a wide range of network sizes and other system parameters.

We then build on Chainsaw and present Token Stealing, our pairwise incentive mechanism for peer-to-peer streaming. Through detailed experimental evaluation, we show that our algorithm offers good service to all participants in the network when the system is resource-rich.

When the system is resource-constrained, however, nodes that contribute resources receive significantly better service than those that do not.

Thus, we show that our system is versatile and scalable, offering excellent performance across a wide range of system parameters and network conditions, with a robust incentive mechanism that promotes resource-rich conditions by encouraging nodes to contribute as much bandwidth to the system as they are able.

Contents

List of Figures

List of Tables

Acknowledgments

First and foremost, I thank Alexander Mohr, my advisor for the first five years of graduate school. Beginning with my very first day of grad school when I took his advanced networking course, he has been a great mentor and inspiration. He has always been amazingly accessible; there was never a time when I couldn't knock on his office door if I needed guidance or advice. Alex has always been a source of great ideas and pointed me towards new and interesting approaches, while giving me the freedom to pursue my ideas independently. His insightful questions often helped me clarify my own ideas and ultimately made me a much better researcher. I also thank Alex for making the time for regular meetings and offering his continued input even after he left Stony Brook University.

I thank Erez Zadok, who graciously took over as my advisor after Alex left the university. Erez has provided me with great guidance over the past three years, as well as earlier in my graduate career. His input has helped further enhance my research and writing skills, and helped me cover the last mile.

I also thank Michael Bender for innumerable useful discussions over the years, which often got my thinking out of a rut and helped me solve problems I had been stuck on. His insight and enthusiasm have been an inspiration and a guiding force, and helped shape my research career in many subtle ways.

I thank Michael Bender, Rob Johnson, and Martin Farch-Colton for serving on my defense committee, as well as R. Sekar and Tzi-Cker Chieuh who served on my RPE committee. Their input at various stages of the PhD program has been invaluable.

I thank the anonymous reviewers of all the papers I submitted over the years. Their comments have vastly improved the quality of my research and helped me identify flaws, discover interesting new directions, and express myself more clearly in my writing.

I thank Sean Callanan, Amit Bhargava, Karthik Shanmugasundaram, Karthik Tamilmani, Josef Sipek, and Justin Seyster who have been great friends through most of my time at Stony Brook. Having these people around during late night sessions with looming deadlines (and the occasional late night movie session) made all the difference. Numerous other lab-mates past and present also gave me valuable input and participated in useful discussions over the years. For this, I thank Vinay Sambhamurthy, Kapil Kumar, Ritesh Maheshwari, Shweta Jain, Nithin Raju, Sandra Tinta, Aarthi Andrade, Hari Krishnan, Aditya Kashyap, Puja Gupta, Akshat Aranya, Avishay Traeger, Rick Spillane, Adam Martin, D.J. Dean, Vasily Tarasov.

I thank Sam Yagan, Chris Coyne and Maxwell Krohn, the founders of OkCupid.com who have supported me financially for the past year through the Strategic Partnership of Industry and Research (SPIR) program, and have been incredibly understanding of my absence from work during busy times.

Finally, I thank my parents. They have always been supportive, encouraging, and above all loving. They have always held me to the highest standards without ever making me feel

pressured to go in directions I did not want to, and have undoubtedly played a huge role in shaping who I am today. Indeed, growing up in a household with two PhDs made going to graduate school almost seem like the obvious choice rather than the intimidating experience it might have been otherwise.

Chapter 1

Introduction

Today, a large majority of people in the United States have broadband Internet connections. According to research conducted by the Pew Research Center in 2009, 77% of families in the United States had broadband Internet connections at home [44].

Consumer-grade services such as DSL, Cable, and fiber-to-the-home are now capable of supporting bandwidth-intensive applications like high-quality audio and video, including DVD-quality (4Mbps) streams. In fact, many connections are even capable of supporting HDTV-quality (20Mbps) streams.

However, the bandwidth cost of supporting large numbers of clients at those data rates is prohibitive. For example, Google's popular video sharing website YouTube spent over $360 million on bandwidth in 2009 [60], which prevented that division of the company from turning a profit despite (1) substantial advertising revenue, (2) the great popularity of its service, and (3) the fact that most videos are only offered at a lower quality.

This problem is further exacerbated in the case of *live* streaming applications, because they must provision for the demands of large numbers of simultaneous viewers when a popular program is broadcasted. Popular events like the 2010 FIFA World Cup can attract large numbers of viewers and would place a significant burden on the content provider's network. This is in contrast to traditional over-the-air broadcasts where the cost to the content provider is relatively constant, but increased viewership can bring in increased advertising revenue.

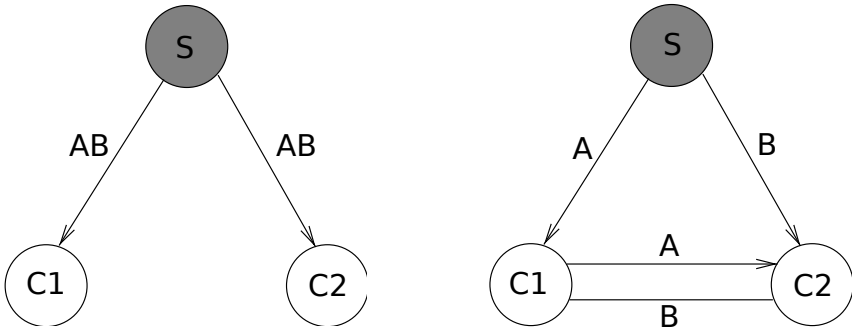
The peer-to-peer model allows viewers to share the burden by contributing bandwidth, thus making it more economical to support large numbers of viewers at high quality levels. This would open up new opportunities for both commercial content producers and individuals who would be able to focus their resources on producing quality content, and reach wide audiences.

1.1 Peer-to-Peer Model

The main reasons for the high bandwidth cost of the traditional client-server model is the fact that every client is directly supported by the provider's servers. This places the entire bandwidth burden on the content provider's network. While this burden might be distributed via commercial content distribution networks like Akamai [91], the cost is still borne by the content provider.

The cost to the provider rises linearly with number of viewers, since it must send a complete

copy of the data stream to every viewer. The cost of the bandwidth and hardware required to support those viewers can quickly become prohibitive.

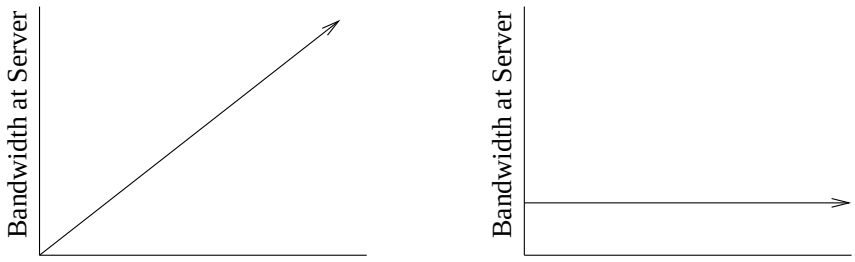


(a) Client-Server Model (b) Peer-to-Peer Model (Ideal)

Figure 1.1: In the client-server model, the server must send all data to every node. In the peer-to-peer model, peers exchange data among each other, keeping the load on the server low. In the ideal case the server only needs to transmit a single copy of each packet, regardless of the number of clients.

An alternative to the client-server model is the peer-to-peer model. In this model, clients connect not only to the provider's servers, but also to some or all of the other clients who are viewing the same content. This allows clients to contribute bandwidth to the system by uploading data to other clients who wish to receive the same content.

In the ideal case, peers, on average, contribute as much bandwidth to the system as they consume, and thus pose no net drain on the system's resources. In this ideal case, the bandwidth demand on the content provider remains constant regardless of the number of clients viewing the content. This would allow the cost to the provider to remain independent of the number of viewers, much as it does for traditional broadcast technologies



(a) Client-Server Model (b) Peer-to-Peer Model (Ideal)

Figure 1.2: In the client-server model, the load on the server increases linearly with the number of clients. In an ideal peer-to-peer system, however, the load on the server remains constant.

Our goal is to build a robust, scalable system for live media streaming. The live streaming model will support applications like high-quality online radio and TV stations.

1.1.1 Need for an Incentive Mechanism

Unfortunately, studies have shown that participants in peer-to-peer networks often try to gain benefit from the system without contributing resources in return. The analogous problem in society has been well studied by economists [7, 49, 61, 74], and is known as *free-riding*. One of the key reasons for free-riding is a lack of a concrete incentive to contribute. For example, Sariou et al. [84] have estimated that 20–40% of Napster users and up to 70% of Gnutella users shared little or no content. Huberman and Adar [2] found that nearly 50% of responses are returned by only 1% of hosts and that nearly 98% of the responses were returned by 25% of the sharing hosts.

An incentive mechanism provides nodes with a concrete incentive to contribute resources to the system by offering better performance to nodes that contribute bandwidth than those that do not. Therefore, we argue that in order to build a robust peer-to-peer streaming system, we must also develop a robust incentive mechanism.

1.2 Our Approach

A major inspiration for our work is BitTorrent [19], a popular system that solves the analogous problem of peer-to-peer *file sharing*. BitTorrent is commonly used to distribute large files like CD images. In addition to a file transfer mechanism, BitTorrent provides an incentive mechanism.

BitTorrent nodes simply measure the rate at which each of the peers it is connected to uploads data to it. The node then *chokes* (i.e., stops sending data to) all but the top few of its peers. This approach has the key benefit of simplicity, since nodes only rely on first-hand observations to implement this algorithm.

However, this mechanism has been shown to be relatively easy to cheat [46, 75]. In Chapter 3, we present SWIFT, an alternative incentive mechanism for BitTorrent-like networks that addresses many of the problems with BitTorrent's incentive mechanism. This shows that while BitTorrent's particular mechanism may be flawed, it is possible for a simple algorithm that depends only on first-hand observations to be effective.

At first glance, file sharing and live streaming might seem to be nearly identical problems. However, there are key differences that prevent direct application of file sharing incentive mechanisms to live streaming systems. The key differences between file sharing and live streaming applications are summarized in Table 1.1.

- With file sharing systems, the entire file is available ahead of time; in live streaming, it may be generated on-the-fly (as with a live sporting event, for example).
- With file sharing, different clients can download entirely different parts of the file while live streaming clients must all download the same data with a small buffer on the order of seconds to allow for reordering. This synchronization requirement adds additional challenges.
- File sharing applications have no target speed, since the file can always be downloaded faster. Therefore, there is no concept of spare capacity. In contrast, streaming applications are limited by the rate of the source stream. If peers are willing to contribute more

	File Sharing	**Live Streaming**
Source Data	Source node has the entire file ahead of time	Data may be generated on-the-fly
Client Synchronization	Not needed	Yes, within a relatively small window of tolerance
Target Download Speed	Faster the better	Equal to the stream rate
Data Useful to Peers	Forever	Only for a short time
Result of slow speed	File downloads slowly, but remains useful	Disruptions in viewing, or degraded quality

Table 1.1: Key differences between file sharing and live streaming systems

bandwidth than the stream rate, the system may have spare capacity. This spare capacity may be exploited to accommodate nodes that are unable to upload data quickly, so long as other altruistic nodes upload enough data to make up the deficit.

- Partial data downloaded in a file sharing application can always be uploaded to other clients since they must assemble the complete file. However, clients in live streaming applications typically have no use for data transmitted in the distant past.
- Limited bandwidth in a file sharing system will still allow the client to download the complete file, even if it takes longer. In a streaming system, however, it will result in either degraded quality or even a complete disruption of service, depending on how the source material is encoded.

As a result of these differences, we cannot directly apply either BitTorrent's incentive mechanism or our improved SWIFT mechanism to a live streaming application. Therefore, we must design a new mechanism with live streaming in mind. However, we seek to retain the simplicity and elegance of those file sharing systems, and rely solely on local decisions based on first-hand observations.

Due to the need for synchronization, traditional peer-to-peer streaming systems are based on *multicast trees*, with the source node at the root. Every participant propagates data it receives from its parent to its descendants. This creates parent-child relationships between nodes, making it difficult to apply pairwise incentive mechanisms because data only flows in one direction between any given pair of interacting nodes.

Therefore, we designed Chainsaw, a peer-to-peer based on an unstructured mesh network. As with BitTorrent and SWIFT, there is no hierarchy between nodes, as a node may download certain packets from a neighbor while sending others to the same neighbor. This makes the network amenable to pairwise incentives. Moreover, Chainsaw achieves this goal while providing excellent performance. Through simulation, we demonstrate that Chainsaw is able to support a large number of nodes with low packet loss, low delay, low overhead, and an excellent resilience to churn (i.e., continuous arrival and departure of nodes). We discuss the design of Chainsaw in Chapter 4.

In Chapter 5, we present Token Stealing, an incentive mechanism built on top of Chainsaw. Token Stealing is a straightforward extension of the standard token bucket algorithm, and fills our desired goal of simplicity.

We show that our algorithm preferentially directs bandwidth to nodes that contribute upload bandwidth to the system, offering them good performance. Moreover, our algorithm also takes advantage of any surplus capacity that may exist to support nodes that are unable to upload data at the stream rate. However, when the system is resource-constrained, nodes that contribute resources to the system see significantly superior performance. This gives nodes an incentive to contribute as much bandwidth as they are capable of, and discourages them from artificially limiting their upload bandwidth.

1.3 Evaluation Methodology

We built a high-performance discrete-event simulator in C++ to implement our system. The simulator implemented all aspects of our protocol and allowed us to investigate various aspects of network performance in a controlled environment.

Beginning with a conservative base setup, we systematically evaluated different aspects and parameters by varying one network or system parameter at a time. We demonstrate that the Chainsaw streaming protocol supports high bandwidth streaming with low packet loss rates, low delay, and quick startup times. We show that the system scales well with size and stream rate, and is robust to churn (i.e., nodes leaving and joining the system). We show that the system is stable and efficient across a wide range of network and system parameters and therefore does not require careful tuning to work.

We also demonstrate that the Token Stealing algorithm achieves our goal of giving lower packet loss rates to nodes that contribute bandwidth in resource-rich systems, while taking advantage of altruistic nodes to give low packet loss rates to all nodes whenever possible. We show that the system reacts quickly to changes in node behavior as well as system-wide resource availability.

Finally, we built a native prototype implementation of our system, and used it to perform experiments on the PlanetLab [18] testbed. Although the limited resources and control offered by PlanetLab nodes did not allow us to replicate all simulator experiments, we were able to demonstrate good performance for most nodes in the system in the resource-rich case, and improved performance for nodes that upload more data in the resource-constrained case. The performance characteristics are similar to those obtained in our simulations, thus helping validate our simulation results.

1.4 Contributions

The key contributions of this dissertation are as follows:

- We designed SWIFT [89], an improved incentive mechanism for BitTorrent-like file sharing systems. We conducted experimental evaluation of this system to demonstrate that pairwise incentives, while simple, can be effective.

- We designed Chainsaw [72], a mesh-based live streaming application which offers high performance and scalability, and has bi-directional relationships between peers, making it amenable to pairwise incentive mechanisms.

- We conducted an extensive experimental evaluation of our streaming protocol to demonstrate performance and scalability, as well as the effects of the various network parameters.
- Our initial paper on Chainsaw [72] has been well received by the community, with over 250 citations. Duijkers et al. [26] have used Chainsaw as a basis for an experimental video streaming application and conducted further evaluation in addition to our own. Biskupsi et al. [8] have proposed an extension of our protocol to further improve performance through changes to the network topology.
- We designed the Token Stealing algorithm [73], a simple but effective pairwise incentive mechanism for our mesh-based live streaming system.
- We conducted an extensive experimental evaluation of the incentive mechanism to demonstrate its effectiveness in various situations as well as changing network conditions and node behaviors.

1.5 Outline

The outline of this dissertation is as follows. In Chapter 2 we discuss background information. In Chapter 3 we present the SWIFT incentive mechanism for BitTorrent-like file sharing applications. In Chapter 4, we describe the Chainsaw incentive-compatible mesh-based streaming protocol. In Chapter 5 we present the Token Stealing incentive mechanism for live streaming applications. In Chapter 6 we present experimental results. In Chapter 7 we present related work. In Chapter 9 we present future directions of research. Finally, in Chapter 8, we conclude.

Chapter 2

Background

In this chapter, we discuss background material in other multicast and peer-to-peer approaches. Furthermore, we also discuss multimedia (i.e., video and audio) codecs. Although our streaming protocol is application-agnostic, video streaming is one of the most popular live streaming applications, and an application for our system.

In Section 2.1, we discuss IP multicast. In Section 2.2 we discuss high level concepts related to peer-to-peer file sharing and streaming. In Section 2.3 we discuss various network topologies commonly used in peer-to-peer networks. In Section 2.4 we discuss the need for incentive mechanisms and common approaches. Finally, in Section 2.5 we discuss video encoding technologies and how they may be applied to peer-to-peer streaming.

2.1 Multicast

Traditional network system design is based on the client-server model where resources of interest are placed at a centralized server, and are transmitted to clients on request. Widely used protocols such as HTTP [32] and FTP [77] use this approach, and it forms the core of the Web today.

However, popular content might be requested thousands or even millions of times by clients, placing a heavy burden on the server. Whereas the technical problem can be addressed by distributing the load using load-balancers, mirrors, and content distribution networks (CDNs), the cost is ultimately borne by the content provider, and can be prohibitive.

In contrast, traditional radio and television broadcasts bear a fixed cost to the provider to provide service to a given area regardless of the number of receivers tuning in. Such broadcast mechanisms rely on the presence of a shared medium (the electromagnetic spectrum, in the case of over-the-air broadcasts), which is not present in wide-area networks such as the Internet. Note that while wireless networking technologies such as 802.11 (i.e., WiFi), and cellular data services do use electromagnetic broadcasts, they are packet networks intended to provide point-to-point links between a router and the end device, and are not comparable to radio and television broadcasts.

IP Multicast [22] is an extension to the Internet Protocol which allows the creation of *multicast groups* that individual hosts could subscribe to. Packets addressed to a multicast group would be transmitted by underlying routers to all hosts subscribed to that group without requir-

ing the content provider to transmit multiple copies of their content. However, IP Multicast relies on wide scale implementations by the underlying network providers, who have been reluctant to enable IP multicast due to concerns about scalability and potential for abuse.

Peer-to-peer technology is an analogous network built at the application level by participants in the network as an *overlay*, allowing application-level support for similar functionality without relying on widespread adoption by network providers. Peer-to-peer systems are also often referred to as *overlay networks* or *application-level multicast*.

2.2 Peer-to-Peer Systems

In this section we provide a broad overview of peer-to-peer systems in general, in order to place our work in context.

2.2.1 Peer-to-Peer File Sharing

Some of the earliest peer-to-peer networks to gain wide acceptance were built around file sharing, to allow users to easily locate and share images, music, and other content. Napster [31], one of the first, relied on a central server to index all the content and help participants locate the files they wanted. Actual files remained on the end users' computers. Other networks like Gnutella *decentralized* the system further by eliminating the indexing server altogether and instead routing searches through other peers.

In both Napster and Gnutella, files located were downloaded directly from the peer hosting it, which could place a significant burden on peers that hosted popular files. Other networks like e-Donkey as well as later versions of Napster and Gnutella allowed clients to download portions of the file from different peers if several clients had the same file.

BitTorrent [19] further improved the situation by allowing peers with *parts* of a large file to exchange pieces of that file with each other in order to assemble a complete file, thus greatly improving the speed of downloads in cases where only a few peers had complete copies but several were actively downloading it.

2.2.2 Peer-to-Peer Streaming

File sharing networks operate on the assumption that the file being shared is available in its entirety, and participants in the network seek to obtain it. Furthermore, the file may not be downloaded in order, and is thus often unusable until the entire download is complete.

These assumptions do not hold for a wide range of streaming applications such as video broadcasts. In live streaming applications, viewers expect to view the content soon after they join the network, rather than waiting for a lengthy download to complete. Moreover, playback is expected to continue smoothly without interruption.

In order to allow the viewer to consume content as it is downloaded, the data must be received in order. The use of buffering allows a small amount of re-ordering to be tolerated, on the order of a few seconds. Moreover, content must be delivered at a relatively steady rate within tolerances offered by the buffer, or else the playback would have to be paused to allows the network to catch up.

We believe that one of the key factors driving BitTorrent's popularity is the great simplicity of the protocol which has lead to a wide range of implementations across numerous platforms. Therefore, we consider simplicity to be a major design goal, in addition to addressing all the challenges posed by high-performance peer-to-peer streaming.

We describe a number of other related peer-to-peer streaming systems and contrast them to our own in Chapter 7.

2.3 Network Topology

A key consideration in a peer-to-peer network is the topology of the network. Peer-to-peer streaming networks can be broadly classified into tree-based and mesh-based approaches.

2.3.1 Tree Networks

The goal of a streaming protocol is to deliver data in order from the source to all the participants in the network. These requirements naturally lend themselves to a tree-based approach.

A tree-based system arranges the participants in a network in a tree rooted at the source node. The source node transmits data to the nodes it is connected to (i.e., its children), who in turn transmit it to their children, and so on. This approach makes for trivial routing of data, as well as easy in-order delivery.

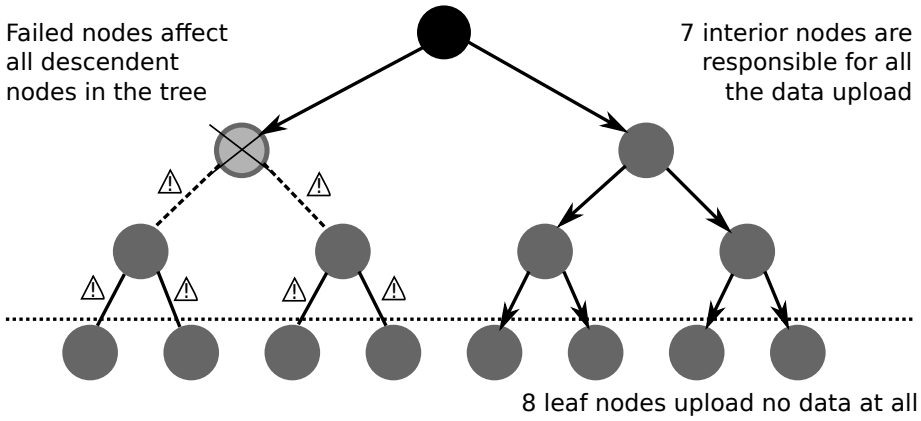


Figure 2.1: An example of a simple tree-based network, and some of its drawbacks. A minority of nodes are responsible for all the data upload. This problem would be further exacerbated in higher degree trees. Failed nodes high up in the tree affect all their descendants.

However, a simple tree-based network has a number of disadvantages, as illustrated in Figure 2.1. A majority of the nodes in a balanced tree are leaf-nodes, i.e., nodes with no children, so the burden is only shared by a fraction of nodes (the internal nodes). Moreover, when an interior node leaves the network or becomes unresponsive, all its descendants are affected until the tree is repaired, which might lead to wide-scale disruptions (albeit temporary).

A number of innovative solutions have been proposed to these problems by constricting multiple trees [11], or periodically reconstructing trees [69]. We discuss these works in more detail and contrast them to our own work in Chapter 7.

2.3.2 Mesh Networks

An alternative approach is the mesh topology based on more general network graphs than trees. As shown in Figure 2.2, a mesh-based network addresses many of the problems of a tree-based network. As there are several paths between any given nodes, a failed node has a limited impact on other nodes. Moreover, since different packets can take different routes, the upload burden can be distributed more fairly among nodes.

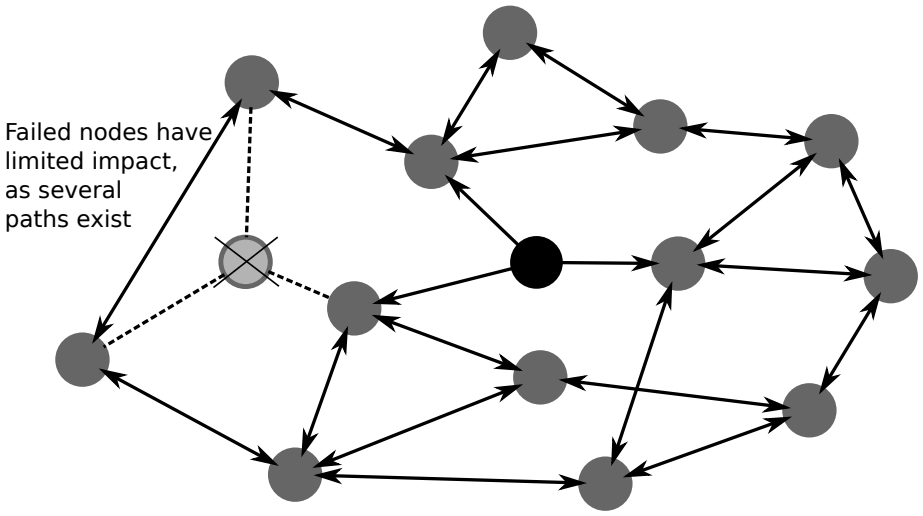


Figure 2.2: An example of a network based on a random graph. The load is distributed more fairly, and single node failures do not partition the network as many alternate paths exist between any two nodes.

Bullet, an early mesh-based system continued to rely on a tree as the primary data dissemination pathway, but allowed for alternative paths to fetch data not delivered by the main tree [51]. In 2005, we proposed Chainsaw [72], one of the first streaming systems suitable for high-performance applications to eliminate the tree entirely in favor of a random graph. While gossip-based protocol that use an unstructured mesh existed well before either Bullet or Chainsaw were proposed, those systems lead to significant duplicate transmissions, and are not suitable for high-bandwidth streaming application. We describe our protocol in detail in Chapter 4. We discuss Bullet and gossip protocols, as well as a number of alternative systems proposed by a number of other authors, and contrast them to our on work in Chapter 7.

2.4 Incentive Mechanisms

Peer-to-peer networks rely on resources contributed by participants. However, participants are often selfish and avoid contributing resources if they can avoid it, leading to poor performance.

An incentive mechanism is a mechanism built into the network that penalizes participants that do not contribute enough resources to the network, and rewards those that do contribute. The penalty may be reduced performance, or complete exclusion from the network.

Incentive mechanisms can broadly be classified into reputation-based systems, and pairwise tit-for-tat systems.

2.4.1 Reputation-Based Systems

Reputation bases systems are incentive mechanisms where a node's contribution over time is monitored and that information is shared with other participants in the network to enable them to reward or penalize a given participant, as appropriate. Reputation-based systems have the advantage of taking a node's long-term behavior into account, as well as the possibility of a system-wide view [48, 69].

However, reputation systems often rely on a centralized server, or on distributed algorithms to propagate information about node behaviors throughout the system. In order to be effective, two key challenges must be addressed. Firstly, a participant must not be able to falsify its reputation either by assuming another participant's identity, or by easily creating a new identity to avoid being penalized for past bad behavior. Secondly, a distributed reputation systems must be resilient to collusion and false reporting since nodes rely on information relayed to them through third parties.

2.4.2 Tit-for-Tat Pairwise Systems

An alternative model is the pairwise, or tit-for-tat approach where participants rely on first-hand observations. A participant that receives good service (e.g., fast downloads) from a peer can reward that node by offering it good service in return. A pairwise system is a often far simpler to design and analyze than a reputation-based system due to a reliance on first-hand observations [19, 53].

However, a tit-for-tat approach is only possible in systems where there is a *mutual* exchange of services. This the case in file sharing systems (like BitTorrent), but not in tree-based streaming approaches where nodes have a parent-child relationship and data only flows down the tree.

Our streaming protocol, Chainsaw, is mesh-based, with a mutual flow of data between nodes Therefore, it is amenable to a pairwise reputation system. In Chapter 5 we present Token Stealing, a simple yet effective pairwise incentive mechanism.

2.5 Multimedia Coding

The Chainsaw streaming protocol is application-agnostic, and merely provides a high performance data-dissemination protocol. However, it is useful to consider the system with a target application in mind. Our protocol naturally lends itself to the popular *video streaming* application, so we briefly discuss video encoding technology, and how video transmission may be affected by network characteristics.

2.5.1 Video Compression

Digital video consists of a series of images that are displayed in succession, typically at rates between 24 and 60 frames per second. Raw video would be extremely large in storage or bandwidth requirement, so video is typically stored and transmitted in a compressed format. Video compression algorithms are known as *codecs*, short for coder/decoder. MPEG [38] and H.264 [98] are examples of video codecs in popular use today.

Video codecs are typically *lossy*, i.e., they sacrifice some fine detail in order to reduce the data needed to encode a given frame. In addition, video codecs typically do not encode frames independently, but take advantage of the fact that large portions of the scene typically do not change from frame to frame. This allows for further reductions in data size.

Many video codecs allow a given source video to be encoded at a number of different quality levels by adjusting the amount of detail that is discarded during compression. Moreover, the size of the stream can be adjusted by scaling the video down to reduce the image size. This allows multiple versions of a given source video to be created that are suitable for different network bandwidths.

However, due to the nature of the compressed data, small corruption or gaps in data caused by packet loss in the network layer can lead to a severe degradation in the quality of the decoded video. Furthermore, the interdependency between frames means that a momentary disruption may cause visual distortions that last several seconds. Therefore, packet loss rates of even a few percent may be intolerable when dealing with video streams.

2.5.2 Erasure Coding

Erasure coding is a type of *forward error correcting code* that is commonly used with a number of types of network transmissions including video transmission. Erasure coding allows the receiver of a transmission to recover from one or more missing packets in the stream. These packets may be missing either because they were never delivered by the network, or they were delivered in a corrupted state, and discarded. The technique is calls forward error correcting because the sender includes redundant information in the original transmission, and does not rely on retransmission requests from the receiver. This is advantageous in situations like broadcasts where a back channel to request retransmissions are either unavailable or impractical.

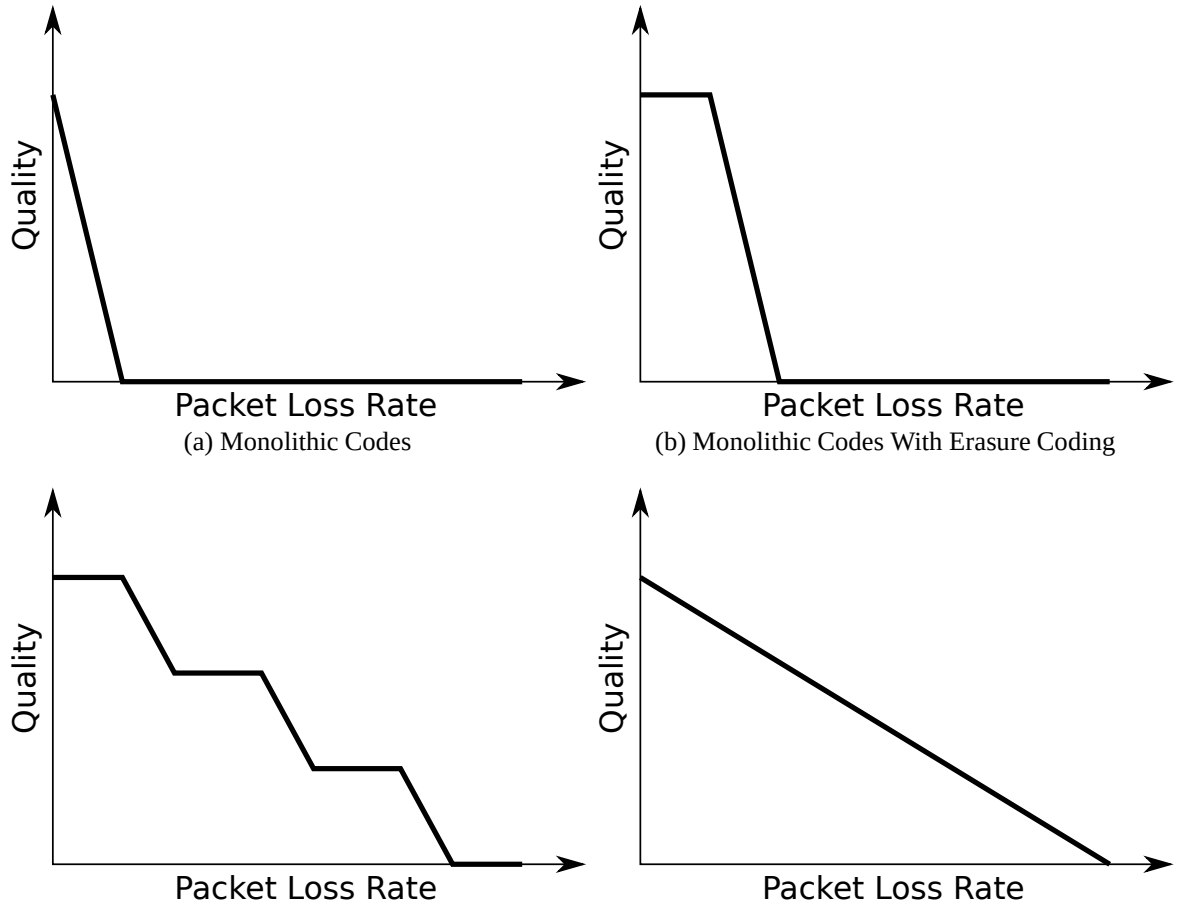
With erasure coding, the stream is divided into packets. Every group of m packets is then encoded into a set of n packets where $n > m$. The mathematical relation between the encoded packets allows the original packets to be recovered if *any* set of m of the n packets are available. Thus, erasure coding allows the original stream to be recovered intact so long as at no more than $n - m$ packets are lost or corrupted in transmission, allowing for more robustness.

Reed-Solomon Codes [97] are a commonly used method of erasure coding. An alternate algorithm known as Tornado Codes [9] offers far higher performance than Reed-Solomon codes, but requires slightly more than m packets on average to decode each group.

The combination of video encoding algorithm and erasure coding determines how much packet loss rate can be tolerated, and the resulting loss rate vs. perceived quality curve. Figure 2.3 shows a qualitative comparison of various techniques.

2.5.3 Layered Codecs

Layered codecs are an alternative to creating entirely separate streams for different bandwidth levels. In a layered codec, the video is encoded as a *base layer* and one or more *enhancement layers*. Receiving the base layer allows the recipient to decode a relatively coarse version of the video. The video may be progressively refined by including successive enhancement layers during the decoding process.



(a) Monolithic Codes

(b) Monolithic Codes With Erasure Coding

(c) Layered Codes With Unequal Loss Protection

(d) Fine Granularity Scalability

Figure 2.3: Quality of the decoded video as a function of packet loss rates for various video encoding methods.

Layered codecs allow for added flexibility. Participants that are unable to receive the highest quality video due to bandwidth limitations may still be able to receive a lower quality one by discarding some or all of the enhancement layers. Moreover, layered codecs can be combined with erasure coding using a technique called *unequal loss protection* [65, 101] to ensure that later enhancement layers are more likely to be lost than the base layer or earlier enhancement layers for any given level of packet loss, thus maximizing the expected quality of the received video. Codecs without layering are known as *monolithic* codes.

2.5.4 Fine Granularity Scalability

Fine granularity scalability [54] is a further improvement over layered codecs. There is a limit to the number of enhancement layers a video may be divided into, and an enhancement layer is generally only useful when received in its entirety along with the base layer and all the enhancement layers below it. Therefore, with layered codecs, the received quality is a step function of the number of packets received.

Fine Granularity Scalability (FGS) allows for a smoother size/quality curve by allowing a

continuous enhancement layer where incremental additions remain useful and lead to improved quality.

To summarize, videos with monolithic encodings suffer rapid degradation with increasing packet loss and are quickly rendered unusable. With erasure coding, a certain amount of packet loss can be tolerated, but the same rapid degradation results once the packet loss rate exceeds the amount of redundancy added by the erasure coding. Layered codecs with simple erasure coding would suffer the same fate as earlier layers are just as likely to get corrupted, but with unequal loss protection, however, base layers and lower enhancement layers are more protected, leading to a series of steps. Finally, fine granularity scalability allows for a smooth quality vs. data size trade-off.

Our protocol is application-agnostic and the issue of content encoding is orthogonal to our work. Therefore, we do not implement the techniques described in this section. Instead, we consider the packet loss rate to be a key figure of merit, and assume that a practical application would use techniques like unequal loss protection, or fine granularity scalability in encoding the video, and that a lower packet loss rate leads to higher perceived quality for the user.

Chapter 3

SWIFT: Economic Incentives for File Sharing

File sharing is the one of the most dominant peer-to-peer applications today. Whereas the first peer-to-peer systems like Napster and Kazaa focused on the problem of searching for content, BitTorrent has emerged as the dominant file *distribution* system. This system is designed for the large-scale dissemination of big files, typically hundreds of megabytes to a few gigabytes in size. Files are broken up into chunks called *packets* or *pieces*, and nodes assemble the target file by acquiring and exchanging various pieces from different participants in the system. This technique is called *swarming*.

Although our ultimate goal is to build a robust live streaming application, we begin by studying the file sharing model in order to establish swarming as a viable technique for large-scale data dissemination, and pairwise incentives as a robust mechanism to incentivise participants to contribute resources to the system.

In this chapter, we first describe the file trading model. We then parametrize node behavior, and describe various strategies participants may adopt. We then propose a desired default behavior for nodes that promotes system stability, and experimentally show that rational self-interested nodes have little incentive to deviate from the specified behavior.

3.1 Introduction

3.1.1 The File Trading Model

Although early file sharing networks such as Napster and Gnutella only allowed peers to download entire files from a single peer only, more recent networks like BitTorrent allow for finer granularity. Files are broken down into packets (or pieces) and a peer can download different packets from different neighbors. Such a mechanism is useful, because a peer can exploit the resources of multiple neighbors simultaneously, and thus obtain the file quicker.

Certain kinds of content can easily attract large numbers of downloaders in a short period of time. A peer-to-peer network can significantly speed up distribution. For example, it was observed in the BitTorrent [20] network that for the first three days after the release of the RedHat 9.0 ISO, there were always more than 2,500 peers simultaneously downloading that

1.6GB file, with a peak of 4,400 peers [37].

A similar example occurs when software vendors release a large patch or update. For example, when Microsoft releases a new security patch of service update, millions of computers running the Windows operating system will all be interested in obtaining that update as quickly as possible. As evidenced in 2004 by the spread of the Witty worm [85] less than 24 hours after a patch was produced for the vulnerability it exploited, these machines should obtain patches as quickly as possible. In this case, peer-to-peer networks may be able to provide the update more quickly than a farm of dedicated servers, because of the large numbers of recipients involved. A robust incentive mechanism gives participants a reason to contribute upload bandwidth in order to speed up their own downloads and reduce their window of vulnerability.

In our system, we assume that a file is broken into packets of equal size and that the authenticity of each packet can be verified by a scheme such as a cryptographic hash or Merkle tree [64]. Our model is pull-based in that peers advertise the packets that they have; other peers then request specific packets from them. We further assume that the file sharing network has some mechanism in place for peers to discover fellow peers and join the system.

In SWIFT, we call peers who exchange packets as *traders*. A trader's main objective is to obtain a complete copy of the file as quickly as possible. Rather than negotiate a packet-for-packet trade as in a barter system, we assume each trader maintains a credit (a pairwise currency) for every peer to which it is connected. When the host receives and verifies a packet from a peer, the host increases the credit rating of that peer in proportion to the size of the received packet. Similarly, when a host fulfills a remote peer's request, the host decreases the credit rating of the peer by the size of that packet. A remote peer's request is satisfied only if it has accumulated credit greater than or equal to the requested packet's size.

In our current implementation, the pairwise currency is only used to reconcile current trading imbalances, not for tracking long-term node behavior. The accounting could be extended across multiple sessions to trade different files, but this is not critical to the working of our system.

In the next section, we introduce three different trading strategies and discuss which to choose.

3.1.2 Trading Strategies

We parametrize the behavior of peers based on how they extend credit to their neighbors. For every byte a peer receives, it extends the sender α bytes of credit in return. We call α the *repayment ratio*. In addition, it expends a fraction β of its total upload capacity U_{max} on *largesse* by uniformly distributing free credit to all N of its neighbors. Finally, a peer also extends every neighbor γ bytes of one-time credit the first time they interact.

The maximum number of bytes $u_{AB}(t)$ that peer A is willing upload to its neighbor B at time t, having received $d_{AB}(t)$ from B, is given by the equation:

$$u_{AB}(t) = \alpha d_{AB}(t) + \frac{\beta U_{max}}{N_A} t + \gamma. \tag{3.1}$$

Note that time t here is meant to represent wall-clock time and not a tit-for-tat mechanism in which time is divided into rounds.

Free-riders, who do not upload, have repayment ratio α, largesse rate β, and one-time free credit γ of zero. Distributors, who have no interest in downloading, have $\beta = 1$ and spend all

Peer Behavior	α	β	γ
Free Rider	0	0	0
Paranoid Trader	1	0	0
One-time Risk-taking Trader	1	0	1
Periodic Risk-taking Trader	1	$0 < \beta \leq 1$	1
Distributor	N/A	1	1

Table 3.1: A summary of the values of the parameters for some common peer behaviors.

their upload bandwidth on distributing packets to their neighbors. Traders who are mainly motivated by their desire to download a file as quickly as possible lie between these two extremes. Based on their choice of parameters, we classify them as paranoid traders, one-time risk-taking traders, or periodic risk-taking traders. The various strategies and the parameters they use are summarized in Table 3.1.

Paranoid traders

Paranoid traders are reciprocative players that wait until they receive a valid packet from a peer before offering to send an equal amount back. They have repayment ratio $\alpha = 1$ and never give out free credit ($\beta = 0$, $\gamma = 0$). This conservative strategy ensures that they will never upload more to a peer than they receive from it and thus will never be taken advantage of.

One-time risk-takers

Another strategy is for a peer to extend one packet of free credit to a peer the first time it is encountered to encourage them to trade. However, there is a chance that the peer will never receive a packet in return, so we call these traders one-time risk-takers. They set α and γ to 1, and β to 0.

Periodic risk-takers

Finally, some traders may be willing to give out free packets periodically. These traders dedicate a fraction $\beta > 0$ of their upload bandwidth giving out free packets to their neighbors. We call this type of free credit *largesse*.

The choice of a strategy

Table 3.1 summarizes the values of α, β and γ for the different types of peers that we have described. It is clear that if the system consists solely of paranoid traders, everyone will wait for their neighbors to make the first move and the system will be deadlocked.

At first glance, it would appear that one-time risk taking is sufficient to break the deadlock by giving peers a basis to start trading. However, we show through simulation in Section 3.3 that one-time risk-taking does not completely eliminate the deadlock. A peer that is not connected to a distributor will receive free packets from its neighbors when it first joins. However, it is

possible that it will acquire packets that none of its other neighbors are interested in, and will then be unable to trade and make further progress.

The one-shot free credit is also insufficient because of the peer identification problem. Allowing a peer to choose its identity will make the system susceptible to a Sybil attack [25]. One way to alleviate this problem is to use the IP address of a peer as its identifier. However, when there are many peers behind a Network Address Translator (NAT), all of them use the same IP address, so only one would receive the one-time free credit and bootstrap into the file sharing network, leaving the others to starve. On the other hand, a periodic risk-taker could distribute the IP's share of the largesse equally to each instance behind a NAT so that all of them are able to join the system.

Finally, if transport across the network is unreliable or subject to corruption, perfect accounting is not guaranteed. For instance, a peer may upload a packet and bill its neighbor for it, but the packet fails the cryptographic checksum and the peer receives no credit for it. The peer may then be stranded with no packets to trade and no credit with any of its peers, resulting in starvation.

Adopting the periodic risk-taking strategy increases the possibility of wasting upload bandwidth on free riders. We show in the next two sections, via mathematical analysis and simulation, that the advantages of this variant of Tit-for-tat [5] outweigh this potential drawback while maintaining robustness against a wide range of competing strategies.

3.2 Analysis

Let us now consider a homogeneous file trading system of N peers with upload and download capacities of $U_{max} = D_{max}$. Given a default strategy of periodic risk-taking, we analyze how that strategy interacts with others.

3.2.1 Bounds on Incentives to Defect

The bounds on the incentive for peers who wish to maximize their download rates to defect from the periodic risk-taking strategy can be made arbitrarily small. Consider the case of a mixed network of rational peers and periodic risk-takers. Let σ be the fraction of periodic risk-takers in the system, each of which contributes a fraction β of their upload bandwidth as largesse. If the share ratio of a peer is defined as the ratio of bytes uploaded to bytes downloaded, then the share ratio of periodic risk-takers $r_{periodic}$ is given by

$$r_{periodic} = \frac{1}{(1-\beta) + \beta\sigma}$$

and the share ratio of rational traders $r_{rational}$ by

$$r_{rational} = 1 - \beta\sigma$$

When $\beta = 0.1$, then in the extreme case of one rational peer among many periodic risk-takers, the greedy trader's share ratio is approximately 1.1. Similarly, in the other extreme of one periodic risk-taker among legions of rational peers, the risk-taker's share ratio is approximately 0.9. Clearly, these bounds can be made arbitrarily close to one by decreasing β.

3.2.2 Paranoid Traders vs. Periodic Risk-Takers

To show that a weak Nash equilibrium [41] can exist between paranoid traders and periodic risk-takers, we assume for simplicity that there are N peers in the system and that they each have the same upload and download capacities C. We also assume that each peer uses fair queuing among its neighbors to share its upload bandwidth.

We observe that paranoid traders will only trade with periodic risk-takers, as two paranoid traders will never risk a packet on each other. Thus, a paranoid trader will trade with σN periodic risk-takers, while a periodic risk-taker will trade with $N - 1$ peers.

If we assume the largesse rate β is sufficiently small, then each connection's capacity will be limited by the fair rate of $\frac{C}{N-1}$ that periodic risk-takers assign to each connection. Periodic risk-takers then achieve upload and download rates of $(N-1)(\frac{C}{N-1}) = C$, whereas paranoid traders achieve rates of $(\sigma N)(\frac{C}{N-1}) = \sigma C$ (as $N \to \infty$). Taking β into account and assuming the worst-case scenario in which none of the largesse is repaid, the download rate of periodic risk-takers falls to $(1-\beta)C$. Paranoid traders will download more quickly than periodic risk-takers when $\sigma > 1 - \beta$ and download less quickly when $\sigma < 1 - \beta$, so the system attains a weak Nash equilibrium point with respect to download speeds when $\sigma = 1 - \beta$. For small β, the equilibrium point is a network consisting almost entirely of periodic risk-takers.

3.2.3 Incentives to Prevent Free-Riding

Consider now a system consisting of a fraction σ of periodic risk-takers and a fraction $1 - \sigma$ of free-riders. Each free-rider will be able to download at a rate of $\frac{\beta C}{N-1}$ from each of the σN risk-takers, which results in a total download rate for the free-rider of $\beta\sigma C$ (as $N \to \infty$). Although free-riders can achieve share ratios of zero, they will download at a rate much lower than the risk-takers. For example, if $\sigma = 0.5$ and $\beta = 0.1$, they will download at a rate only 5% that of the risk-takers. Furthermore, as the number of free-riders increase, the incentive to become a risk-taker increases.

3.3 Experimental Results

We built a discrete-time simulator for our system. The simulator was implemented in nearly 2,000 lines of C codes, and allowed us to simulate networks with a desired set of parameters for each node. It produced a detailed trace of the experimental run allowing us measure the performance of each node and gain insights into the working of the network.

The simulator distributes bandwidth evenly between all connections and assumes that the bottleneck is always at the end-hosts' connection to their ISP. Download capacity, upload capacity, repayment ratio α, largesse rate β, and the one-time free credit γ can be set on a per-link basis. In practice, we used the same values of α and β for all links originating from a given node, while using a random value of γ to avoid synchronization artifacts when all peers accumulate enough largesse to download a packet simultaneously. We ran all of our experiments with a single seed and 100 peers who want to download the file. In each experiment we report average rates after the system has achieved a steady state.

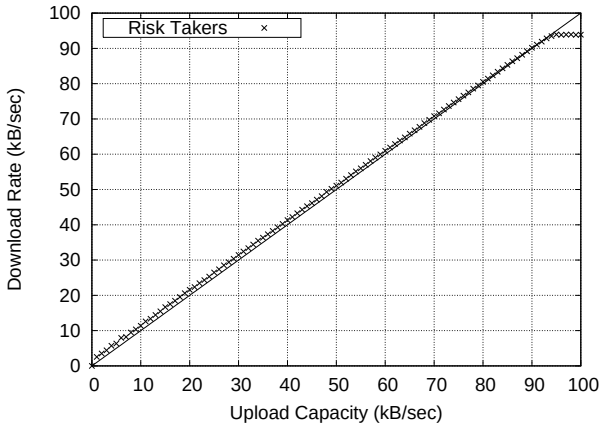


Figure 3.1: The download rate obtained by a peer as function of its upload capacity. Every "x" represents one peer. The diagonal line represents $x = y$.

While a 100 node network is relatively small, our goal in designing SWIFT is to study the feasibility of an effective incentive mechanism based on a pairwise algorithm in order to design a robust peer-to-peer streaming system. In our evaluation of our streaming system, later in this dissertation, we analyze much larger networks, including a network with over 100,000 nodes.

3.3.1 Download vs. Upload Rates

In our first experiment we show that peers have a strong incentive to upload as much as they can. All 100 peers used repayment ratio $\alpha = 1$, largesse rate $\beta = 0.01$, and a random one-time free credit γ between 1 and 2. All peers had download capacities of 100 kB/sec, but upload capacities were uniformly limited to values between 1 and 100 kB/sec. The topology used was a complete graph and the file had 100,000 packets.

Figure 3.1 shows the resulting download rates obtained by peers as a function of their upload capacity, with the straight line representing equal upload and download rates. It is evident that periodic risk-takers with upload capacities less than 94 kB/sec receive download rates comparable to their upload capacity, with most peers receiving slightly more than they upload because of the free packets they receive from the seed. In SWIFT, peers clearly have incentives to set high upload rates.

Peers with upload capacity greater than 94 kB/sec operate below slightly capacity. Our analysis indicates that this degradation is an artifact of the random packet picking strategy that we employed in our simulator. The problem would be mitigated if the packet picking algorithm were to take into account the frequency of packets in the system, with a bias towards rarer ones, similar to the rarest-first algorithm used by BitTorrent [19].

3.3.2 Paranoid Traders vs. Periodic Risk-takers

In Section 3.2.2 we claimed that in a mixed network of paranoid traders and periodic risk-takers, the risk-takers download faster. We modified the previous experiment to demonstrate that claim by changing half the peers into paranoid traders who did not upload a packet unless they first

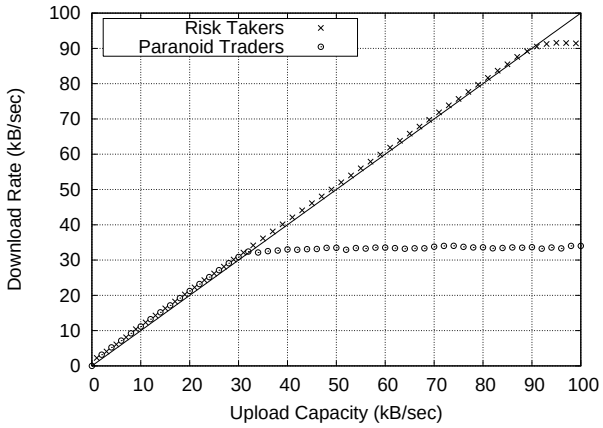


Figure 3.2: The download rate obtained by a peer by paranoid traders and periodic risk-takers as a function of their upload capacity. The diagonal line represents $x = y$. Paranoid traders are unable to utilize all their upload capacity.

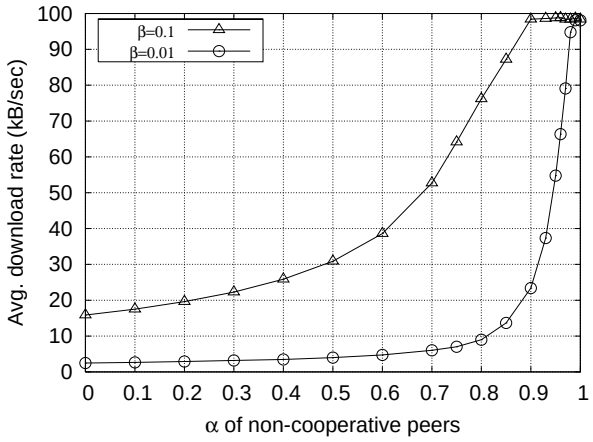


Figure 3.3: The average download rate received by non-cooperating peers as a function of their repayment ratio α.

received one.

Figure 3.2 shows the resulting download rates obtained by peers as a function of their upload capacity. The average download rate of paranoid traders was 28.5 kB/sec, whereas the average download rate of risk-takers was 50.5 kB/sec. We noticed that the paranoid traders traded with only with the risk-takers and thus downloaded at a much slower rate, as predicted in Section 3.2.2.

3.3.3 Effect of Non-Cooperative Peers

Our third experiment studied the behavior of non-cooperative peers that use repayment ratios α other than the default value of 1. As claimed in our analysis in Section 3.2.1, we show that a peer has very little incentive to deviate from the default behavior. Once again, we used a complete graph. Half of the peers were obedient and used $\alpha = 1.0$ whereas the remaining half used values uniformly distributed between from 0 and 0.99. In the first run all peers used a

Peer Behavior	Download Rate
Free-Rider	6 kB/sec
Periodic Risk-Taker	50 kB/sec
Seed	N/A

Table 3.2: Mean download rates of the various classes of nodes in a system with one seed, 50 free-riders and 50 periodic risk-takers.

largesse rate $\beta = 0.1$ whereas in the second run they used $\beta = 0.01$. All peers had an upload and download capacity of 100 kB/sec.

As shown in Figure 3.3, the download rate received by non-cooperative peers was much less than 100 kB/sec for peers with small values of α, but rose sharply as α approached 1. When $\beta = 0.1$, non-cooperative peers must still upload about 90% of what they receive in order to saturate their download link. With $\beta = 0.01$, the effect is more pronounced: non-cooperative peers must use a repayment ratio α very close to 1 to saturate their download link. Selfish peers are quickly penalized for their non-cooperative behavior.

3.3.4 Incentives to Prevent Free-Riding

In Section 3.2.3, we showed analytically that free-riders download at a much slower rate compared to periodic risk-takers. To demonstrate this, we ran an experiment of 100 peers with half of the peers free-riding and the other half being periodic risk-takers with upload capacity of 100 kB/sec, $\alpha = 1$, $\beta = 0.1$, and γ set randomly between 1 and 2.

Table 3.2 summarizes the download rates received by the various classes of nodes. We observed that the free-riders downloaded at only 6 kB/sec, whereas the periodic risk-takers downloaded at 50 kB/sec. Of the 6 kB/sec that free-riders received, 1 kB/sec was received from the seed, whereas 5 kB/sec was received from periodic risk-takers as predicted in Section 3.2.3.

3.3.5 Case for Non-Zero β

It is quite clear that having both $\beta = 0$ and $\gamma = 0$ will deadlock the system as no peer other than a seed will ever upload a packet. However, in this experiment we now demonstrate that a simple one-time credit is not sufficient to solve this problem, even when no packets are corrupted or lost in the network, and no peers leave the system till the end of the experiment.

We created a random graph with one seed, 100 other peers and an average node degree of 20. All peers have upload and download capacities of 100 kB/sec. We allowed the simulation to run for 30 second to achieve steady state and report average speeds over the next 60 seconds. In one run all peers used $\beta = 0$ and in the other they used $\beta = 0.01$.

Figure 3.4 shows the distribution of peers receiving various download speeds. In the absence of largesse, half the peers had download rates of zero and were deadlocked. These are peers that are not directly connected to the seed and no longer have packets that their neighbors were interested in trading for. Since these peers never again receive a free packet, they will never reach completion.

The dark bars show the resulting distribution when everyone risks just 1% of their bandwidth

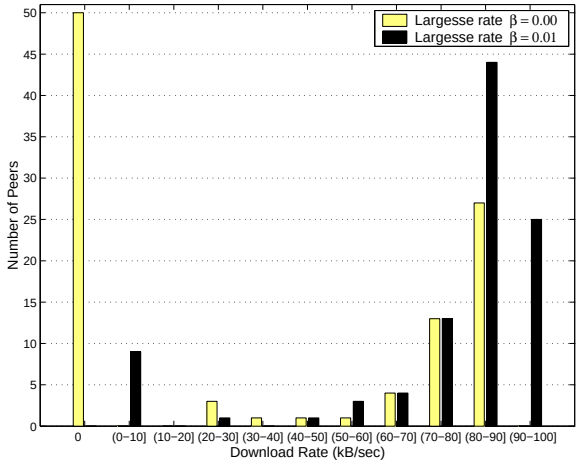


Figure 3.4: Distribution of peers by their download rate in two experiments. One, shown in gray, used $\beta = 0$ and the other, shown in black, used $\beta = 0.01$.

giving away free credit. Giving away this small among of free credit completely eliminates deadlocks. Moreover, with everyone making progress, peers are more likely to be able to trade with their neighbors, thereby increasing overall download speeds. With largesse, about a third of the peers were able to download at over 90% of their download capacity as compared to about 2% without. This shows that having all peers risk a small fraction of their bandwidth on giving away free packets not only improves overall system performance, but is also likely to bring a high return-on-investment for the peers themselves.

3.4 Conclusions

In this chapter, we studied incentive mechanisms for file sharing networks in order to build a platform for our work on incentives for live streaming systems. We outlined the file sharing model, and described various strategies that a node may adopt with regard to sharing it upload bandwidth. We also presented SWIFT, an incentive model for file sharing systems.

We argue that an overall upload to download ratio close to 1 is essential for system stability and prescribed the desired default behavior for nodes; we show that nodes have little incentive to deviate from the prescribed default parameters. We experimentally demonstrated that nodes that limit their upload bandwidth are penalized by their peers, and suffer a corresponding drop in their download bandwidth. Nodes are unable to free-ride the system without severely degrading their own download rates.

Thus, we have demonstrated an effective incentive mechanism that relies solely on local information, but promotes system-wide stability by incentivising nodes to contribute as much bandwidth to the system as they consume.

In Chapter 4 we describe Chainsaw, our streaming protocol based on an unstructured mesh network, with a similar request-response protocol that is amenable to pairwise incentives. In Chapter 5 we present the Token Stealing algorithm, our pairwise incentive mechanism for peer-to-peer streaming.

Chapter 4

Chainsaw: Incentives-Compatible P2P Multicast

Traditional tree-based streaming protocols create a parent-child relationship between nodes. On any link in the system, data flows in only one direction (i.e., from the parent to the child). This lack of reciprocal transfer makes it impossible to apply pairwise incentives.

Therefore, we adapt the simple request-response protocol used in SWIFT (and other unstructured file transfer protocols like BitTorrent) for streaming. In this chapter we present Chainsaw, and show it to be robust, and have good performance.

4.1 Design Goals

Our aim is to design a robust incentives-compatible peer-to-peer streaming protocol for applications like live audio and video. In order to give users a satisfactory experience, we need to provide a consistent, high quality data stream, with a quick ramp up time.

From the content provider's perspective, a good system must be scalable and robust to churn and other network conditions.

4.1.1 Compatibility with Pairwise Incentive Mechanisms

We have argued that an concrete incentive mechanism is essential in order to encourage nodes to contribute bandwidth to the system, thus promoting system scalability and performance. Moreover, a *pairwise* incentive mechanism is desirable, because of its simplicity and reliance on direct observation rather than information provided by third parties.

In Chapter 3, we have shown that a pairwise incentive mechanism can be effective. In this chapter, we show that with a few enhancements, a pull-based system can provide a high performance transport layer for streaming, while remaining compatible with pairwise incentives. In Chapter 5, we present the Token Stealing algorithm, our pairwise incentive mechanism for peer-to-peer streaming, based on the streaming protocol presented in this chapter.

4.1.2 Support Large Numbers of Simultaneous Participants

We aim to support distribution of content to large numbers of simultaneous participants. Therefore, an important goal is scalability with size. We show that our system is able to scale to large numbers of simultaneous participants, while still providing excellent performance.

4.1.3 Drive Packet Loss to Zero

Most multimedia codecs are highly sensitive to corrupted/missing data. The loss of even a small fraction of packets in an MPEG stream can cause severe distortion in the decoded video. While these effects can be mitigated by either relying on codecs that support graceful degradation, or with erasure coding as described in Section 2.5, both of these approaches come at the cost of an overhead in bandwidth.

A low packet loss rate allows a stream to be encoded with a lower level of redundancy, reducing the bandwidth overhead. Dedicating less bandwidth to redundancy allows us to support a higher quality stream for users with a given Internet connection speed.

4.1.4 Quick Startup Time

In order to facilitate browsing or "channel surfing" the media must begin to play within a few seconds at most. It would be completely unacceptable for a television set to take 30 seconds to switch channels.

4.1.5 Robust to Network Conditions

In a typical peer-to-peer network, nodes join and leave the system continuously. Therefore, our system must be robust to churn, and degrade gracefully with increasing levels of churn.

4.2 Protocol Design

We built a request-response based high-bandwidth data dissemination protocol called Chainsaw. Our protocol draws upon gossip-based protocols and BitTorrent. Like BitTorrent, we divide data into a series of packets with unique sequence numbers. The key difference between Chainsaw and BitTorrent is that whereas BitTorrent is designed to distribute a fixed-length file, there is a potentially infinite sequence of packets in Chainsaw. We do not require strictly sequential transmission of data, but allow reordering within a sliding window of sequence numbers.

4.2.1 Network Topology

Like BitTorrent, Gnutella and most Gossip-based protocols, Chainsaw is built on an unstructured random graph topology. This topology has the advantage of being flexible, easy to build and maintain, and robust to churn. A node that wishes to join the system must merely connect to a set of randomly chosen nodes in the system. We call the set of nodes a given node is connected to its *neighbors*.

A node may obtain a list of random neighbors in several ways. A highly decentralized solution would be to start with one node discovered through out-of-band means (for example, by word of mouth, or from a list of long-lived hosts running on well known addresses) and doing a random walk from there. Random graphs are expander graphs with high probability, so one may reach any node in the system with uniform probability in a logarithmic number of hops [43].

4.2.2 Membership Server

In our implementation we opt for the simpler centralized solution as used in BitTorrent. Nodes in the system periodically announce their existence to a *membership server*, which keeps track of all the nodes in the network. When a node needs new neighbors to connect to, it requests a list of random nodes from the membership server.

Although our implementation uses a single, central membership server, we do not consider this to be a major limitation for three main reasons:

1. A membership server only needs to maintain very minimal information about nodes in the system, for example, the IP address and port. Therefore, even with millions of nodes in the system, the memory requirements are modest.

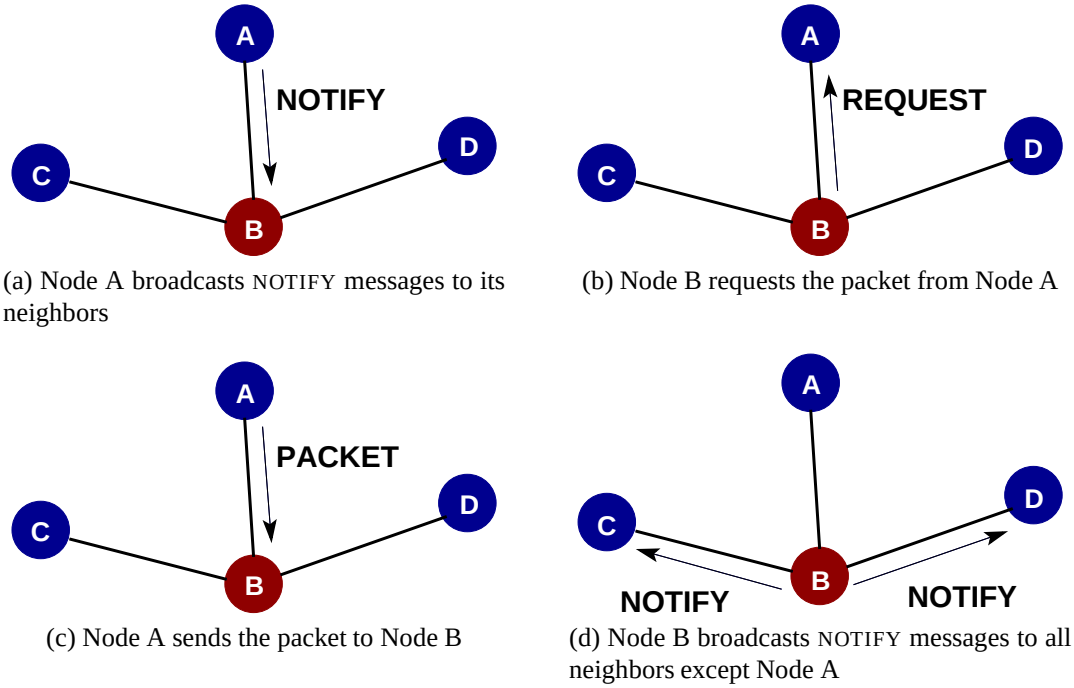
2. Nodes merely need to announce their presence to the membership server periodically. With a million nodes in the system sending keep-alive messages every 60 seconds with a 50-byte message, the bandwidth needed is only 6.67 Mbits/sec, which is available even from consumer-grade broadband connections.

3. It would be easy to extend the scheme to multiple membership servers which periodically exchange information with each other to balance the load out. The membership server is only used to find a random set of neighbors, so minor inconsistencies between instances is not a significant problem.

4.2.3 Data Dissemination

New data is injected into the system by a special node called the *seed*. The seed generates a series of packets with monotonically increasing sequence numbers. In general, multiple seeds can be supported with no changes to the protocol. Different seeds would only need to ensure that their sequence numbers are synchronized and all packets injected with a given sequence number are identical. The protocol can also be extended to support multiple channels or many-to-many multicast applications by replacing the sequence number with a $\langle$stream-id, sequence #$\rangle$ tuple. In this dissertation, however, we limit ourselves to a single channel system supplied by a single seed node.

Chainsaw's receiver-driven architecture eliminates the need for complex distributed routing algorithms. When a node receives a packet, it announces the availability of that packet to its neighbors by sending them NOTIFY messages. The availability information for a given packet is only of interest to a neighbor if that neighbor does not already have the packet. Therefore, nodes do not send NOTIFY messages to neighbors who have already announced availability of that packet.



(a) Node A broadcasts NOTIFY messages to its neighbors

(b) Node B requests the packet from Node A

(c) Node A sends the packet to Node B

(d) Node B broadcasts NOTIFY messages to all neighbors except Node A

Figure 4.1: Chainsaw uses a receiver driven request-response protocol. A node that receives a new packet broadcasts NOTIFY messages to its neighbors, who may then request the packet. Upon receipt of the packet, the node in turn will then broadcast availability of that packet to other neighbors.

As sequence numbers are small compared to the size of TCP/IP and other headers, it is wasteful to send a NOTIFY message as soon as a packet is received each time. Therefore, for each peer, we gather sequence numbers and send them together in one bulk message every *min-notify-interval* seconds. In our experiments we used a delay of 0.5 seconds, since that value would result in a payload size greater than the estimated size of TCP/IP headers in our setup. Longer delays bring diminishing returns in terms of overhead reduction.

For every neighbor, a node maintains a list of *desired packets*—packets the node needs to acquire, that the neighbor has available. A node normally picks a random packet from the list to request; but as discussed in Section 4.2.6, it may use a different strategy in the startup phase, shortly after joining the system. Having picked a sequence number, the node requests that packet from its neighbor via a REQUEST packet.

Nodes mark packets that have been requested from a neighbor to avoid wasting bandwidth by downloading multiple copies of the same packet. However, if a neighbor does not respond to a request after a specific timeout interval has passed, the packet is unmarked so that it may be requested from other neighbors.

On receiving a request from a neighbor, nodes may respond either by sending the data for that packet or with a NAK message, if they do not have enough bandwidth to satisfy the request. In addition, they may decline to fulfill a request from a given neighbor as determined by our Token Stealing incentive algorithm that we will describe in Chapter 5.

A node that receives a NAK message from a neighbor due to a lack of credit would most likely get a NAK again if it made another request too quickly. This might result in delays and

overhead as a node repeatedly requests packets from a given neighbor when it could request that packet from other neighbors instead. A fixed back-off interval would require fine-tuning based on many factors. To avoid this, we use an *adaptive* system. Every node calculates an *inter-request interval* (IRR) for each of its neighbors. This is the minimum interval of time that elapse between making two consecutive requests from a given neighbor. Nodes start at an arbitrary positive value of IRR and adapt the interval based on a multiplicative-increase, multiplicative-decrease (MIMD) strategy. When a node successfully receives a packet, it decreases IRR by multiplying it with a value between 0 and 1, and when it receives a NAK, it increases the value of IRR by multiplying it with a value greater than 1. Since the MIMD algoritm is merely an optimization for the requesting node, the MIMD parameters do not need to be tuned with great precision. In our implementation, we used values of 0.9 (i.e., a 10% reduction) and 1.5 (i.e., a 50% increase) respectively.

4.2.4 Seeding Strategy

When the seed creates a new packet, no other node in the network has a copy of that packet. We take advantage of this fact to reduce the propagation delay for packets with a *Seed-Push* strategy. When a seed generates a new packet, it immediately forwards it to a subset of its neighbors, and sets a *PushTTL* on the packet. Neighbors who received a pushed packet decrement the value of PushTTL and immediately forward the packet to one random neighbor.

This strategy ensures that new packets quickly propagate to a set of nodes in the system. Ideally the value of $PushTTL$ is comparable to the diameter of the network, ensuring that newly created packets are scattered well throughout the system and can spread quickly from there.

When a packet is newly created, it is rare in the system, so the probability of a node receiving duplicate pushes is quite low. Thus, the expected value of bandwidth wasted on duplicate packets received due to the Seed Push algorithm is low.

4.2.5 Buffer Management

Whenever a node receives a packet, either by a push or in response to a request from its neighbor, it adds the packet to its buffer. Whenever there is a contiguous block of packets at the start of the buffer, those packets are *emitted* (i.e., passed up to the application layer and removed from the buffer). Packets that have been emitted are called *old packets*. A node retains a certain number of recently emitted packets in order to serve requests made by its neighbors.

When a packet is added to the buffer, a node sets a timeout on that packet for *MaxBufferTime*. Once that timeout expires, all packets with lower sequence numbers are emitted immediately and those that have not been received yet are considered lost.

Lost packets can lead to disruption in the media stream, and should be avoided as far as possible. Packet loss rate is the primary metric we use in gaging the performance of our system.

4.2.6 Startup Strategy

As discussed in Section 4.1, one of our important design goals is to enable a new node that joins the system to begin playback as quickly as possible. We designed a startup strategy to help

achieve this goal.

When a node joins the system and starts connecting to neighbors, each neighbor will send the node a send a lists of packets available for download. The node must decide which sequence number to start downloading from. The range of packets available may be quite large, because the node's neighbors may not be precisely synchronized, and may offer a substantial number of old packets to its neighbors. The only limit to the range of packets available to a node is the amount of memory that the node's neighbors are willing to spend storing old packets.

Chainsaw's receiver-driven architecture means that the node has the flexibility to decide where in the range of available packets to begin downloading. Viewers who missed the start of a program may have the opportunity to "go back in time" a few minutes.

However, in our implementation, we assume that the goal is to receive data as current as possible. In achieving this goal, there is a trade-off between currency and the probability of "hiccups" in starting up because the very newest packets have not propagated to most neighbors. To avoid this situation, a node looks for a sequence of n consecutive packets with m or more sources each. In our experiments we looked for a sequence of 10 consecutive packets with 2 or more sources. We avoid packets with single sources, to avoid startup problems because of errant nodes, or nodes with limited upload capacity.

Having decided on the sequence number to start from, the node discards all information it has received about packets prior to that sequence number, and enters LINEAR mode. In this mode, the node always picks the packet with the lowest sequence number when deciding on a packet to request from each neighbor. This allows the node to ramp up quickly and pass up a block of data to the application layer. Once the node has played back a sequence of p packets, the node enters NORMAL mode, in which it starts requesting packets at random, as described in Section 4.2.3. In our experiments we set p to be equal to one second's worth of data.

In Section 6.3 we demonstrate experimentally that this startup strategy allows most nodes in the network to start playback within a few seconds.

Summary

In this chapter we presented Chainsaw, a pull-based peer-to-peer streaming network on top of an unstructured topology. The unstructured pull-based mechanism provides a bidirectional flow of data between pairs of interacting nodes: in general, a node will both send packets to, and receive packets from a given neighbor. This property makes the system amenable to pairwise incentive mechanisms, because a node may penalize a neighbor that does not upload data to it by refusing to answer that neighbor's request for packets.

Next, in Chapter 5, we present Token Stealing, an incentive mechanism for live streaming applications that takes advantage of this fact. In Chapter 6 we present detailed experimental results that show that the Chainsaw protocol supports high-bandwidth streaming with low packet loss and delay, and is highly scalable and resistant to churn.

Chapter 5

Token Stealing: Incentive Mechanism for P2P Multicast

In Chapter 4 we presented the design for a robust and effective live streaming protocol based on an unstructured network. In this chapter, we expand on that design and incorporate an incentive mechanism.

At first glance, one might assume that the same incentive mechanism as used in SWIFT will be effective for streaming, because the Chainsaw protocol is quite similar to the pull-based mechanism used in SWIFT. However, as we explain in Section 5.2, there are key differences between file sharing and live streaming as a result of a need to achieve a target bandwidth, and the limited time window over which data packets remain useful. We present our Token Stealing algorithm, our incentive mechanism for live streaming that overcomes these constraints.

In Section 5.1 we present our design goals. In Section 5.2 we explain the key differences that make incentive mechanisms designed for file sharing systems unsuitable for live streaming applications. In Section 5.3 we explain how a bandwidth allocation strategy instead of a strict tit-for-tat approach can help prevent under-utilization of resources. In Section 5.4 we describe our Token Stealing bandwidth allocation algorithm. Finally, in Section 5.5 we discuss possible strategies that selfish nodes may use to game the system, and how we can defeat those strategies.

5.1 Design Goal

A peer-to-peer network relies on bandwidth contributed by its participants. While there are often a number of *altruistic* nodes that will contribute resources willingly, many do not. We therefore wish to provide a concrete incentive for nodes to contribute at least as much bandwidth as they consume, thus imposing no net resource drain on the system. We provide this incentive by preferentially directing bandwidth at nodes who contribute the most, thus improving their performance.

A secondary goal is to take advantage of excess bandwidth provided by altruistic nodes to support nodes that are unable to contribute resources to the system, for example, nodes connected via an asymmetric DSL connection which offers a significantly lower upload capacity than download capacity. However, if there are not enough altruistic nodes to make up the resource deficit created by the non-contributors, we wish to ensure that the contributors get signifi-

cantly better performance. In Section 6.11 we show through simulations that our Token Stealing algorithm is able to achieve these goals.

5.2 Attempts to Adapt SWIFT to Chainsaw

We designed the Chainsaw streaming protocol to be receiver-driven and compatible with pairwise incentive mechanisms like our SWIFT protocol presented in Chapter 3. Therefore, we initially attempted to adapt the tit-for-tat scheme used in SWIFT to our Chainsaw live streaming protocol, and enforce a fair balance of trade between every pair of interacting nodes.

Although our this attempt was unsuccessful, we present a summary of our findings because they provide insight into key differences between the dynamics of file sharing and live streaming applications, and illustrate the need for a different approach.

5.2.1 Naïve SWIFT Algorithm

In SWIFT, every node maintains *credit* for each of its neighbors and honors packets requests only when the neighbor has enough credit. Whenever it receives a packet from a neighbor, the node extends it α packets worth of credit. In addition, trading is jump-started by initializing neighbors with γ packets worth of credit instead of zero, and deadlocks are avoided by periodically extending nodes a small fraction β of their total upload capacity in credit every second, regardless of data received from it.

So long as nodes consistently upload data to their neighbors, they will keep earning credit with their neighbors and be able to download packets from them. Nodes that do not upload will soon deplete their credit with their neighbors and not be able to download from them anymore, except for small trickle of free credit they receive from their neighbors in the form of β.

Although SWIFT was very effective at ensuring fairness in file-transfer applications, we found that mechanism to perform very poorly when applied to live streaming. In our simulations we found that over time, a large fraction of nodes started to suffer severe ($> 50\%$) packet loss even in a system where every node tried to upload as much as their capacity allowed. This was caused by small imbalances between nodes (e.g., due to different delay characteristics, distance from seed, number of neighbors) being amplified by an undesired positive-feedback loop.

Consider a pair of nodes A and B, where A is closer to the seed than B. In this situation, Node A is likely to receive new packets before Node B. As a result, Node B will have fewer opportunities to upload packets to Node A, resulting in a net loss of credit over time. Eventually, Node B will run out of credit with Node A and will no longer able to download from that node. This puts Node B at a significant disadvantage because Node A was most likely a source of packets of interest to Node B's other neighbors, given it proximity to the seed. Therefore, the loss of node A as a trading partner puts node B in a less favorable position to trade with the rest of its partners. This creates a positive feedback loop where a slight disadvantage is ultimately amplified to the point where a node is unable to earn enough credit to avoid packet loss.

5.2.2 Compensating for Trading Imbalances

We evaluated two strategies for correcting these trading imbalances. First, we implemented a system where nodes attempted to correct imbalances they encountered by allocating bandwidth to neighbors they had the most trade deficit with before other nodes, regardless of the order in which they received packet requests. We also evaluated a strategy where nodes that were at an advantage intentionally attempted to reduce the number of packets it uploaded in order to improve its trade balance with its neighbors.

Preferential Uploading

We found that nodes ran out of credit because they were unable to upload enough packets to some of their neighbors to maintain a stable supply of credit. Therefore, we implemented a system where nodes that were running out of credit with neighbors prioritized requests from those neighbors and satisfied them as quickly as possible. Quickly satisfying existing requests will generally result in more requests for packets from that neighbor.

Counterintuitively, this strategy made the problem *worse* over time. Some nodes, such as nodes that were closer to the seed nodes were at an advantage with respect to most of their neighbors. This led to a race among their neighbors to upload as quickly as possible to the advantaged node. This created a new positive feedback loop where advantaged nodes were put at an increasingly greater advantage by neighbors aggressively uploading packets to them. Eventually, the neighbors that lost the race ran out of credits as they did in the naïve tit-for-tat system.

Advantaged Nodes Back Off

Our next approach was to have advantaged nodes attempt to proactively reduce the amount of data they upload to give other nodes an opportunity to upload data and earn credit. By default, nodes send NOTIFY messages to all their neighbors when they receive a new packet in order to enable them to request that packet. Nodes that are closer to the seed will often receive packets before any of their neighbors, resulting in several requests for those packets. We compensated for this effect by reducing the number of NOTIFY messages sent by nodes when they detected that they had large amounts of unused credit with most of their neighbors.

Every node kept track of its mean upload-to-download ratio with its neighbors, which we called the *balance ratio*. Nodes with balance ratios below 1 continued to employ the default behavior of notifying every neighbor of new packets in order to maximize their chances of receiving packet requests and improving their balance ratio. However, as the balance-ratio increased above 1, nodes linearly reduced the number of neighbors notified. This ensured that the advantaged nodes did not receive a large number of requests for new packets, thus giving other, less advantaged nodes an opportunity to earn credit by uploading those packets after they received them.

This algorithm is beneficial to the advantaged nodes, disadvantaged nodes, and the system as a whole. The advantaged nodes benefit by being relieved of some of the burden of uploading packets. The disadvantaged nodes benefit by having more opportunities to upload packets to their neighbors and earn credit. The overall amount of upload bandwidth in the system is gener-

ally not reduced because some of the burden of uploading packets is shifted from the advantaged to the disadvantaged nodes.

This algorithm was effective in correcting imbalances in initial experiments with small, uniform networks with no churn. However, in experiments with more realistic scenarios, we found that this algorithm was insufficient to prevent the cascading failures we observed before. We were unable to compensate for larger imbalances without throttling the advantaged nodes down to the point of causing significant system-wide performance degradation.

5.2.3 Lessons Learned: Need for a Different Approach

As a result of insights gained in these studies, we concluded that a SWIFT-like tit-for-tat approach was unsuitable for live streaming, and a different approach was needed for three key reasons.

Firstly, in a live streaming system, packets have a limited useful lifetime because nodes are only interested in a small window of data at any given time. In general, older packets are neither useful for playback nor as trading commodities because all participants are approximately synchronized. A packet in a file sharing network, the other hand, remains useful as a trading commodity until every node in the network has obtained a copy. If new nodes join continuously, packets always remain useful, and a node will ultimately be able to trade them for other packets and make progress towards assembling the complete file.

We found that the limited useful life of packets makes systems that enforce strict pairwise equality unstable. A node may tend to receive packets slightly later on average than other nodes as a result of its connection speed, network latency or position in the network. As a result, this node may often be unable to upload packets to its neighbors even if it is willing to because its neighbors would already have obtained those packets from other sources. The node will eventually run out of credit with its neighbors and get locked out. Unlike the file sharing network, a small largesse rate will not help alleviate this problem, because the node will not be able to trade the largesse to obtain a net positive flow of credit, and the node effectively gets locked out of the system. Once this happens, a different node is now the slowest node, and eventually gets locked out too. This eventually leads to vastly reduced system performance.

Secondly, in a live streaming system, data must be downloaded at an average rate equal to the stream rate, or it will lead to degraded user experience (either through repeated re-buffering, or packet loss). With a file sharing network, on the other hand, slower download speeds, while undesirable, still contribute to the node's ultimate goal of acquiring the file being traded.

Finally, peers in a file sharing network attempt to download the file as quickly as they can. The ideal download rate is infinity. Although peers have a higher download capacity than upload capacity, there is no spare capacity in the system, because every bit of available bandwidth could be used to improve the download speed of some participant. With a streaming system, however, the ideal rate is equal to the rate at which the source generates new data (i.e., the *stream rate*). As a result, altruistic nodes who contribute more bandwidth than they consume contribute to surplus system capacity. This surplus can be used to support nodes that are unable to contribute upload bandwidth at the full stream rate.

Whereas the first two differences pose additional challenges compared to file sharing networks, the third offers an opportunity to relax the need for strict pairwise fairness and take advantage of any surplus capacity in the system.

5.3 Bandwidth Allocation Strategy

In order to overcome the challenges posed by live streaming, we shifted our focus from the strict pairwise fairness used in SWIFT to a bandwidth *allocation* strategy. Instead of limiting upload rates, nodes allocate their upload capacity between competing neighbors based on the amount of data they receive from those neighbors. This permits effective utilization of any surplus capacity that may exist in the system as a result of altruistic nodes, promoting system performance and stability.

By utilizing all available resources, we maximize the number of participants that can be supported. Excess bandwidth provided by altruistic nodes can be leveraged to forgive nodes with low upload rates so long as the system remains resource-rich. When there is insufficient bandwidth to support all nodes in the system, however, nodes with high upload rates receive much higher bandwidth than the low-bandwidth nodes.

This behavior gives nodes an incentive to contribute as much upload bandwidth to the system as they are capable of. Users of the system would be encouraged to increase their upload rates, for example, by relaxing artificial constraints on upload bandwidth, or by closing other applications that consume upload bandwidth.

5.4 Token Stealing Algorithm

Our Token Stealing algorithm builds on the standard *token bucket* model [90] commonly used to regulate bandwidth flow in networking applications and routers. Our Token Stealing algorithm sets up local markets at every node where neighbors compete for the node's upload capacity. When the demand for bandwidth at a node exceeds the node's upload capacity, neighbors that have been uploading the most data to the node receive preferred service. This constraint is relaxed when there is enough bandwidth to fulfill all requests.

Standard Token Bucket Algorithm

The token bucket algorithm works by having a virtual bucket into which tokens are added periodically. Whenever a packet is transmitted, an equivalent number of tokens must be removed from the bucket—packets may only be transmitted when there are a sufficient number of tokens available in the bucket. Thus, the overall bandwidth can be controlled by controlling the rate at which tokens are added to the bucket.

The number of tokens that may accumulate in the bucket is limited to some maximum value to prevent a large number of tokens from accumulating during periods of low demand and causing a large burst during a subsequent period of high demand. The basic token bucket algorithm only ensures that the overall bandwidth does not exceed a specified limit.

Our Extension: Token Stealing

Our Token Stealing algorithm is a straightforward extension of the token bucket algorithm. Every node maintains a standard token bucket that we refer to as the *shared bucket* into which tokens are added periodically, at a rate equivalent to the node's upload capacity. In addition, the

node maintains a separate bucket for each of its neighbors. We refer to these as *private buckets*. Whenever a node receives a packet from one of its neighbors, it removes tokens from the shared bucket and transfers them to that neighbor's private bucket. This has the effect of reserving a portion of the node's upload bandwidth to repay the neighbor for the packets it has uploaded.

Like the shared bucket, private buckets are limited in size. This prevents neighbors from reserving large amounts of bandwidth that they never utilize (for example, because they are connected to other nodes with large upload capacities). Tokens that overflow the private buckets are returned to the shared bucket.

A neighbor may download a packet so long as there are enough tokens between the shared and private buckets. The maximum number of bytes B_i a neighbor i may download at any given time is:

$$B_i = \min(P_i, S + \sum_{k=0}^{N} P_k)$$

where S is the number of bytes in the shared bucket, P_i is the number of bytes in neighbor i's bucket and N is the total number of neighbors.

Note that unlike the standard token bucket algorithm, the number of tokens in the shared bucket may go negative with Token Stealing because of transfers to private buckets. In this case, it is necessary to check to total number of bytes in all private buckets to prevent the overall upload rate from exceeding the rate of addition of tokens to the shared bucket (i.e., the node's upload capacity).

If the number of bytes available to the neighbor is greater than the size of the packet requested, the node deducts the appropriate number of tokens, and transmits the packet. The algorithm used to deduct the tokens from the shared and private buckets is discussed in Section 5.4.1. If the neighbor has insufficient tokens available to satisfy the request, the node sends a NAK message.

As discussed in Section 4.2.3, on receiving a packet or NAK message, nodes adjust the rate at which they request packets from their neighbors based on an MIMD strategy. It has been shown [3] that MIMD strategies lead to unfair allocation of bandwidth in situations where congestion signals (NAK messages in this case) are synchronous. It is highly likely that nodes with similar contribution levels (number of tokens in their private buckets) run out of available tokens simultaneously leading to synchronous congestion signals.

To mitigate this, we use a scheme similar to the Random Early Detection [33] congestion avoidance scheme used by IP routers to avoid congestion in TCP traffic. When the number of available bytes falls below a certain threshold T, we deny the request with a NAK message with probability p even if the neighbor has sufficient tokens available. The value of p is zero then $B_i \geq T$ and linearly increases to 1 as B_i goes to 0. This probabilistic signaling does not reduce the upload rate in the long run compared to deterministic NAKs, but it ensures that different neighbors that are close to exhausting their credits are denied at slightly different times, thus preventing undesirable synchronization between neighbors.

5.4.1 Which Bucket First?

The question of which bucket to deduct tokens from when a neighbor requests a packet is interesting. One may choose to deduct tokens from the private bucket first and dip into the shared

bucket only if there are not enough tokens in the private bucket, or one may use up tokens from the shared bucket first.

In our experiments we found that the both strategies give the neighbors that upload (and therefore have tokens in their private buckets) an advantage, but that advantage is considerably greater in the latter case. When tokens are deducted from the private buckets first, neighbors that upload do not compete in the market for shared tokens unless their private buckets are empty. This makes it easier for neighbors that do not upload to receive a portion of the bandwidth.

When tokens are deducted from the shared bucket first, all neighbors compete equally in the market for shared tokens before dipping into their private buckets, which act as a reserve. This amplifies the priority given to the neighbors that upload the most packets to the node. Therefore, the strategy we choose is to deduct tokens from the shared bucket first and only dip into the private bucket when the shared bucket is empty.

5.4.2 Analysis

With our Token Stealing algorithm, the total upload capacity of the node is still limited by the rate at which tokens are added to the token bucket (i.e., the upload bandwidth limit). However, unlike a simple token bucket system where all neighbors have an equal opportunity to use up tokens from the bucket, our Token Stealing algorithm favors neighbors that upload the most packets to the node.

Whenever a neighbor uploads a packet to a node, the node reserves tokens for that neighbor's use. Every packet the neighbor uploads to a node increases the chances that the neighbor will be able to download a packet in the future.

If all neighbors upload equally, all private buckets will have the same number of tokens in them, which gives all neighbors equal priority. However, a neighbor that does not upload will not have tokens in its private bucket and will be limited to competing with other neighbors for tokens from the shared bucket.

Whether or not the non-uploading neighbor succeeds in downloading depends on the total supply and demand at that node:

Node has excess upload capacity

If the node has more than enough upload capacity to fulfill the demand of all of its neighbors, the shared bucket will have tokens in it and the neighbor that does not upload will still be able to download. This ensures that a node's upload capacity is utilized as much as possible.

It is possible for a few nodes, known as *free-riders*, to try to leach off the system by selectively connecting to nodes with excess capacity. This strategy will work so long as the number of free-riders is small. If a large number of nodes attempt to leach off the system, they will compete among each other for tokens from the shared token bucket. This makes the effect of free-riders self-limiting.

Node has limited upload capacity

If the node does not have enough capacity to satisfy all requests, most of the tokens will be moved to the private buckets of the neighbors that do upload, and the shared bucket will gener-

ally be empty. As a result, the neighbors that upload will be able to use the tokens from their private buckets to download packets, but nodes that do not upload will be forced to compete for the scarce tokens from the shared bucket.

5.5 How Our Algorithm Prevents Gaming

In this section, we briefly discuss various ways in which nodes may attempt to game the system, and explain why those attempts would be largely unsuccessful, and not compromise the scalability or stability of the system.

5.5.1 Misreporting Information

Our system does not rely on nodes to report any information about themselves other than the availability of packets. This makes it difficult for malfunctioning or selfish nodes to gain an unfair advantage over their neighbors by lying about their upload rates.

A node may announce the availability of packets it does not have, but this does not confer any advantage, because their neighbors will not move any tokens to their private buckets until requests are fulfilled. As we discuss in Section 5.5.4, a node that attempts to earn private tokens by uploading fake data will easily be discovered.

5.5.2 Selectively Connecting to High-Bandwidth Nodes

A node may attempt to game the system by selectively connecting to high bandwidth nodes. It might seem that this strategy will allow the node to gain more than its fair share of the bandwidth provided by that node. However, in a resource-constrained system, the shared bucket will typically be empty. The amount of bandwidth received by a node will be dominated by the private credit it receives for uploading data to that node.

In Section 6.15 we experimentally demonstrate this fact, and show that there is no performance gained by nodes adopting this strategy whether just a few nodes adopt it or a vast majority.

5.5.3 Sybil Attacks

A Sybil Attack [24] is an attack where a node attempts to improve its performance by assuming many identities in the system. This might allow a node to, for example, escape consequences for its bad behavior.

However, our system does not rely on building long-term reputations for nodes. Instead, every node is judged by its peers on its recent upload rates. If a node builds up tokens in its private bucket by uploading rapidly for a while, those tokens will soon be depleted if it stops uploading. This lack of long-term memory renders Sybil attacks against our system ineffective. An attacker would gain little by assuming several identities in the system, and would be better off pooling upload resources into a single identity in order to gain the most private tokens with its neighbors.

5.5.4 Uploading Bogus Data

A node may attempt to gain an unfair advantage by advertising packets it does not actually possess, and uploading bogus data if those packets are requested by neighbors. This will, of course, cost a node upload bandwidth, but a node may attempt to do so either out of a malicious desire to harm the system, or to minimize its delay by earning private tokens from its neighbors.

This problem can be solved by using cryptographic techniques to authenticate the data they received from neighbors. BitTorrent solves this problem by distributing cryptographic hashes of the individual pieces of the file in the *.torrent* file. Unfortunately, this solution cannot be used with a streaming application, because the number of packets is potentially infinite.

Instead, the seed can digitally sign packets on the fly as it injects them into the network, and attach the digital signature to every data packet. The public key needed to verify the signatures may either be distributed to nodes in a metadata file analogous to the *.torrent* file, or by a certificate server. For example, the DSA algorithm specified in the FIPS 140-2 [70] standard produces 320 bit signatures, which amounts to a modest 4% overhead if the stream uses 1,000 byte packets. The DSA algorithm also does not require excessive CPU resources. In a test using the OpenSSL [92] implementation of DSA under Ubuntu Linux 10.10 on a dual-core AMD Athlon 4400+ at 2.3 GHz, we were able to perform 5,619 SIGN and 4,916 VERIFY operations per second. At 25 packets per second, this would result in an CPU load of under 0.5%.

Summary

In this chapter, we argued for a different incentive model for live streaming application than the strict tit-for-tat approach advocated for file sharing applications. We presented the Token Stealing bandwidth allocation algorithm that gives nodes an incentive to contribute as much upload bandwidth as they can, while forgiving nodes (like ADSL) nodes when other altruistic nodes make up the deficit by uploading more than their fair share. We discussed various strategies selfish nodes may attempt, and how those strategies can be defeated.

Next, in Chapter 6 we present extensive experimental results that demonstrate the effectiveness of our Token Stealing algorithm.

Chapter 6

Experimental Evaluation

In this chapter we present detailed experimental evaluation of both the basic Chainsaw streaming protocol and the Token Stealing algorithm with a discrete-event simulator. We also built a prototype implementation of our system to validate our simulation model.

In Section 6.1 we describe our simulator model along with the parameters of the experiment. In Section 6.2 we provide a more detailed overview of the experimental results presented in this chapter. In Section 6.3 we study the performance of a system with reasonable default parameters which serves as the basis for other experiments in this chapter. In Section 6.4 we demonstrate that our system scales well with increasing network size. In Section 6.5 we demonstrate scalability with increasing bandwidth, whereas in Section 6.6 we evaluate the effect of changing the packet size while keeping the bandwidth constant. In Section 6.7 we demonstrate robustness to churn. In Section 6.8 we show that our system performs well over a wide range of network latencies. In Section 6.9 we study the effect of network graph degree.

In Section 6.10, we evaluate the effect of our Token Stealing algorithm in a resource-rich system, whereas we study progressively resource-starved systems in a series of experiments in Section 6.11. In Section 6.12 we show that the system copes well with changes in resource availability. In Section 6.13 we demonstrate that the system also reacts quickly to changes in an individual node's upload rates, and nodes are promptly rewarded for increasing their upload rates and penalized for decreasing them. In Section 6.14 we show that the time taken to respond to changes in node behavior can be adjusted by varying the private bucket limit parameter.

In Section 6.15 we evaluate a possible strategy that could be attempted to game the system in order to gain an unfair advantage by taking advantage of altruistic nodes, and show that this strategy does not benefit the selfish nodes.

In Section 6.16 we study a more realistic scenario where upload rates fall into a range of values rather than a few distinct classes of parameters. Finally, in Section 6.17 we validate our simulation results by presenting results from experiments performed with a prototype implementation on the PlanetLab testbed.

6.1 Simulation Model

We built a high-performance discrete event simulator in order to enable us to simulate large networks with a diverse range of parameters. The code is highly optimized and scalable, while

accurately simulating the application layer protocol. Our largest experiment had a mean network size of over 100,000 nodes, with about 1.8 million nodes participating over the course of the experiment, having about 14.2 billion messages routed between them. That experiment ran over the course of two weeks and consumed 14 gigabytes of memory. Smaller experiments, with a few thousand nodes required under a gigabyte of memory and only a few hours of CPU time, thus enabling us to run many experiments and study a vast range of system parameters, within a reasonable amount of time.

On the whole, the experiments presented in this chapter represent the analysis of 62 gigabytes of raw experimental results gathered from over 400 individual network simulations and more than 4,000 hours of CPU time. Over the course of these experiments, we simulated more than 22 million nodes, with about 21 trillion messages exchanged between them, representing nearly 1.1 petabytes of network traffic.

The simulator consists of over 2,700 lines of C++ code, and features a highly flexible XML-based configuration system to allows us to tweak every parameter of the system as desired, and specify a wide range of network and node behaviors without writing additional code. Figure 6.1 shows a block diagram of our simulator. For scalability and speed, our simulation operates at the level of application-level messages rather than TCP/IP packets.

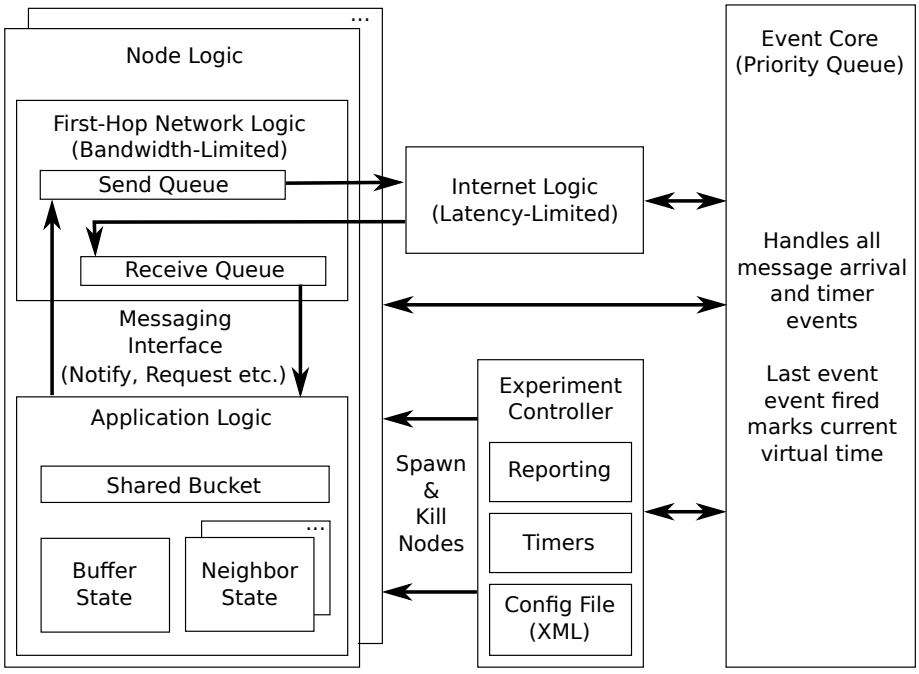


Figure 6.1: The simulation model: Nodes have independent upload and download queues. These queues are limited by bandwidth. The network core is considered to be adequately provisioned and congestion-free, thus solely limited by latency.

Every node in the network represents one Internet host. Every node has independent upstream and downstream queues which represent a user's connection to their Internet Service Provider (the first hop). The bandwidth limits can be set independently on the two queues. Since the links from a consumer to the ISP's local point-of-presence (POP) tend to be short, we neglect the latency on these links.

Section	Network Characteristic
6.3	Base system: Basis for remaining experiments
6.4	Network size varying from 25 to over 100,000 nodes
6.5	Stream rate (bandwidth)
6.6	Packet size
6.7	Mean node lifetime (churn rate)
6.8	Inter-node network latency
6.9	Number of neighbors (graph degree)
6.10	Token Stealing Enabled?
6.11	Fraction of ADSL nodes (resource availability)
6.12	Changes in resource availability
6.13	Changes in node upload rates
6.14	Stabilization time: private bucket limit
6.15	Attempt to game the system through selective connection
6.16	A system with a range of upload rates
6.17	Prototype implementation on PlanetLab

Table 6.1: Summary of experimental evaluation. We begin with a typical system configuration, and use that as a basis for further experiments by systematically varying each parameter one at a time. In addition, we study strategies that may be attempted to game the system.

Having traversed the upstream link, packets reach the "Internet cloud" which represents the core of the Internet. Studies have shown that the backbone links on the Internet typically are well provisioned and delays are dominated by the speed of light [34]. Therefore, in our simulation we assume that the core of the network is adequately provisioned and has a significantly higher capacity than the edge. A packet's traversal time through the core is solely limited by latency. Once it arrives at the destination node's downstream link, propagation time is once again determined by bandwidth.

Although we use a somewhat simplified network model, we do not simplify the application-level protocol in any way. Nodes are logically isolated from each other, and can only gain information about other nodes through application-level messages. We simulate all aspects of the protocol described in Chapters 4 and 5.

6.2 Overview of Experiments

In this chapter, we systematically study every aspect of the Chainsaw mesh-based streaming system, as well as the Token Stealing incentive mechanism. Table 6.1 shows a summary of the experiments evaluation we performed, and the sections in which we present the results.

First in Section 6.3, we demonstrate that Chainsaw accomplishes the primary goal of delivering a live stream to a large number of simultaneous clients with very low packet loss rates. We do this by running an experiment where we pick reasonable values for all parameters and measure the performance of a dynamic network with an average of 800 nodes. Through this experiment, we also show demonstrate that we meet the secondary goals of quick startup time, low delay, and low overhead. In order to provide a second point of reference, we repeat this

experiment with ten times as many nodes (i.e., 8,000 nodes).

Then in Sections 6.3 through 6.9, we show that our system is stable and performs well across a wide range of network and system parameters. We use our initial experiment as a basis for systematically studying every parameter by running a series of experiments where we vary a single aspect of the system and study its effects. We focus on our primary performance metric, packet loss rate, but also present graphs for delay or overhead as needed to help explain the packet loss rate curves we observed. As before, we repeat all experiments with both 800 and 8,000 node networks.

Next, in Sections 6.10 through 6.16, we move away from resource-rich networks to resource-constrained networks where there is insufficient cumulative upload bandwidth to support all clients in the system with no packet loss. In these experiments, we show that our Token Stealing algorithm plays its part in fairly allocating the available bandwidth to nodes that contribute resources to the system. We also show that our Token Stealing algorithm adapts quickly to varying network resource availability conditions, and to various ways in which selfish nodes may attempt to game the system through strategic behavior.

Finally, in Section 6.17 we present results from experiments performed on the PlanetLab testbed, using our prototype implementation. We show that we obtain similar results with the prototype implementation as we do with our simulations, thus validating our simulator.

6.3 Performance of a Typical Network

In this section we demonstrate that Chainsaw offers low packet loss, low delay, quick startup, and acceptable bandwidth overhead, by studying a network with reasonable default values. This setup forms the basis of the remaining experiments in this chapter, as we systematically vary isolated aspects of the system to study their effects.

We simulated a 200kbps stream which is comparable to moderate quality video streams (such as those offered by YouTube), as well as high-quality audio streams (many radio stations offer 128kbps or 160kbps MP3 streams). We divide the stream into 1,000 byte packets, giving a rate of 25 packets/sec. We individually vary and study the effects of stream rate and packet size in Sections 6.5 and 6.6 respectively.

In this experiment we assumed that the system is well provisioned with altruistic nodes who contribute more bandwidth to the system than the stream rate, and all nodes have sufficient upload and download capacity. We study several aspects of resource-constrained systems later, in Sections 6.11 through 6.16.

We set the node upload and download bandwidth to twice the stream rate, or 400kbps. The seed node is responsible for quickly injecting new packets into the system. In addition to answering normal packet requests, the seed must also push copies of each packet to its neighbors. In order to reduce the chance of a node leaving the system after receiving a packet but before passing it on to other nodes, it must push at least two copies. Moreover, since the seed always has packets available to it that no other node does, it is likely to service more requests than other nodes. Due to these extra bandwidth demands, we set the seed's upload bandwidth higher, at six times the stream rate (twice the requests as other nodes, plus two copies of each packet pushed preemptively), or 1.2Mbps. It does not seem unreasonable to assume that the publisher of a stream will provide higher bandwidth than a typical participant.

In addition to actual stream data (i.e., the payload), nodes must transmit and receive control traffic such as packet availability data (NOTIFY messages) and packet requests (REQUEST messages). Moreover, like any other application, we incur additional overhead from TCP/IP headers. In our system we use the same connection for both data and control traffic. Therefore, nodes must set aside some portion of their bandwidth for control traffic. Since the BitTorrent documentation suggests setting the upload rate at 80% of the raw upload capacity [23], leaving 20% for control traffic and other overhead, we use the same value. Thus the maximum upload rate for the seed was 960kbps while that of the non-seed nodes was 320kbps. For brevity, when we use the term *upload rate* in this chapter we refer to this value rather than the raw upstream capacity. We experimentally show that the actual overhead is much lower (about 7%) with this setup, so the 20% margin is quite conservative.

We set the latency between nodes to 100ms (200ms round-trip). This is an extremely pessimistic value, comparable to the transit time between nodes across the world. By contrast, typical round-trip time between Internet hosts in nearby cities tends to be on the order of 25ms and the round trip time between New York and California is on the order of 100ms. We study the effect of different values of network latency in Section 6.8.

We began the experiment with only the seed node, and then added other nodes to the system. Non-seed nodes then joined the system with a standard Poisson arrival process with a mean inter-arrival time of 125ms. The seed node had an infinite lifetime and remained in the system for the entire duration of the experiment, while other nodes had a mean lifetime of 100 seconds with an exponential distribution. This results in a mean network size of 800 (100sec / 125ms).

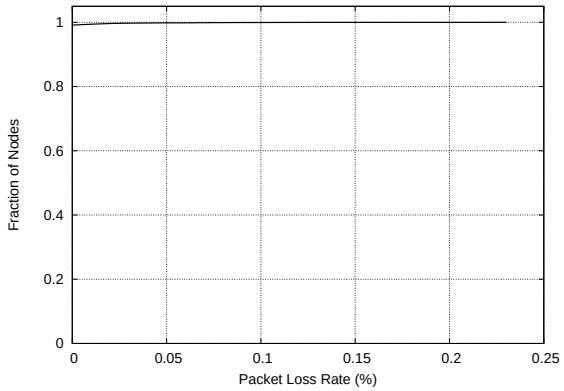
We consider a mean lifetime of mean lifetime of 100 seconds to be very conservative; the Web analytics company comScore has found that the average length of videos watched on the Web is 228 seconds [21]. Moreover, although the exact values we used for lifetime and network size are somewhat arbitrary, we systematically vary each of these parameters to study the effects of network size in detail in Section 6.4, and mean node lifetime in Section 6.7. Additionally, we repeated each experiment with a mean inter-arrival time of 12.5ms to give a network size of 8,000 nodes, in order to provide a second point of comparison with respect to scalability.

Every node connected to 20 neighbors. This is comparable to the default values used in other networks like Gnutella and BitTorrent, which use a similar network topology. We study varying network degree (i.e., number of neighbors) in Section 6.9.

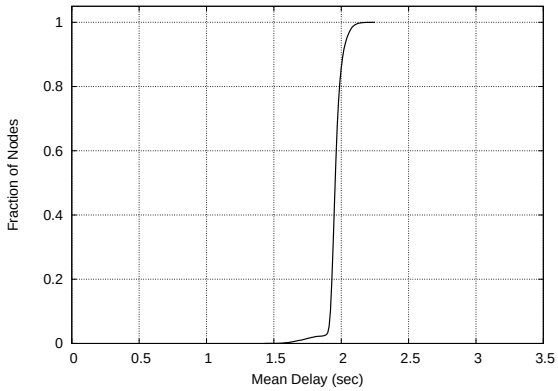
We ran the experiment for a total of 30 minutes of simulated time, and started gathering data once the system size reached its steady state, about five minutes (simulated time) into the run. Nodes do not report packet loss until the packet requests time out, so nodes with very short lifetimes may report artificially low loss rates. Therefore, in the interest of being conservative, we disregard results from nodes with lifetimes under 10 sec. Whereas this experiment studies the network in a steady state, we also study dynamic network behavior later in this chapter by varying network characteristics over time in Section 6.12, and varying node characteristics over time in Section 6.13.

In order to provide a second point of comparison, we also repeated every experiment with a network 10 times the size of the previous setup, or 8,000 nodes. Comparing the two results reinforces the fact that our system scales well with network size.

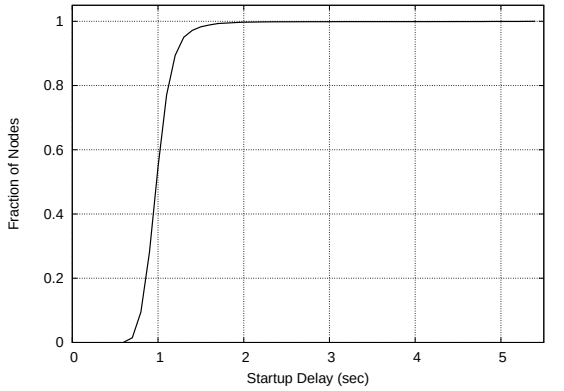
Figure 6.2a shows the distribution of packet loss rates of nodes in the network. Of the 11,070 nodes who participated in the system over the course of the experiment, 10,949 (98.9%) nodes



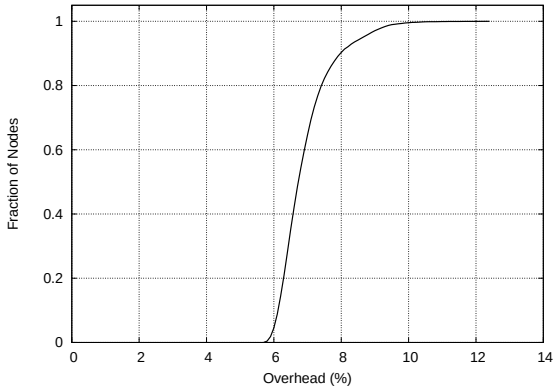
(a) Cumulative Distribution of Packet loss rate



(b) Cumulative Distribution of Mean Delay



(c) Cumulative Distribution of Startup Time

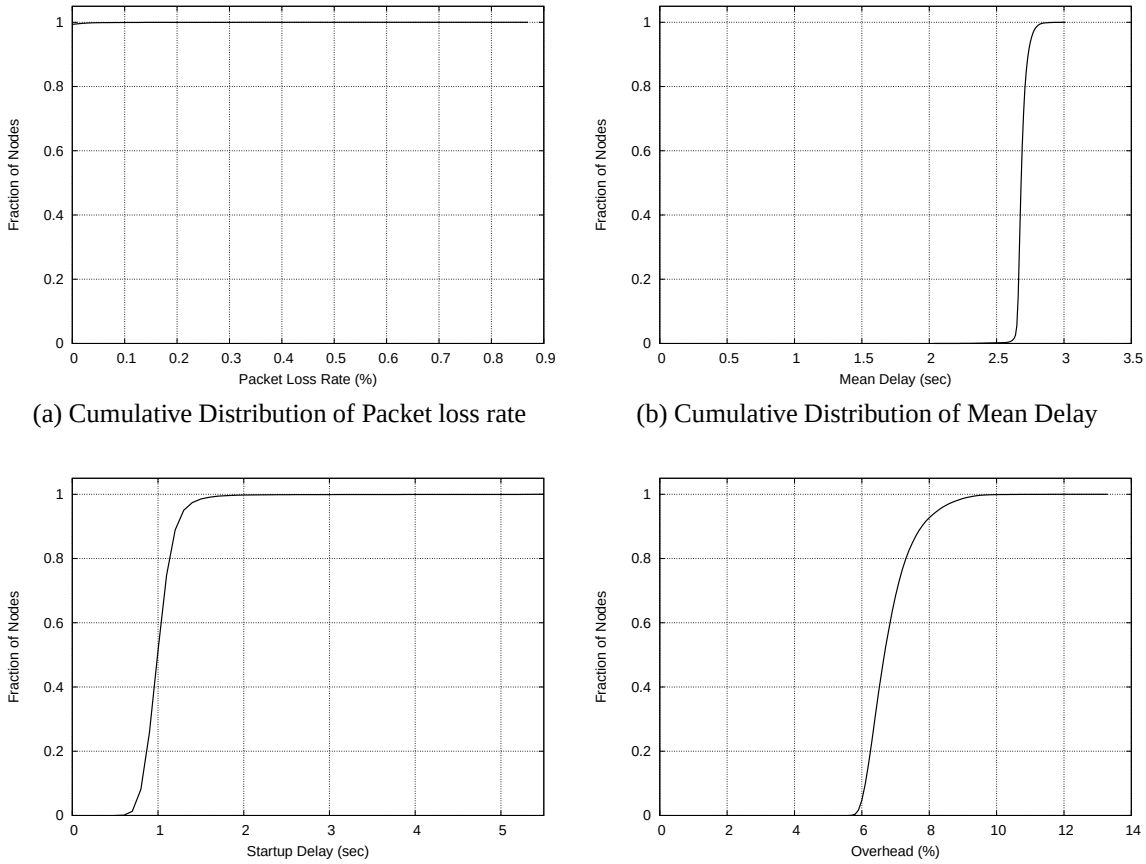


(d) Cumulative Distribution of Overhead

Figure 6.2: Cumulative distribution of packet loss rate, mean delay, startup time and overhead in a typical network (800 nodes).

had perfect delivery and did not lose a single packet. The remaining nodes lost between 1 and 4 packets over the course of the entire experiment, for a system-wide average packet loss rate of only 0.0005%. Figure 6.3a shows the corresponding graph for the experiment with 8,000 nodes. Once again, we find that a large majority of nodes suffer no packet loss at al. Of the 108,619 nodes who participated in this system, 107,924 (99.4%) suffered no packet loss, and the system-wide packet loss rate was 0.0003%.

Figure 6.2b shows the distribution of mean delay as measured from the time a packet is generated by the seed to the time a node receives it. Mean delays are distributed on a narrow bell curve with a mean of 1.80 seconds. Over 95% of the values are between 1.75 sec and 1.90 sec. The highest mean delay experienced by any node in the system was 2.007 seconds. The long tail between 1.23 and 1.7 sec is caused by the few nodes that are close to or directly connected to the seed, and thus receive packets very quickly. This means that most nodes do not lag far behind the seed. When the network size was increased to 8,000 nodes, the mean delay increased to 2.69 sec, as seen in Figure 6.3b. This increase is because the mean distance to the seed increased from 2.23 ($log_{20}800$) to 3 ($log_{20}8000$) as a result of the increased network size with a constant degree of 20.



(a) Cumulative Distribution of Packet loss rate

(b) Cumulative Distribution of Mean Delay

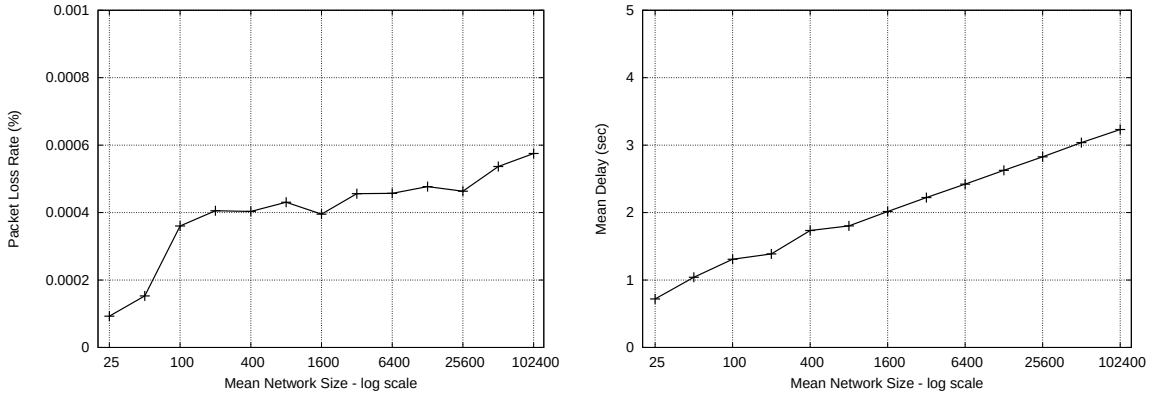
(c) Cumulative Distribution of Startup Time

(d) Cumulative Distribution of Overhead

Figure 6.3: Cumulative distribution of packet loss rate, mean delay, startup time and overhead in a typical network (8,000 nodes).

Figure 6.2c shows the distribution of startup times of nodes in the network. The startup time measures the shortest time after joining a network that the node could begin playback and avoid "re-buffering" events. A vast majority of nodes (9,815, or 88.7%) had startup times of under a second. The system-wide mean was only 0.80 seconds. This implies that nodes are able to being play back soon after joining the system and do not have to spend a long time waiting for the buffer to fill up, making for a good user experience. As with the mean delay, the mean startup time increased as a result of an increased number of hops to the seed when the network was expanded to 8,000 nodes. As seen in Figure 6.3c, the mean startup time for the larger network was 1.06 seconds.

Figure 6.2d shows the distribution of overhead. The overhead includes all non-data traffic received by a node, including notify messages, packet request messages, and per-packet overhead (such as headers). The mean overhead is only 6.7%, with a vast majority (over 90%) of nodes having overhead between 6 and 10%. We do not expect increased network size to have a significant impact on overhead, since nodes only communicate with other nodes in their vicinity, and there is no global control traffic. As expected, Figure 6.3d shows that the overhead remained essentially unchanged with the 8,000 node network, at 6.8%.



(a) Packet Loss Rate (b) Mean Delay

Figure 6.4: Performance of the system with varying network size. Figure (a) shows the packet loss rate as a function of the network size. Packet loss rates do not increase significantly even when the size of the network grows by nearly four orders of magnitude, with a loss rate under 0.0006% even with 102,400 nodes. Figure (b) shows the mean delay as a function of network size. Mean delay increase logarithmically with network size, resulting in the observed small increase in packet loss rate.

This experiment shows that our system is able to achieve the main goals of low packet loss rates as well as secondary goals of low delay and quick startup time, and low overhead. For the remainder of this chapter, we focus on packet loss rates as the primary performance metric, and provide other measurements as needed to fully explain the observed performance characteristics.

6.4 Scalability with Network Size

In this experiment, we demonstrate that Chainsaw scales well with network size. We started with the same basic setup as the network used in Section 6.3, but adjusted the node arrival rate to vary the expected network size in 2x increments. We ran a series of simulations with mean network sizes ranging from 25 to 102,400. Figure 6.4 shows the mean packet loss rates and mean delay as a function of network size. Note that the X-axis has a logarithmic scale.

In the smallest experiment, we have a graph that is almost fully connected, because the expected network size is 25 nodes, and nodes maintain 20 neighbors. Most nodes are directly connected to the seed, and all nodes receive some fraction of data very quickly via seed push. Therefore, the packet loss rate was nearly zero: in fact only a single node lost one packet over the course of that 30 minute experiment. With larger network sizes, the packet loss rates climbed slightly, but remained very low: between 0.0004% and 0.0006%, or about one in 200,000 packets.

The mean delay exhibits nearly perfect logarithmic growth, ranging from 0.72 sec to 3.23 sec. The logarithmic growth in delay is to be expected. Nodes maintain a constant degree; therefore the diameter of the graph grows logarithmically with the size of the network. The number of hops needed for packets to reach nodes farthest from the seed is equal to the diameter of the graph. Therefore, logarithmic growth in delay is precisely what would be expected.

Note that even though we use a five second timeout, that does not imply that the system will fail when the delay exceeds five seconds. Packet timeouts are determined in relation to other packets received by the node, whereas the mean delay plotted here is a measure of delay from the time a node generates the packet to the time a node receives it.

This experiment demonstrates that our system scales extremely well with network sizes. Although our main focus is to demonstrate that our network scales well to large network sizes, the fact that the system shows good performance at small network sizes is also significant. This implies that our system works well even with only a few users and does require adoption by a large number of users to provide good performance, thus simplifying the problem of gaining adoption by end users.

At 102,400 nodes, the aggregate bandwidth delivered to the system is 20.46 Gbps. Serving that bandwidth with a conventional client-server model would be impossible without a large dedicated datacenter or a content-distribution network like Akamai [91]—the total bandwidth demand would require a dedicated OC-768 [39], which is normally used by major ISPs backbones [4]. The peer-to-peer model, however, allows the seed to use a mere 960kbps of bandwidth which is easily available in consumer-grade connections, while requiring viewers to contribute no more bandwidth than is available from typical consumer broadband connections.

6.5 Scalability with Stream Rate

In this experiment, we demonstrate that Chainsaw scales well with increasing stream rates. Starting with the base setup described in Section 6.3, we increased the stream rate by keeping the packet size constant, but increasing the number of packets injected by the seed per second. We also proportionally increased every node's upload and download capacity. This experiment studies the ability of the system to route a large number of packets within the desired time constraints.

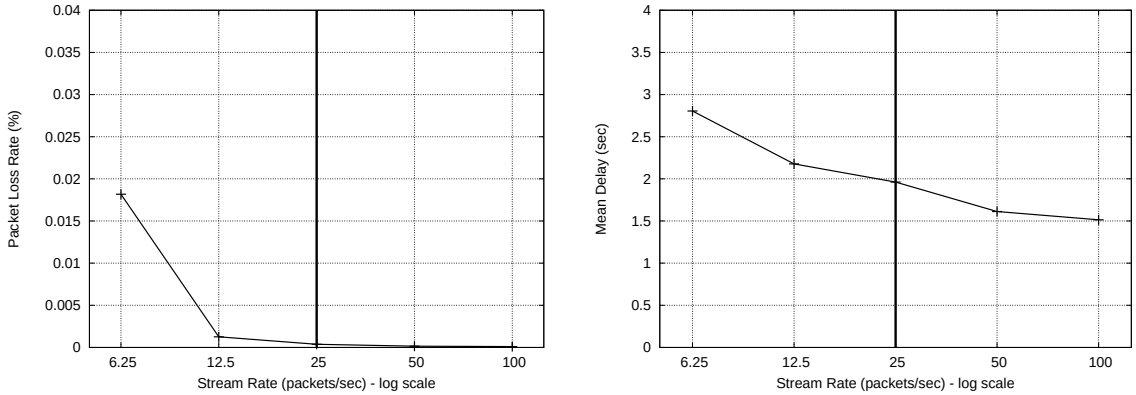
We study the complementary experiment where we change the number of packets per second without altering the overall bandwidth in Section 6.6.

Figure 6.5 shows packet loss rates and mean delay as a function of stream rate. Once again, the X axis has a logarithmic scale. We find that the mean delay and packet loss rates both go down as the data rate increases. This seems counter-intuitive at first, but is explained by the fact that nodes have increased bandwidth, but a constant packet size. As a result, individual packets propagate more quickly through the network, resulting in lower mean delay.

Packet loss in a resource-rich network are caused by random fluctuations causing packets to be delayed beyond the deadline (five seconds, in this case). The quicker propagation time gives nodes more opportunities to request each individual packet before the timeout, thus lowering the chance that an individual packet will be delayed beyond the deadline. Therefore, we observe lower mean packet loss as well.

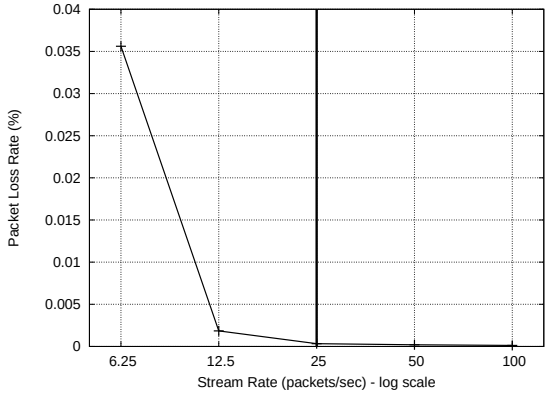
6.6 Effect of Packet Size

In this experiment, we study the effect of varying packet sizes while keeping overall bandwidth constant, by adjusting the number of packets per second accordingly. This is in contrast to the

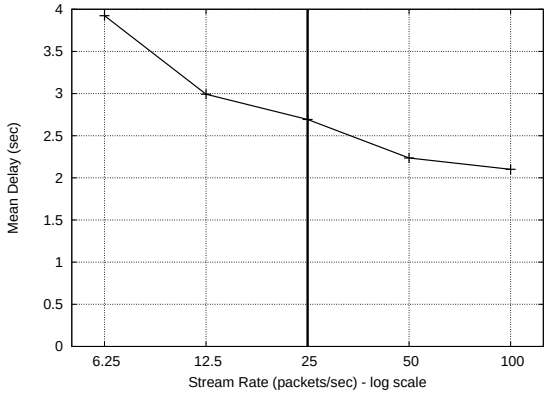


(a) Packet loss rate (800 nodes)

(b) Mean delay (800 nodes)



(c) Packet loss rate (8,000 nodes)



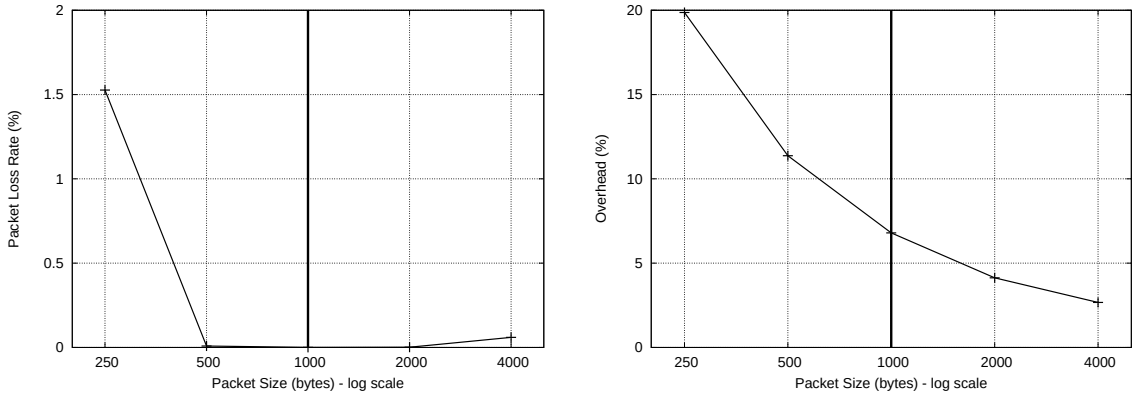
(d) Mean delay (8,000 nodes)

Figure 6.5: Performance of the system with varying stream rate. Figures (a) and (b) show the packet loss rate and mean delay for the 800 node network. Figures (c) and (d) show the corresponding graphs with an 8,000 node network. Higher raw bandwidth leads to lower propagation delay, which in turn lowers packet loss rate with increasing stream rate. The vertical line marks the base system data point, with a stream rate of 25 packets/sec.

previous experiment in Section 6.5, where we kept the packet size constant to adjust bandwidth. As we are keeping the overall stream rate constant, unlike Section 6.5, we do not alter the nodes' upload and download capacities in this experiment.

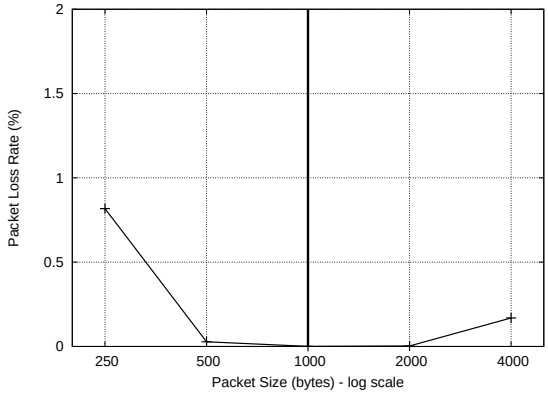
Figure 6.6 show the packet loss rates and overhead as a function of packet size. Much of the control traffic in the form of packet notifications, packet requests, etc. is incurred *per packet* regardless of the size of the payload. In order to maintain the stream rate (bytes/sec) we must increase the number of packets per second with smaller packets, and decrease the number of packets per second with larger packets. Therefore, the overhead (i.e., the ratio of overhead bytes to payload bytes) is very high for small packets and much lower for large packets.

Higher overhead increases the probability of temporary congestion where too much bandwidth is taken up by control traffic, leading to insufficient bandwidth available to service data requests. Therefore we observed higher packet loss rates with 250 byte packets, 1.5% with the 800 node network, and 0.8% with the 8,000 node network.

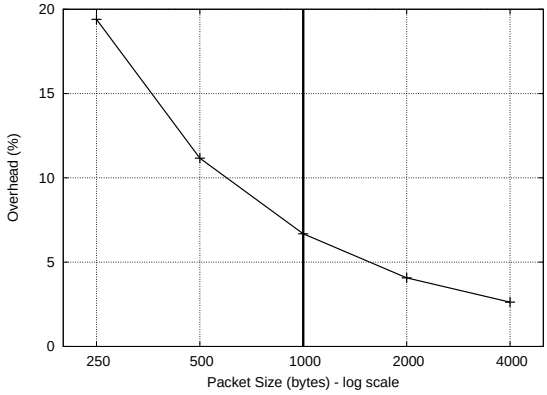


(a) Packet loss rate (800 nodes)

(b) Overhead (800 nodes)



(c) Packet loss rate (8,000 nodes)



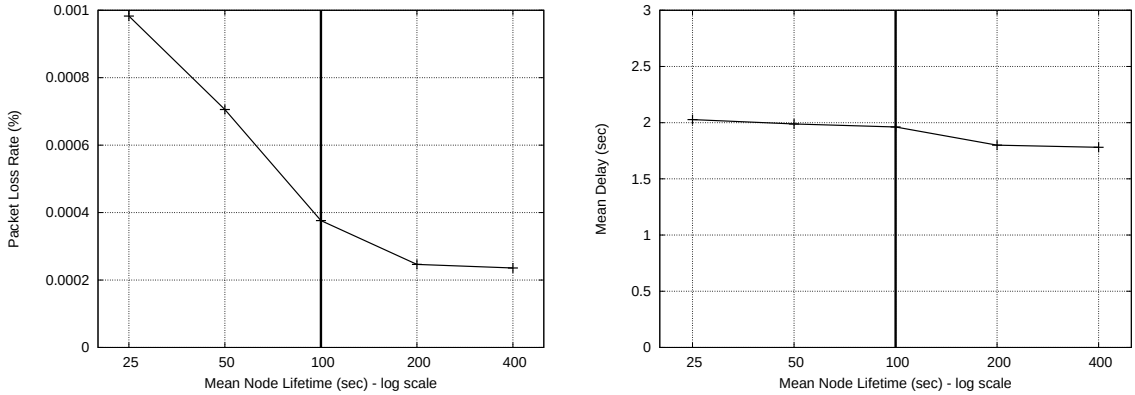
(d) Overhead (8,000 nodes)

Figure 6.6: Performance of the system with varying packet sizes, with a constant stream rate. Figures (a) and (b) show the packet loss rate and overhead for the 800 node network, and Figures (c) and (d) show the corresponding graphs for the 8,000 node network. The vertical line marks the base system point with a packet size of 1,000 bytes.

Larger packets have two effects. Firstly, larger packets take longer to transmit and receive, slowing down their propagation to the periphery of the network (i.e., nodes that are farthest from the seed). Furthermore, fewer packets per second leads to decreased parallelism since fewer neighbors are uploading packets at any time. This reduces the effective degree of the network, and increases the impact of any individual neighbor leaving the network or suffering temporary congestion. Therefore, we observed a slight increase in packet loss rate with larger packet sizes.

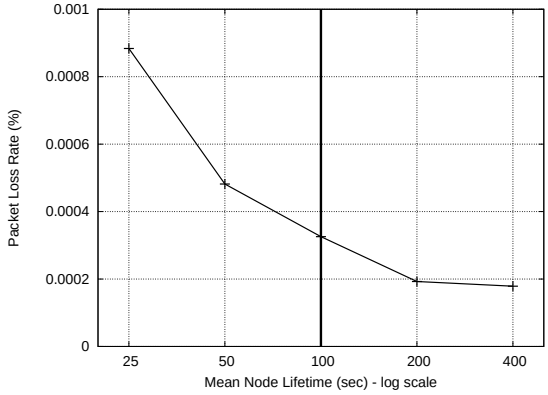
6.7 Robustness to Churn

In this section, we demonstrate the robustness of our system to churn, the rate at which nodes enter and leave the system. A high rate of churn may be problematic for networks that need to propagate routing information or form propagation trees.

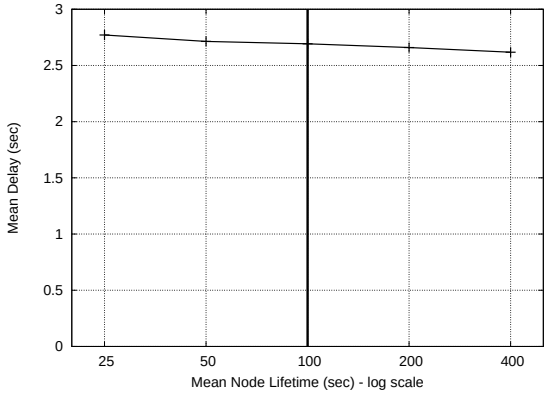


(a) Packet loss rate (800 nodes)

(b) Mean delay (800 nodes)



(c) Packet loss rate (8,000 nodes)



(d) Mean delay (8,000 nodes)

Figure 6.7: Performance of the system with varying levels of churn. We varied mean node lifetimes, and adjusted the mean inter-arrival to maintain the mean network size of 800 and 8,000, respectively, and Figures (a) and (b) show the packet loss rate and mean delay for the 800 node network, and Figures (c) and (d) show the corresponding graphs for the 8,000 node network. Packet loss rates increase only modestly even with very short average node lifetimes. The vertical line marks the base system point with a mean node lifetime of 100 sec.

Nodes in the base system of Section 6.3 had a mean lifetime of 100 seconds. We simulated networks with longer lifetimes, as networks with lifetimes as low as 25 seconds. This represents an extreme situation with highly ephemeral participants. In order to keep the network size constant, we adjusted the mean inter-arrival rate to compensate for varying lifetimes.

In our protocol implementation, in an effort to be conservative, we simulated the worst case behavior where nodes abruptly leave the system without notifying their neighbors of their intent, or disconnecting gracefully. A node might have outstanding requests from its neighbor, but will leave them unfulfilled as it leaves the network, requiring the neighbor to find a new source for that packet. In a more well-behaved implementation, nodes might linger for a few seconds and service outstanding packet requests while refusing to accept new requests, which will lead to even better performance than our worst-case implementation.

Figure 6.7 shows packet loss rates and mean delay as a function of mean node lifetime.

Lower lifetimes indicate increasing levels of churn. Sub-figures (a) and (b) show the packet loss rate and mean delay, respectively, as a function of the mean node lifetime for the 800 node network. Sub-figures (c) and (d) show the corresponding graphs for the 8,000 node network. Note that the X-axis has a logarithmic scale.

We find that while an extreme level of churn does adversely affect the system, the packet loss rate remains very low, under 0.001% in both cases. The increase in packet loss rate is primarily caused by nodes suddenly losing neighbors. With increasing churn, it becomes increasingly common for a node to request a packet from a neighbor only to have that neighbor leave the system before fulfilling the request. On rare occasions, this will leave the node with no more sources of that packet, leaving the node unable to obtain that packet.

When a node's neighbor leaves the system without fulfilling an outstanding request, the node must then request that packet from another neighbor after it either times out, or detects that it has been disconnected from the neighbor. Issuing a new request and waiting for the new neighbor to respond leads to additional round-trip delays, leading to a small increase in delay with shorter node lifetimes. However, even with an extremely short mean lifetime of 25 seconds, the delay does not increase significantly. With the 800 node network, the mean delay increased from 1.96 sec for the base system with a mean node lifetime of 100 sec to 2.03 sec with a mean node lifetime of 25 sec. With the 8,000 node network, the mean delay increased from 2.69 sec for the base system to 2.77 sec with a mean node lifetime of 25 sec.

This experiment shows that Chainsaw is highly robust to churn, even in the extreme situation where nodes linger only for very short periods of time, and a worst-case implementation where nodes abruptly leave the network, leaving their neighbors' requests unfulfilled. A practical system would have even better performance than demonstrated in this worst-case experiment for two reasons. Firstly, in a real implementation, nodes would likely leave the network more gracefully. Secondly, extremely short mean node lifetimes are unlikely to arise in practice. We consider even the default value of 100 seconds to be quite conservative because we expect viewers to linger much longer on average for common applications like video streaming.

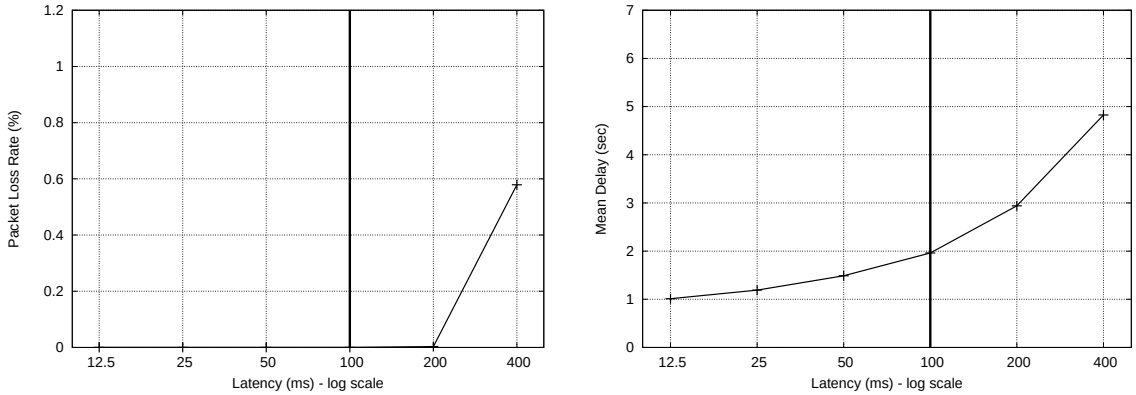
6.8 Effect of Network Latency

In this experiment, we evaluate the effect of network latency, the time taken by packets the traverse the network from one node to another. Note that we refer to *one way* latency, rather than the round-trip delay reported by common network diagnostic tools like `ping`.

As mentioned in Section 6.3, we use an extremely pessimistic value of 100ms one-way latency for the base system. In this experiment, we varied the network latency in logarithmic steps from 12.5ms (comparable to typical Internet transit type to a host in a nearby city), to 400ms, which is quite pessimistic even for satellite links, or long intercontinental terrestrial links—the propagation delay for radio waves to a satellite in geostationary orbit is 120ms.

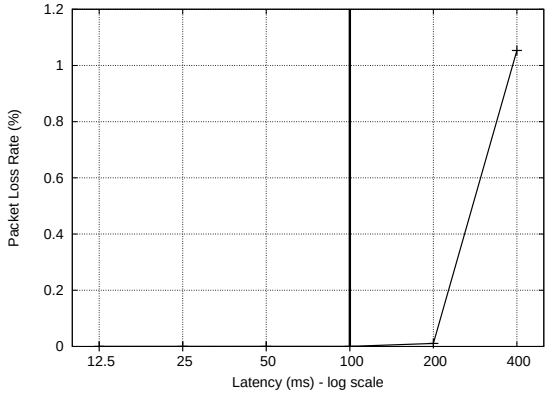
Figure 6.8 shows packet loss rates and mean delay as a function of network latency. The X axis has a logarithmic scale for both graphs, while the Y axis has a linear scale for the delay graph and logarithmic for the packet loss graph.

We observe that increasing network latency leads to both an increase in packet delay, which leads to a higher packet loss rate. However, even with a latency of 400ms, we observed packet loss rates under 1% in both the 800 and 8,000 node networks.

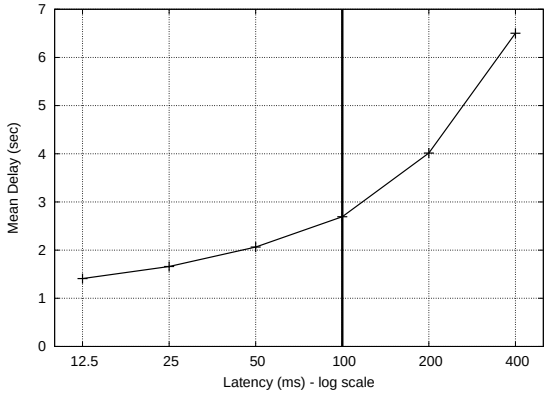


(a) Packet loss rate (800 nodes)

(b) Mean Delay (800 nodes)



(c) Packet loss rate (8,000 nodes)

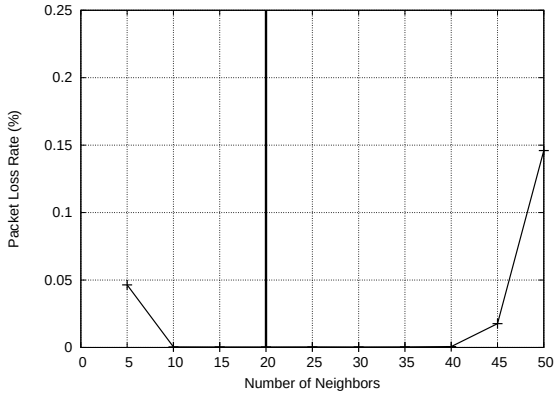


(d) Mean Delay (8,000 nodes)

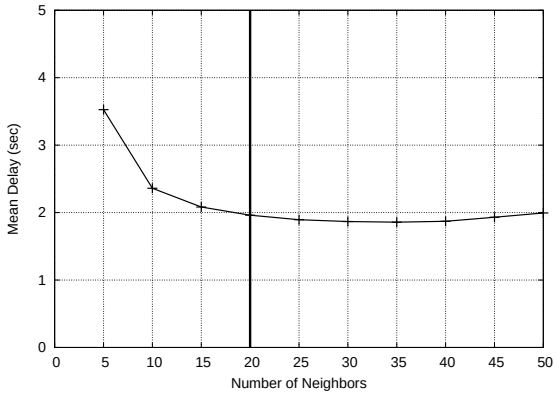
Figure 6.8: Performance of the system with varying inter-node network latency. Figures (a) and (b) show the packet loss rate and mean delay for the 800 node network, while Figures (c) and (d) show the corresponding graphs for the 8,000 node network. As expected, higher network latency leads to increased propagation delay, which leads to higher packet loss. Packet loss rates remain under 1% even with a very high 400ms latency in each direction. The vertical line marks the base system point with a latency of 100ms.

Note that the increase in packet delay is *sub-linear*—while we increased the network latency by a factor of 32, from 12.5ms to 400ms, the mean delay increased from 1 sec to 5 sec. This result might seem counter-intuitive and one might expect a linear increase. However, some delays are mitigated due to pipelining effects—nodes will often transmit a request for the next packet from a given neighbor while it is in the process of downloading another data packet from that neighbor, thus avoiding the latency penalty for the second packet request.

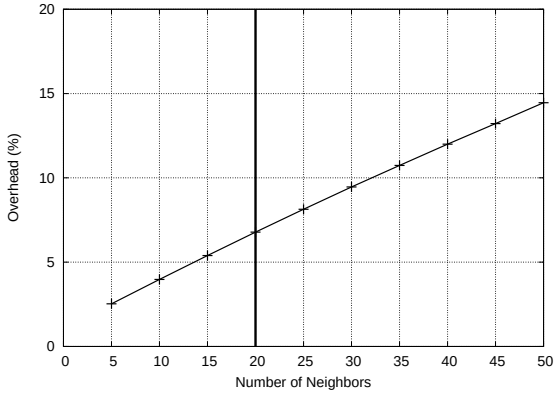
Increasing delay with a constant cut-off time does lead to an increase in packet loss as it is not possible to sustain a low lag behind the seed over a multi-hop network with high network delay. For instance, if temporary congestion or the departure of a neighbor prevents a node from receiving the packet from the neighbor it originally requested it from, it will take several round trips for the node to detect that fact, request the packet from another neighbor, and receive the packet. With a high network latency, and a relatively low lag timeout of 5 sec, there is little



(a) Packet loss rate (800 nodes)



(b) Delay (800 nodes)



(c) Overhead (800 nodes)

Figure 6.9: Performance of the system with varying number of neighbors, i.e., graph degree with the 800 node network. At low graph degrees, the high diameter of the graph drives packet loss rates higher. At higher degrees, increasing overhead may cause higher packet loss rate. Note that packet loss rates remain relatively low, under 0.2% at both extremes. The vertical line marks the base system point with 20 neighbors.

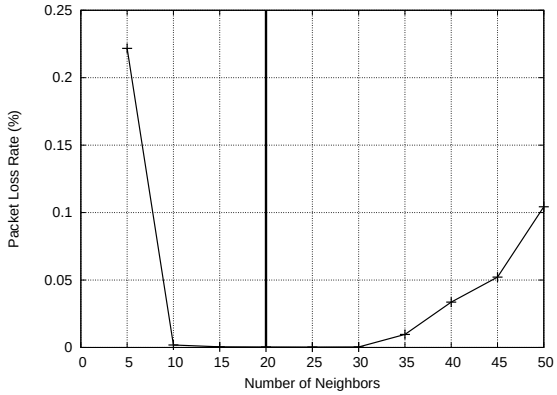
margin for rerouting or other delays, leading to the observed increase in packet loss.

This experiment demonstrates that the network is able to operate quite effectively, with packet loss rates under 1%, even under challenging conditions with a one-way network latency of 400ms, which is a highly pessimistic value even for networks distributed globally. In a real world implementation, we would expect to see significantly lower delays.

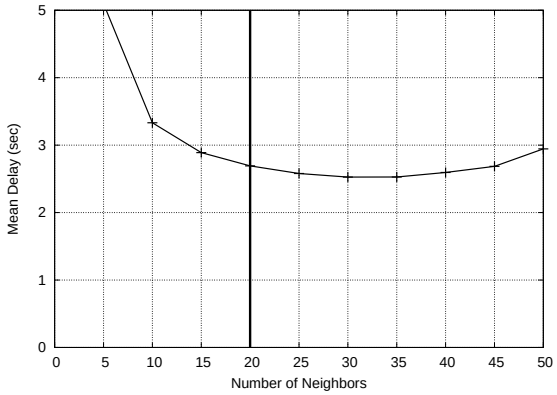
6.9 Effect of the Number of Neighbors

In the base system, we have used a network degree of 20. That is, every node in the system maintained connections to 20 neighbors. In this experiment, we study the effect of this parameter. We varied the network degree from 5 to 50 in increments of five.

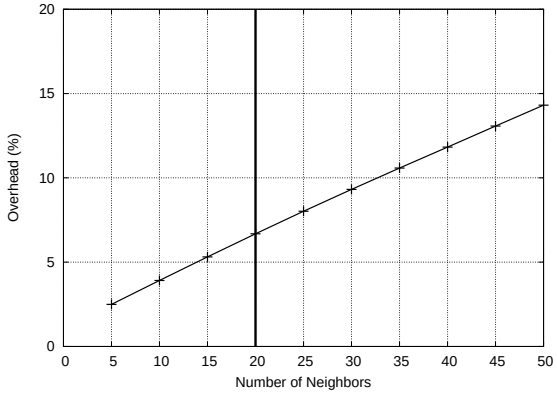
Figure 6.9 shows packet loss rates, mean delay, and overhead as a function of network degree



(a) Packet loss rate (8,000 nodes)



(b) Delay (8,000 nodes)



(c) Overhead (8,000 nodes)

Figure 6.10: Performance of the system with varying number of neighbors, i.e., graph degree with the 8,000 node network. At low graph degrees, the high diameter of the graph drives packet loss rates higher. At higher degrees, increasing overhead may cause higher packet loss rate. Note that packet loss rates remain relatively low, under 0.25% at both extremes. The vertical line marks the base system point with 20 neighbors.

and Figure 6.10 shows the corresponding experiment with 8,000 nodes. In both the 800 and 8,000 node network we observed that packet loss rates are higher with very low graph degree, and then rapidly fall very close to zero. As we continue to increase graph degree, the packet loss rate starts to climb gradually.

This behavior is explained by the opposing effects observed in the delay and overhead graphs in sub-figures (b) and (c), respectively. When the graph degree is very low, i.e., nodes connect to a very small number of nodes, the network has a very high diameter. For example, with an 800 node network with degree 5, a majority of nodes will be 4 hops away from the seed. Therefore packets will take longer to reach nodes on the periphery, leading to longer delays and higher packet loss. With the 8,000 node network, a majority of nodes will be 6 hops away, so this effect is even more pronounced. We observed packet loss rates of 0.04% with the 800 node network, and 0.23% with the 8,000 node network.

Higher network degree (i.e., more neighbors) brings diminishing returns, but increasing cost.

Nodes must broadcast packet availability notifications to every neighbor they are connected to, and will receive notifications from every neighbor. This leads to a linear increase in the amount of control traffic (i.e., overhead). The increase in overhead also contributes to a small increase in delay as packet requests and transmission get queued behind notification messages. Therefore, as we increased the network degree to 50, we observed that packet loss rates increased gradually to 0.15% with the 800 node network, and 0.10% with the 8,000 node network.

Between the two extremes, we observed a wide range of network degrees where the packet loss rates were extremely low, under 0.0001%. This wide range shows that the network is stable and does not need to be tuned with great precision to achieve low packet loss rates. Furthermore, while packet loss rate at the two extremes was higher than the packet loss rate in the sweet spot towards the center of the graph, it remained quite low on an absolute scale, under 0.25% in all cases. Moreover, a practical implementation would be able to measure the amount of overhead traffic they are incurring and reduce the number of neighbors they connected to in order to stay out of the extremes with higher packet loss rates.

6.10 Token Stealing in a Resource-Rich System

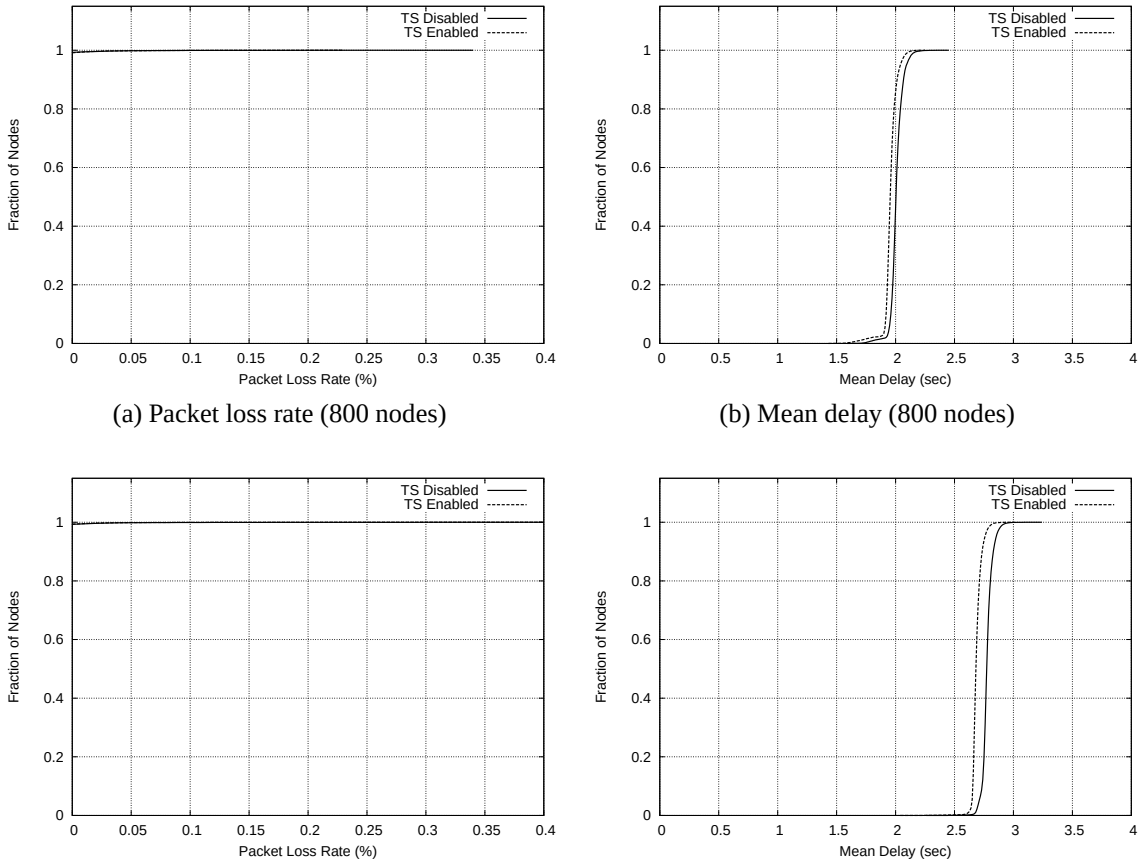
In this section, we compare the performance of the system with the Token Stealing algorithm disabled, to the base system described in Section 6.3. The goal of this experiment is to demonstrate that the Token Stealing algorithm does not impose a performance penalty on a resource rich system, i.e., when all nodes in the system are altruistic and contribute at least upload bandwidth as the stream rate.

Figure 6.11 shows the cumulative distribution of packet loss rates and mean delay with the Token Stealing algorithm disabled, and enabled for both the 800 and 8,000 node cases. As with our previous experiments, we observe that the packet loss rates are very low, with a system-wide mean of only 0.0004% in the 800 node experiment both with the Token Stealing algorithm disabled and enabled. Enabling the Token Stealing algorithm actually resulted in a slight improvement in the mean delay from to 2.01 with Token Stealing disabled to 1.96 with the algorithm enabled.

We observed very similar results with the larger, 8,000 node network. Once again, the packet loss rate was very low, with a system-wide average of 0.0004% without Token Stealing and 0.0003% with Token Stealing enabled. As before, enabling Token Stealing resulted in a slight improvement in mean delay with the mean value dropping from 2.80 with Token Stealing disabled to 2.69 sec with the algorithm enabled.

At first glance, these results might seem counter-intuitive. One might expect the imposition of additional constraints to reduce the performance of the system. However, the Token Stealing algorithm does not restrict uploads the way a tit-for-tat system would, but merely redistributes it among various neighbors. In a tit-for-tat system, a node may leave a portion of its upload capacity unutilized if it does not receive a comparable amount of download bandwidth. However, Token Stealing merely adjusts the priority of neighbors in response to their download bandwidth, and will not leave upload capacity unutilized if there is adequate demand. In fact, it encourages packets to take more efficient paths by giving preferential treatment to nodes that upload more data.

Thus we show that the Token Stealing algorithm not only does not degrade performance



(a) Packet loss rate (800 nodes)

(b) Mean delay (800 nodes)

(c) Packet loss rate (8,000 nodes)

(d) Mean delay (8,000 nodes)

Figure 6.11: Performance of a resource-rich system with and without Token Stealing enabled. In this experiment all nodes are altruistic. Figures (a) and (b) show the cumulative distributions of packet loss rates and mean delay for the 800 node network, and Figures (c) and (d) show the corresponding graphs for the 8,000 node network. The performance of the network in the resource-rich case very similar with Token Stealing enabled as without.

by imposing additional constraints, but may lead to slightly better performance through more efficient routing of packets.

6.11 Resource-Constrained Systems

In this section, we move from resource-rich systems where all nodes are altruistic (i.e., are willing to contribute more upload bandwidth to the system than the stream rate) to resource-constrained systems where some nodes are unable or unwilling to contribute as much upload bandwidth as the stream rate.

In this series of experiments, we introduce nodes to the system with limited upload capacity. For brevity, we refer to these as ADSL nodes, because we model their behavior after consumer Asymmetric DSL connections which typically offer a significantly lower upload rates

Node Type	Capacity as Fraction of Stream Rate		
	Upload Rate	**Raw Upstream**	**Raw Downstream**
Altruistic	1.6	2	2
ADSL	0.32	0.4	2
Fake ASDL	0.32	2	2

Table 6.2: Upload and download capacities of Altrustic, ADSL and Fake ADSL nodes. Altruistic nodes have a raw upload and download capacity equal to twice the stream rate, and a maximum upload rate (in terms of data packets) equal to 80% of the raw upload rate, i.e., 1.6 times the stream rate. ADSL nodes have the same download capacity, but an upload capacity only 1/5 that of the altruistic nodes. Fake ADSL nodes have the same upload and download capacity as altruistic nodes, but artificially limit their upload rate to that of the ADSL nodes.

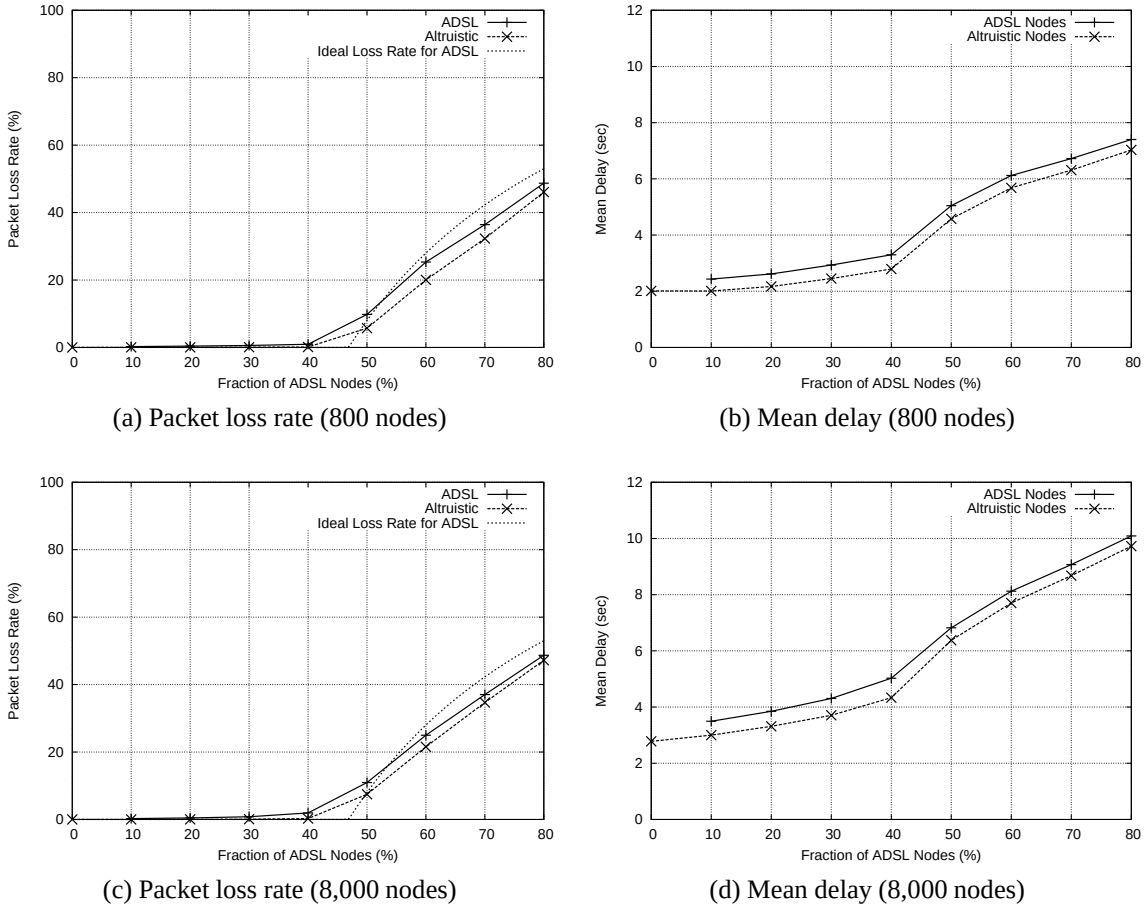
than download rates. It is common for such connections to have a ratio of about 1:5 (for instance, 2 Mbps up and 10 Mbps down), so we used the same ratio, and set the upload rate of ADSL nodes to 20% that of the altruistic nodes in this series of experiments. As before, the altruistic nodes have an upload capacity 1.6 times the stream rate. Therefore, the ADSL nodes had an upload rate 0.32 times the stream rate. Recall that the upload rate refers to the amount of data packets a node is willing to upload, not the raw line bandwidth. We also introduce a second class of nodes called Fake ADSL nodes that have the same raw upload capacity as Altruistic nodes but artificially limit their upload capacity to resemble the ADSL nodes. Table 6.2 summarizes the parameters used by the three classes of nodes.

In this section, we first demonstrate the need for an incentive mechanism by disabling our incentive mechanism, the Token Stealing algorithm. We then ran an identical series of experiments with the Token Stealing algorithm enabled, demonstrating that despite its simplicity, our incentive mechanism fulfills our goals of providing a strong incentive for nodes to contribute upload bandwidth to a system, when the system is resource-constrained, while accommodating all nodes while resource-rich.

6.11.1 The Need for an Incentive Mechanism

We ran a series of simulations with different fractions of ADSL nodes, starting with all altruistic nodes, and increasing the fraction to 80% ADSL nodes, in 10% increments. We achieved the desired ratio of the two types of nodes by introducing ADSL and altruistic nodes via separate Poisson processes, with the mean inter-arrival rates adjusted to produce the desired ratio. As before, we ran one series of experiments with the expected total size of the network maintained at 800, and another with the mean total size maintained at 8,000.

Every node regardless of its upload capacity attempts to download the complete stream. Therefore, bandwidth demanded from the system by every node is equal to the stream rate. On the other hand, the supply of bandwidth contributed to the system varies between the two classes of nodes. If we consider the stream rate to be one unit, the upload bandwidth, or supply contributed by nodes is 0.32 for ADSL nodes, and 1.6 for ADSL nodes. The overall ratio of bandwidth supply to demand in the system is therefore:



(a) Packet loss rate (800 nodes)

(b) Mean delay (800 nodes)

(c) Packet loss rate (8,000 nodes)

(d) Mean delay (8,000 nodes)

Figure 6.12: Performance with varying fraction of ADSL nodes with the Token Stealing algorithm disabled. As the system becomes resource-constrained, both ADSL and altruistic nodes suffer increasing packet loss. With the Token Stealing algorithm disabled, there is very little difference between the performance of the two classes of nodes.

$$\frac{supply}{demand} = 0.32x + 1.6(1 - x) \tag{6.1}$$

where x is the fraction of ADSL nodes in the system. The supply becomes equal to demand at $x = 0.47$. Therefore, when the system has fewer than 47% of ADSL nodes, the system is resource-rich, and the systems becomes progressively resource-constrained beyond that.

Figure 6.12 shows the packet loss rates and mean delay of experienced by ADSL and altruistic nodes, as a function of the fraction of ADSL nodes in the system. As before, we repeated the experiment for both 800 and 8,000 node networks.

With 0% ADSL nodes, the setup is, of course, identical to that in Section 6.3, and the packet loss rates are very close to zero. Moreover, all nodes in the system see very low packet loss rates with 10–40% ADSL nodes in the system. The altruistic nodes had under 0.25% for packet loss rates both the 800 and 8,000 node networks, while the ADSL nodes about 0.9% with the 800 node network, and 1.9% with the 8,000 node network. Recall that the system is resource-rich,

so long as there are under 47% ADSL nodes in the system, the point marked by the solid vertical line in the graphs.

In the region of the graph with 50–80% ADSL nodes, the packet loss rates for both classes of nodes increased rapidly, but there was very little difference between the packet loss rates of the two classes of nodes, just 1–3%. This small difference is unlikely to provide a strong incentive for nodes to contribute upload bandwidth to the system, thus demonstrating the need for an incentive mechanism.

It might seem surprising that there is any difference at all between the ADSL and altruistic nodes at all with no incentive mechanism. However, recall that in this experiment the ADSL nodes had a lower line speed compared to the altruistic nodes, resulting in slower propagation of data and control traffic, increasing delays. We observe in Figure 6.12 that the ADSL nodes on average had delays about half a second longer in the 800 node network, and nearly a second more in the 8,000 node network. This longer delay increased their tendency towards packet loss slightly, explaining the slight difference in packet loss rates.

To further illustrate this point, we ran another series of experiments with a modification to the parameters of the ADSL nodes, that we refer to as "Fake ADSL" nodes. While these nodes still set their upload rates (i.e., maximum data upload rate) to 0.32 of the stream rate—same as the ADSL nodes, they have a line speed equal to the altruistic nodes, and would thus not be encumbered by slower propagation times for their data and control packets. This class represents selfish nodes who have the physical capacity to upload data at the stream rate or higher, but unlike the altruistic nodes, they choose to limit their upload rates.

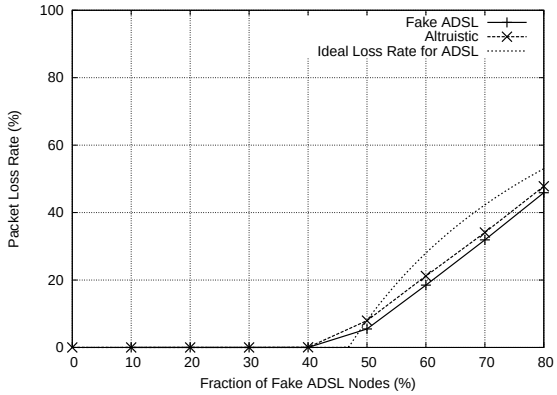
Figure 6.13 shows the results of this experiment. As with the experiment with real ADSL nodes, we plot the packet loss rates, and mean delay of both classes of nodes as a fraction of the percentage of non-altruistic nodes (Fake ADSL, in this case). Once again, we repeated the series of experiments with both 800 and 8,000 node networks.

As before, we observe that the packet loss rates are very low for all nodes in the 0–40% range, while the system as a whole is resource-rich. In fact, in this range, the packet loss rates are even lower than in the previous experiment—under 0.1% for both classes of nodes in both the 800 and 8,000 node experiments. This is because the packets that the Fake ADSL nodes do upload propagate to their neighbors much faster due to the higher line speed compared to the real ADSL nodes.

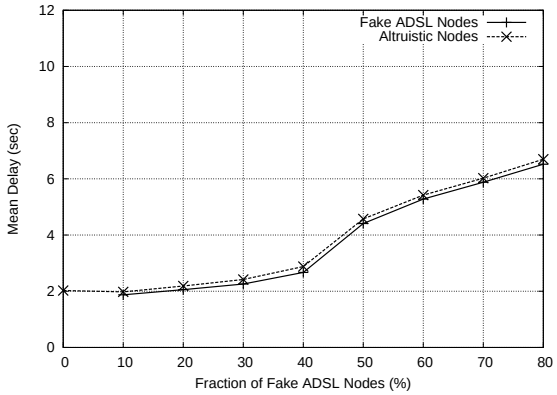
In the resource constrained range, however, we observe that the Fake ADSL nodes actually performed slightly better than the altruistic nodes. Although the difference in packet loss rates between the two classes of nodes is small, under 1% with both the smaller and larger network, this is an highly undesirable outcome. In this scenario, selfish nodes saw *better* performance than the altruistic nodes.

As before, this difference is explained by the delay graphs in sub-figures (b) and (d) for the 800 and 8,000 node networks, respectively. Whereas the Fake ADSL nodes have the same line speed as the altruistic nodes, they spend much less of their bandwidth on uploading data packets, and thus are able to transmit packet requests and other control traffic more quickly, resulting in lower delay. As before, this lower delay translates to lower packet loss rates. Once again, we observe similar results with both the 800 and 8,000 node networks.

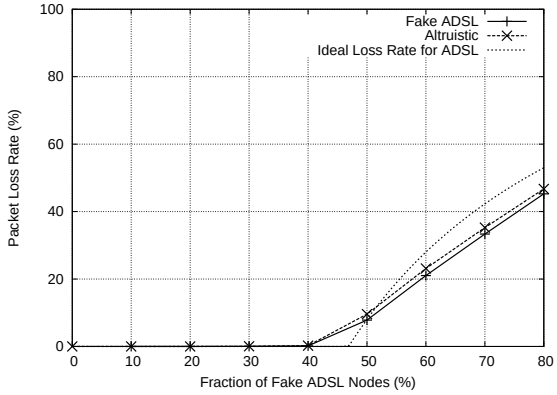
Although the difference in packet loss rates between the altruistic nodes and Fake ADSL nodes is small, if at all there was an incentive created by the difference in packet loss rates, it



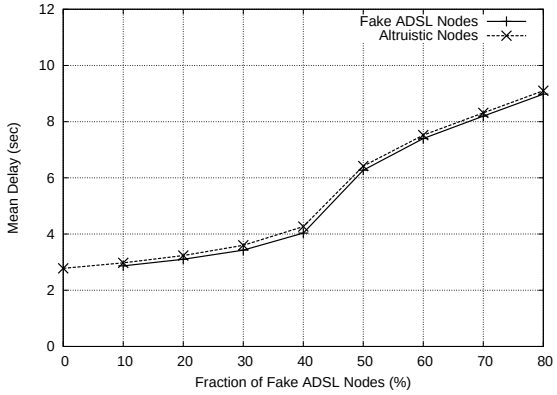
(a) Packet loss rate (800 nodes)



(b) Mean delay (800 nodes)



(c) Packet loss rate (8,000 nodes)



(d) Mean delay (8,000 nodes)

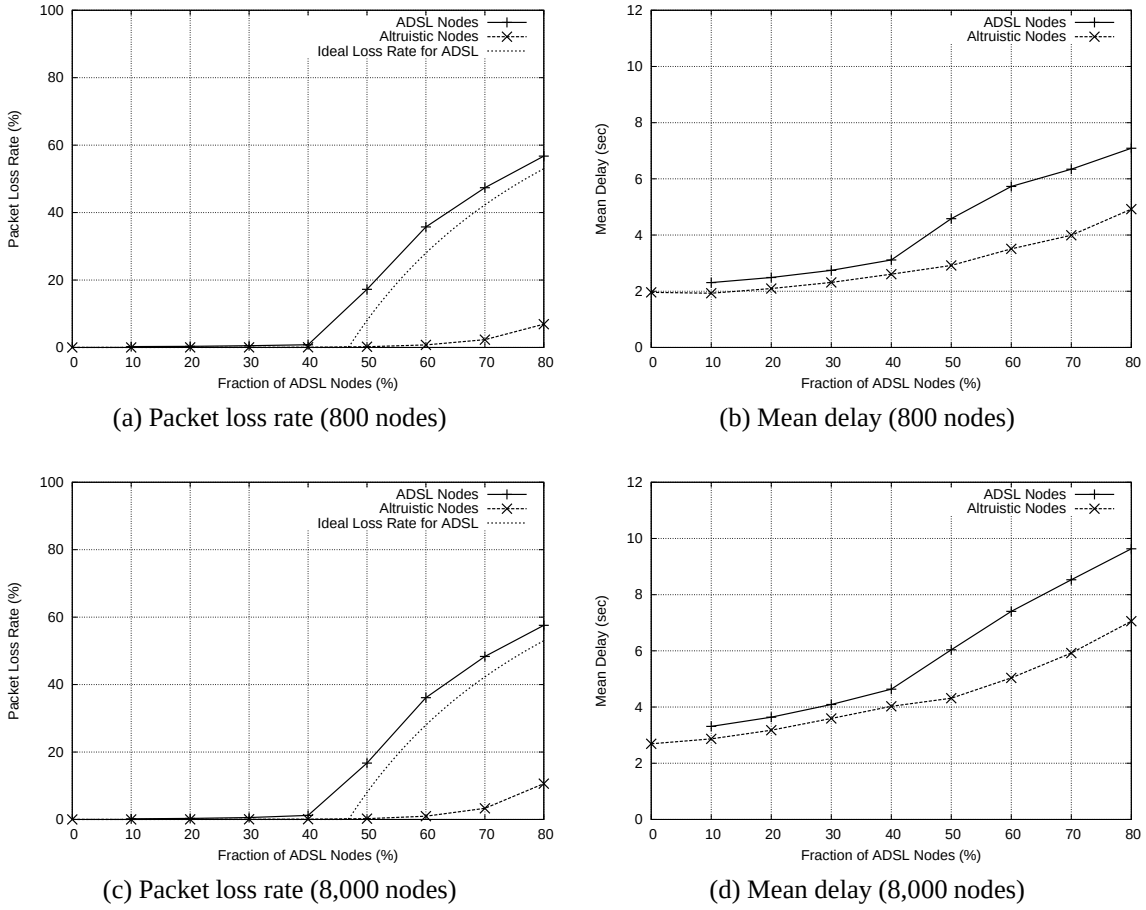
Figure 6.13: Performance of the system with varying fraction of Fake ADSL nodes with the Token Stealing algorithm disabled. Fake ADSL nodes have the same physical capacity as the altruistic nodes, but are selfish restrict and their upload rates to masquerade as ADSL nodes. With their high bandwidth connection relatively unencumbered by uploading data to their neighbors, the Fake ADSL nodes actually see slightly lower packet loss rates and mean delays than the altruistic nodes—a highly undesirable outcome.

would be to be selfish rather than altruistic. Clearly, this outcome is highly undesirable from the overall system's point of view, thus demonstrating the need for an explicit and effective incentive mechanism.

6.11.2 Resource-Constrained Systems with Token Stealing Enabled

In Section 6.11.1, we demonstrated the need for an incentive mechanism. In this section, we repeat the same series of experiments with the Token Stealing algorithm re-enabled in order to show that our incentive mechanism does, in fact, provide nodes with a strong incentive to contribute upload bandwidth to the network.

As before, we first ran a series of experiments with a varying fraction of ADSL nodes whose upstream line speed limits their ability to contribute upload bandwidth to the system (i.e., *real*



(a) Packet loss rate (800 nodes)

(b) Mean delay (800 nodes)

(c) Packet loss rate (8,000 nodes)

(d) Mean delay (8,000 nodes)

Figure 6.14: Performance with varying fraction of ADSL nodes with the Token Stealing algorithm enabled. While the system is resource-rich, both ADSL and altruistic nodes suffer low packet loss rates. When the system becomes resource-constrained, though, altruistic nodes suffer much lower packet loss rates than the ADSL nodes.

ADSL nodes).

Figure 6.14 shows the packet loss rates and mean delay experienced by ADSL and altruistic nodes, as a function of the fraction of ADSL nodes in the system. As before, we repeated the experiment for both 800 and 8,000 node networks.

It is useful to compare Figure 6.14 to Figure 6.12, the results of our previous experiment with the Token Stealing algorithm disabled. As before, in the resource-rich 0–40% range, all nodes experienced low packet loss rates. In fact, the altruistic nodes experienced even better service than before, with packet loss rates under 0.1%. While slightly degraded compared the altruistic nodes, the ADSL nodes also received better service in the resource-rich network with the Token Stealing algorithm. The ADSL nodes had a 0.8% packet loss in the 800 node network, and 1.2% in the 8,000 node network, compared to 0.9% and 1.9%, respectively, with the Token Stealing algorithm disabled. This reaffirms our findings in Section 6.10, where our Token Stealing algorithm leads to a small improvement in overall network performance in the resource-rich network by routing around bottlenecks, preferentially sending packets to neighbors who are more likely

to upload those packets to other nodes.

The key difference though, is in the 50–80% range where the system becomes progressively more resource-constrained. With the Token Stealing algorithm enabled, there is a significant difference between the performance of ADSL and altruistic nodes when the system is resource-constrained. The packet loss rates of the altruistic nodes remains low, even as we increase the fraction of ADSL nodes, and the system becomes increasingly resource-constrained. At the 80% mark, there is only enough upload capacity in the system to fulfill 57.6% of the demand, but the altruistic nodes only suffer a packet loss rate of 6.8% in the 800 node network, and 10.6% in the 8,000 node network.

In contrast, the ADSL nodes experience a rapid increase in packet loss rates in the resource-constrained cases. At the 80% mark, they suffer packet loss rates of 57.4% in the 800 node network, and 57.6% in the 8,000 node network. Thus, the ADSL nodes see six to eight times the packet loss rates that the altruistic nodes see in the severely resource-constrained case. In fact, even at the point where the system is only slightly resource-constrained, at the 50% ADSL node mark, with 96% capacity, the ADSL nodes experience a visible increase packet loss rates—over higher 10%.

This shows that our incentive mechanism is accommodating and offers good service to all nodes regardless of their upload rates so long as the system is resource-rich due to altruistic nodes who make up the deficit created by ADSL nodes. However, when the number of ADSL nodes increases to the point where the system is resource-constrained, the altruistic nodes experience significantly better service (i.e., lower packet loss rates).

Next ran a series of experiments analogous to the previous experiment, with ADSL nodes replaced by Fake ADSL nodes. Recall that the Fake ADSL nodes are nodes with the same physical connections as the altruistic nodes that choose to be selfish by artificially limiting their upload rates to that of the ADSL nodes.

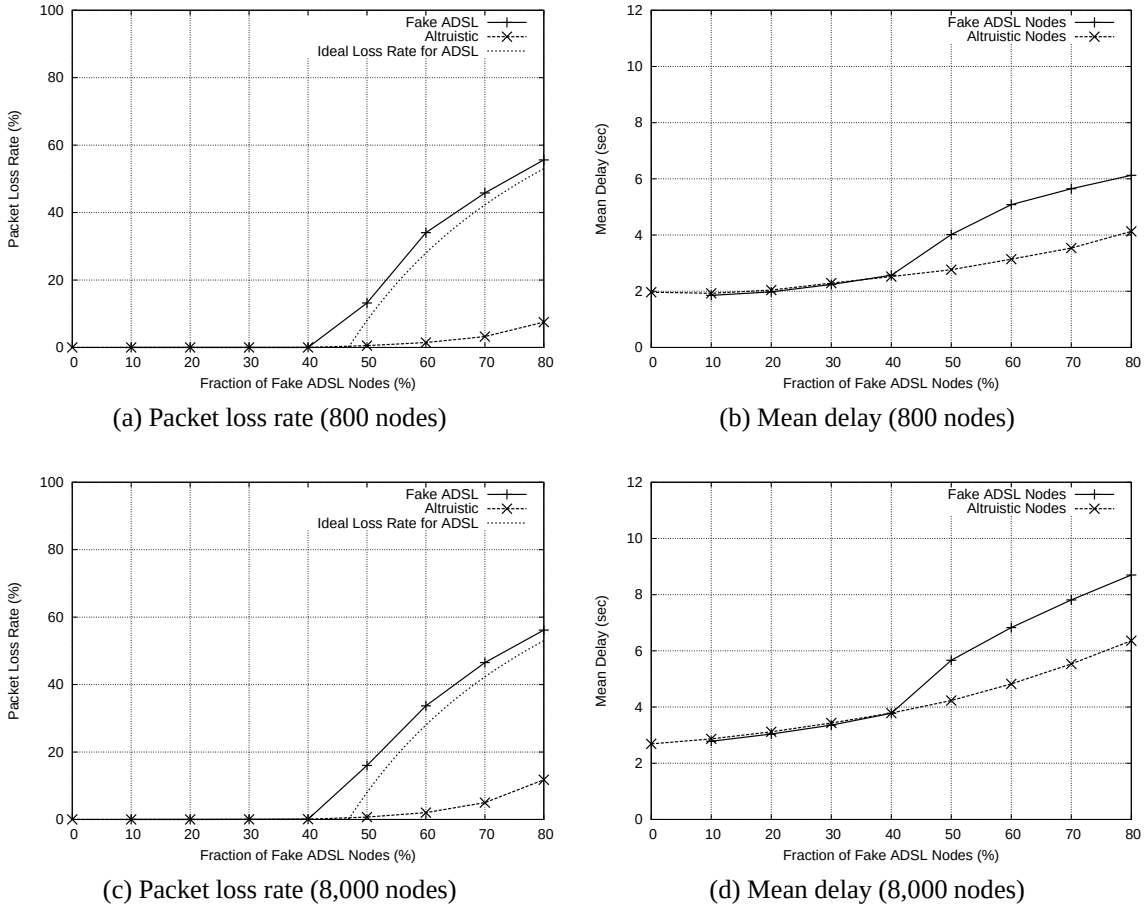
Figure 6.15 shows the result of this experiment. Figures (a) and (b) show the packet loss rates and mean delay experienced by the altruistic nodes, and Fake ADSL nodes for the 800 node network, while Figures (c) and (d) show the corresponding graphs for the 8,000 node network.

As before, in the 0–40% range, while the system is resource-rich, all nodes duffer very low packet loss rates, well under 0.1% for both classes of nodes in both the 800 and 8,000 node networks.

However, in the resource-constrained region with 50–80% Fake ADSL nodes, there is a significant difference in the packet loss rates seen by the two classes of nodes. As with the experiment with the real ADSL nodes, the altruistic nodes suffer only modest packet loss rates around 10% even with a large fraction of Fake ADSL nodes in the system.

In stark contrast to the results in Figure 6.13 though, the fake ADSL nodes have significantly higher packet loss rates than the altruistic nodes. With 80% Fake ADSL nodes in the system, they have a packet loss rate of 55.6% in the 800 node network and 56.2% in the 8,000 network, virtually identical to the packet loss rates seen by the real ADSL nodes in the experiment.

Clearly, this creates a highly undesirable situation for both the real and fake ADSL nodes in resource constrained systems. Real ADSL nodes might not have much of a alternative if their physical connections do not allow them to contribute more bandwidth to the system, and they might leave the system while it is resource-constrained, thus improving the overall ratio of



(a) Packet loss rate (800 nodes)

(b) Mean delay (800 nodes)

(c) Packet loss rate (8,000 nodes)

(d) Mean delay (8,000 nodes)

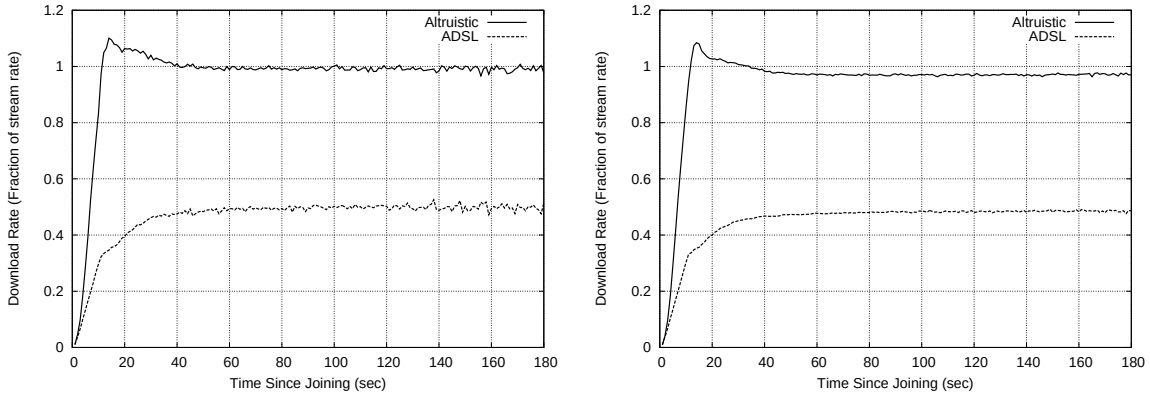
Figure 6.15: Performance with varying fraction of Fake ADSL nodes with the Token Stealing algorithm enabled. Fake ADSL nodes have the same physical capacity as the altruistic nodes, but selfishly restrict their upload rates to masquerade as ADSL nodes. Once the system becomes resource-constrained, the Fake ADSL nodes suffer much higher packet loss rates than the altruistic nodes and would be incentivized to remove the artificial cap on their upload rate.

supply to demand. Fake ADSL nodes do have a choice, and may be encouraged to remove the artificial limits on their upload capacity in order to reduce their packet loss rate, and improve the quality of service they receive. In either case, the self-interested response of nodes improves the overall ratio of supply to demand in the system, thus benefiting the system as a whole.

6.11.3 Steady-State Behavior

Although the difference in packet loss rates for the two classes of nodes is significant, and the 6–8% packet loss rates suffered by altruistic nodes is quite small and easily corrected by erasure coding and other techniques, the lifetime average packet loss suffered by nodes does not tell the whole story.

Recall that in order to maintain more realistic conditions, all our experiments were performed with a dynamic network with nodes constantly joining and leaving the system. When a new node



(a) Download rates soon after joining the network (800 nodes)

(b) Download rates soon after joining the network (8,000 nodes)

Figure 6.16: When a new node joins the network, it has no reputation with its neighbors regardless of its intended behavior, ADSL or altruistic. Both types of nodes initially have low download rates, thus high packet-loss rates. Altruistic nodes quickly ramp up their download rates close to the stream rate, whereas ADSL nodes remain under 50% of the stream rate. Figure (a) shows the download rate as a fraction of stream rate of ADSL and Altruistic nodes as a function of the time since joining the network in the 800 node network, while Figure (b) shows the corresponding graph for the 8,000 node network.

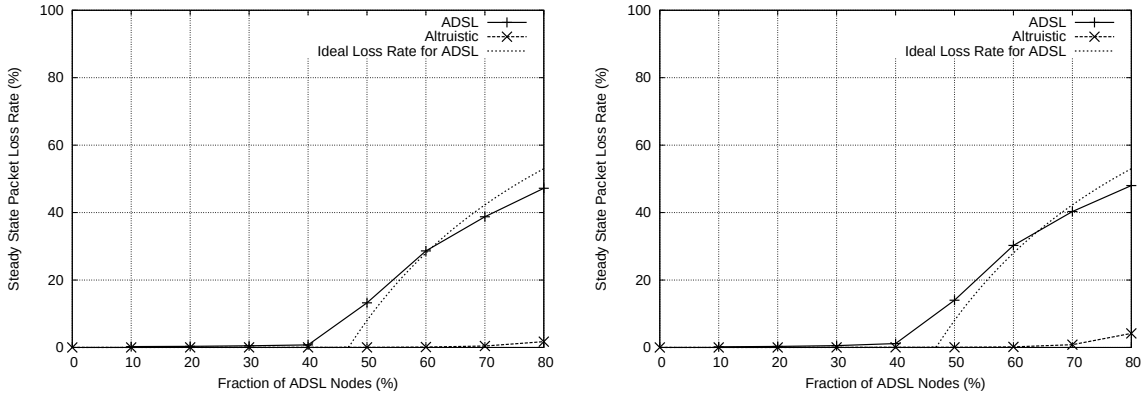
joins the system and connects to other nodes as its neighbors, those neighbors do not have an immediate way to determine whether the newly connected node is likely to contribute upload bandwidth or not. In our system, nodes only rely on first-hand observations rather than long-term reputations or information obtained from other nodes.

Therefore, newly joined nodes are initially treated the same by their neighbors regardless of whether they are an ADSL or altruistic node. However, as they upload data to their neighbors, or do not, in case of ADSL nodes, their neighbors learn and respond to their behavior, leading to better or worse service to those nodes. To be more precise, newly joined nodes initially have no tokens in their private buckets and must compete with all nodes for shared bucket tokens, which will be scarce in a resource-constrained network. However, altruistic nodes will soon begin to accumulate credits in their private bucket as they obtain packets and upload them to other neighbors. ADSL nodes, on the other hand, will accumulate much fewer credits in their private buckets and will continue to compete for scarce shared bucket tokens.

To illustrate this behavior, we analyzed the behavior of an individual altruistic and ADSL node in the resource-constrained experiment with 80% ADSL nodes.

Figure 6.16 shows the download rate (as a fraction of the stream rate) of altruistic nodes compared to ADSL nodes as a function of the time they joined the network, with the 800 node network in sub-figure (a) and 8,000 node network in sub-figure (b). We observe that both ADSL and altruistic nodes initially had low download rates, a small fraction of the stream rate. This leads to high packet loss rates for both classes of nodes in the initial few seconds after they join the system.

Altruistic nodes initially have no data to upload to their neighbors despite a willingness



(a) Steady state packet loss rates (800 nodes)

(b) Steady state packet loss rates (8,000 nodes)

Figure 6.17: Packet loss rates for ADSL and altruistic nodes as a function of the fraction of ADSL nodes in the network. Figure (a) shows the results for the 800 node network, while Figure (b) shows the corresponding graph for the 8,000 node network. The steady state packet loss rate is the packet loss rate experienced by nodes neglecting the first 30 seconds when nodes are still ramping up their download rates.

to do so. However, as they acquire packets and upload them to other neighbors, their upload rate quickly ramps up. In fact, their rate briefly exceeds 100% as they upload enough data to build up credit, and then rapidly download missing packets to fill up their buffer. Their packet loss rate eventually settles down to a value just under 100%, consistent with the low (but non-zero) packet loss rates observed in Section 6.11.2 for altruistic nodes in a severely resource-constrained network with 80% ADSL nodes.

ADSL nodes also see a ramp up in their download rate in the first few seconds after joining because they do upload some data to their neighbors, albeit at a rate much lower than the stream rate. Unlike the altruistic nodes though, the ADSL nodes only manage to ramp up their download rate to around 45–47% of the stream rate. Again, this is consistent with the packet loss rate we observed for ADSL nodes in our previous experiments.

Since both ADSL and altruistic nodes suffer higher packet loss rates in the initial period after they join, we computed the *steady state* packet loss rate for nodes in the network by disregarding the first 30 seconds of packet loss for every node, and plotted the results in Figure 6.17. Once again, sub-figure (a) shows the results for the 800 node network, while sub-figure (b) shows the corresponding graph for the 8,000 node network.

In the steady state, altruistic nodes suffer a packet loss rate of under 2% in the 800 node network, and under 5% in the 8,000 node network even when 80% of the nodes in the system are ADSL nodes.

Thus, we show that the Token Stealing algorithm offers good performance to all nodes in the system when the system is resource-rich, i.e., the supply of bandwidth exceeds the demand. However, when the system is resource-constrained, nodes that contribute the most resources see the best performance. This behavior tends to drive the system towards higher supply to demand ratios either by encouraging low capacity nodes to leave the system, or by raising artificial caps on upload rates, and contributing more resources to the system.

6.12 Change in Resource Availability

In our experiments so far, we have studied systems in steady state, where the fraction of ADSL nodes in the system does not change over the course of the experiments (other than small random fluctuations). In practice, one might expect the mix of nodes to change over time.

A robust incentive mechanism should be able to react to changes in resource availability. If a formerly resource-rich network becomes resource-constrained, the network should react accordingly and transition from offering all nodes low packet-loss rates to ensuring that altruistic nodes receive low packet-loss rates at the expense of ADSL nodes. Similarly, if enough altruistic nodes join the system (or ADSL nodes leave) to make a resource-constrained system resource-rich, the system should take advantage of that fact quickly and offer improved performance to the remaining ADSL nodes who were suffering high packet loss rates.

6.12.1 Resource-Constrained Network Becomes Resource-Rich

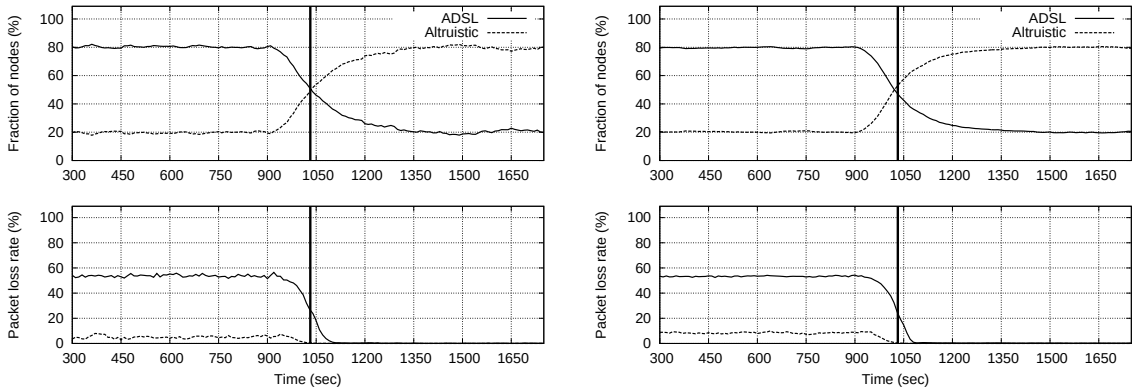
In this experiment we begin with a network with 80% ADSL nodes, a resource-constrained state. Starting at the 900 second mark, we transition the system over to having the inverse proportion of nodes—20% ADSL nodes and 80% altruistic nodes, making it resource-rich.

Abruptly changing the behavior of a large number of nodes in the system to achieve the new distribution would be contrived and unrealistic. Recall that the network in our system is *dynamic* and nodes are constantly joining and leaving. The ADSL and altruistic nodes are drawn from independent Poisson processes to maintain the expected ratio at the desired level. We bring about the change in resource availability by altering the mean inter-arrival rates of the Poisson processes to cause four times as many altruistic nodes to join the system on average as ADSL nodes in any time period. Since all nodes have a mean lifetime of 100 seconds, this gradually leads to the desired outcome of 20% ADSL and 80% altruistic nodes, the inverse of the initial distribution. The transition is nearly complete in about 400 seconds, at the 1,300 second mark.

Figure 6.18a shows the packet loss rate over time for both the 800 node network, and Figure 6.18b shows the corresponding result for the 8,000 node network. In both figures, the top graph shows the proportion of ADSL and altruistic nodes at any point in time, while bottom graph shows the packet loss rates of the two classes of nodes at the corresponding point in time.

As in our previous experiments, with the system in its initial resource-starved state, the ADSL nodes suffer very high packet loss rates, around 57%, while the altruistic nodes suffered much lower packet loss rates, around 5%.

Once the transition towards a more resource-rich state began at the 900 second mark, the system immediately began to take advantage of the additional upload bandwidth available, and both the altruistic and ADSL nodes began to see lower packet loss rates. At the 1,032 second mark, the fraction of ADSL nodes fell under 47% and the system transitioned to being resource-rich, at the point designated by the vertical line. Although it would be theoretically possible for all ADSL nodes to have a 0% packet loss rate, the system is not perfectly ideal, and the ADSL nodes still had a 22% packet loss rate in the 800 node network and 25% in the 8,000 node network at that point. However, the packet loss rate for ADSL nodes fell under 1% in both the 800 and 8,000 node network at the point with about 40% ADSL nodes in the system, when the ratio of supply to demand was 109%.



(a) Node dist. / Packet Loss Rate (800 nodes)

(b) Node dist. / Packet Loss Rate (8,000 nodes)

Figure 6.18: Packet loss rates of altruistic and ADSL nodes over time, as a resource-rich system gradually becomes resource-constrained. The system begins with 80% ADSL and 20% altruistic nodes. Starting at the 900 second mark, it transitions over to the inverse distribution. The top graph in each figure shows the proportion of nodes, while the bottom one shows the packet loss rates. The solid vertical line denotes the break even point where supply=demand.

This shows that our system is able to take advantage of additional resources as they become available, and offer low packet loss rates to all nodes in the system once the system becomes resource-rich. Therefore, as the incentive mechanism encourages individual nodes to contribute more bandwidth to the system, the system is able to take advantage of those resources quickly, leading to improved performance.

6.12.2 Resource-Rich Network Becomes Resource-Constrained

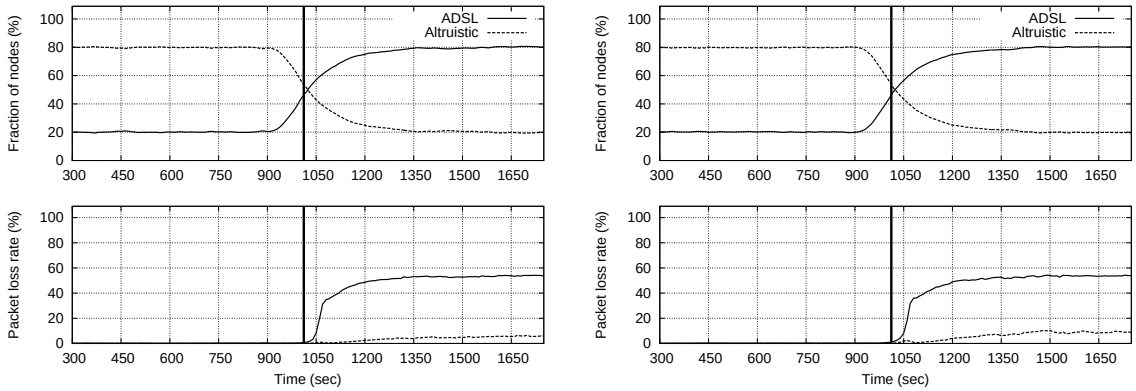
In this experiment, we ran the complementary experiment, where the system was initially resource-rich, but became resource constrained as altruistic nodes left the system and were replaced by ADSL nodes.

Analogous to the previous experiment, we started the experiment with the Poisson generators set to create a resource-rich network with 80% altruistic nodes, and 20% ADSL nodes, and began to transition the system to the inverse distribution beginning at the 900 second mark.

Figure 6.19a shows the packet loss rate over time for both the 800, and Figure 6.19b shows the corresponding figure for the 8,000 node networks. In both figures, the top graph shows the relative proportion of altruistic and ADSL nodes in the system at a given point in time, while the bottom graph shows the packet loss rates of the two classes of nodes at the corresponding point in time.

As expected, initially both altruistic and ADSL nodes have very low packet loss rates since the system is resource-rich. As the system becomes resource-constrained, the ADSL nodes experience a rapid increase in packet loss rates, while the packet loss rate for the altruistic nodes remains relatively low.

This experiment demonstrates that the system responds well to changes in resource availability, and that the packet loss rates of the two classes of nodes are determined by resource



(a) Node Dist. / Packet Loss Rate (800 nodes)

(b) Node Dist. / Packet Loss Rate (8,000 nodes)

Figure 6.19: Packet loss rates of altruistic and ADSL nodes over time, as a resource-constrained system gradually becomes resource-rich. The system begins with 20% ADSL and 80% altruistic nodes. Starting at the 900 second mark, it transitions over to the inverse distribution. The top graph in each figure shows the proportion of nodes, while the bottom one shows the packet loss rates. The solid vertical line denotes the break even point where supply=demand.

availability at any point in time, regardless of the history of how the system got to that state. This is an important property, because it shows that our system adapts well to changes in network conditions.

6.13 Change in Node Behavior

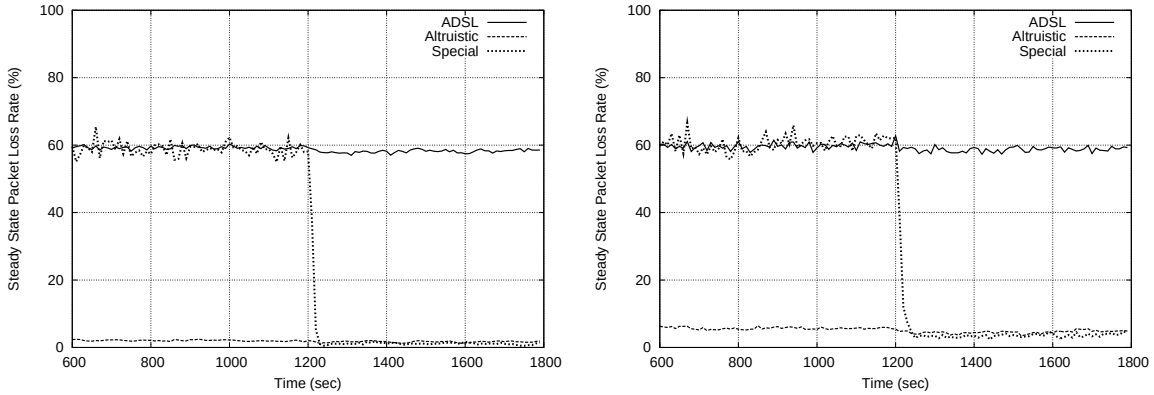
In Section 6.11, all nodes in the system were configured to select a behavior model (altruistic, ADSL, or Fake ADSL) and maintain that behavior throughout the course of their participation in the experiment. In Section 6.11.2 we argued that the difference in packet loss rates between the altruistic and ADSL nodes would encourage nodes to increase their upload rates in order to improve their performance.

Therefore, it is important that former ADSL nodes that increase their upload rates and change their behavior to be similar to the altruistic nodes are quickly rewarded with lower packet loss rates. On the other hand, if an altruistic nodes reduces its upload rate, it should quickly see an increase in packet loss rate to a level similar to that seen by other ADSL nodes.

6.13.1 Fake ADSL Nodes Become Altruistic

In this first experiment, we have a resource-constrained system with 80% ADSL nodes as before. We also introduced a small number of Fake ADSL nodes to the system, denoted by *special* nodes. As before, these represent nodes who have sufficient capacity, but limit their upload rates and masquerade as ADSL nodes.

However, unlike the normal ADSL nodes, the special nodes abruptly change their behavior at the 1,200 second mark. They increase their upload rate to match that of the altruistic nodes. These nodes represent the situation where a node decides to remove an artificial cap on upload



(a) Packet Loss Rate (800 nodes)

(b) Packet Loss Rate (8,000 nodes)

Figure 6.20: ADSL nodes can improve their performance by increasing their upload rate. "Special" nodes initially act as Fake ADSL nodes, but change to altruistic nodes at the 1,200 second mark. On increasing their upload rate, the special nodes quickly see an improvement in performance (i.e., decrease in packet loss rate).

rate in order to receive better performance. As before, we repeated the experiment with both 800 and 8,000 node networks.

Figure 6.20 shows the packet loss rate for the special nodes, with the normal altruistic and ADSL nodes for comparison. Before the 1,200 second mark the special nodes have the same low upload rate as the ADSL nodes, and therefore suffer the same high packet loss rate, around 55–60% as the rest of the ADSL nodes.

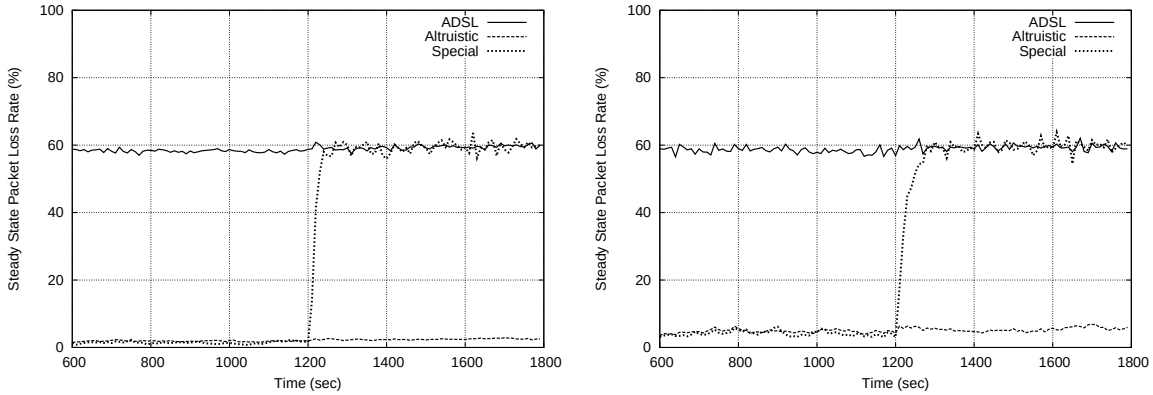
However, at the 1,200 second mark they increase their upload rates to match the altruistic nodes, and their neighbors begin to react immediately: their packet loss rates begin fall. In about 30 seconds, the special nodes have a low packet loss rate, comparable to the remaining altruistic nodes: around 2% in the 800 node network, and 6% in the 8,000 node network.

This shows that the Token Stealing algorithm allows a formerly selfish node to quickly regain good performance and low packet loss rates when they increase their upload rate. One can imagine a GUI client where a user receiving poor quality video pauses another application competing for upload bandwidth, or removes and artificial cap, and notices an immediate improvement in video quality, thus encouraging the user acting out of self-interest to behave in a way that benefits the system.

6.13.2 Altruistic Nodes Become Selfish

In this section, we study the complementary situation, where a set of altruistic nodes in a resource-constrained system abruptly change their behavior and reduce their upload rate to that of the ADSL nodes, in essence turning into Fake ADSL nodes.

As with Section 6.13.1, we begin with a resource-constrained system with 80% ADSL nodes, and introduce a class of special nodes. In this experiment, the special nodes initially join the network as altruistic nodes, and limit their upload rate at the 1,200 second mark. Once again, we repeated the experiment with 800 and 8,000 node networks.



(a) Packet Loss Rate (800 nodes)

(b) Packet Loss Rate (8,000 nodes)

Figure 6.21: Altruistic nodes cannot exploit their reputation indefinitely. The "Special" nodes initially act as altruistic nodes, but reduce their upload rate to that of the ADSL nodes at the 1,200 second mark. Their packet loss rate rapidly increases to the same level as the rest of the ADSL nodes.

Figure 6.21 shows the packet loss rate for the special nodes, with the normal altruistic and ADSL nodes for comparison. Before the 1,200 second mark the special nodes have the same high upload rate as the altruistic nodes, and therefore have good performance with a packet loss rate around 1–2%.

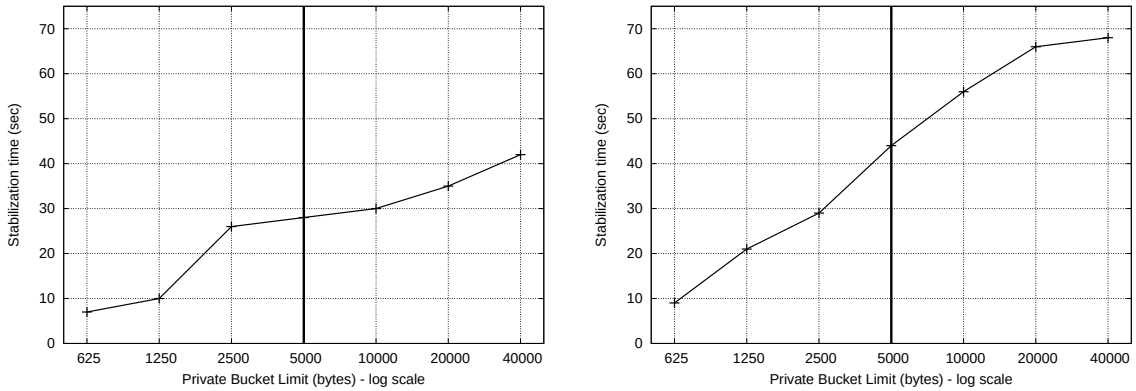
However, at the 1,200 second mark they decrease their upload rates to match the ADSL nodes. As observed before, their neighbors begin to react to this change immediately, and in about 30 seconds their performance is identical to that of the normal ADSL nodes, with a high packet loss rate of 55–60%.

This shows that the Token Stealing algorithm does not allow nodes to establish a good reputation and exploit that reputation to avoid uploading data to the network in the future. Nodes monitor their neighbors' behavior constantly, and rapidly react to changes. In order to maintain low packet loss rates, nodes must continue to exhibit altruistic behavior and contribute upload bandwidth to the system.

6.14 Stabilization Time

In Section 6.11, we showed that in a resource-constrained system, all nodes are initially treated the same and even nodes that are willing to upload data will suffer packets loss initially until their neighbors observe and react to their altruistic behavior. Likewise, a node that increases or decreases its upload rate will not be rewarded or penalized for a few seconds. We refer to this period as the *stabilization time*.

A short stabilization time is valuable from a user experience standpoint—a user that causes the upload rate to decrease below the stream rate (for instance by throttling it, or starting another application that competes for bandwidth) should quickly see a reduction in quality of the stream so they observe the connection between the two events even if they do not understand the technical details. Similarly, allowing more upload bandwidth should result in a prompt increase in



(a) Stabilization Time (800 nodes)

(b) Stabilization Time (8,000 nodes)

Figure 6.22: Stabilization time is the time for a node that alters its behavior from altruistic to ADSL (or vice versa) to see a complete transition in packet loss rate. This time is governed by the Private Bucket Limit. Lower limits allow quicker transitions. The vertical line marks the default value of the Private Bucket Limit (5,000).

quality.

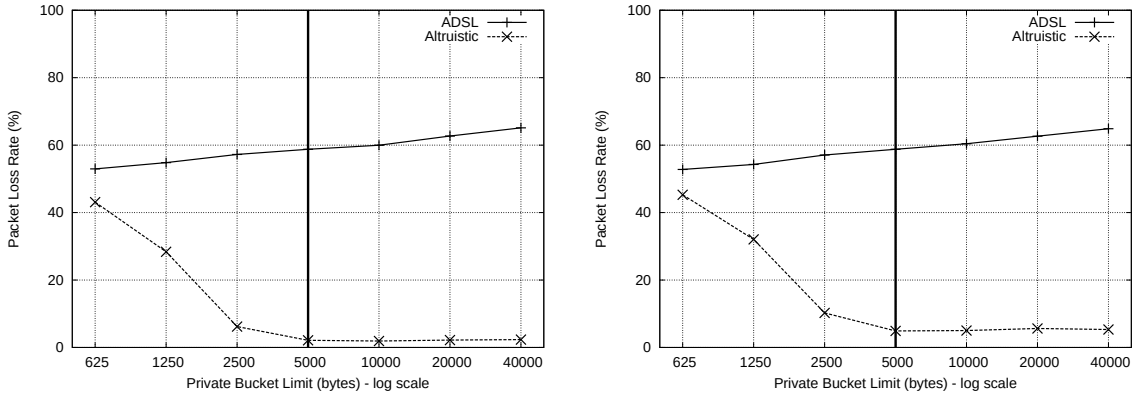
The stabilization time is governed by the *private bucket limit* parameter, which is the maximum number of private bucket tokens a neighbor will allow a node to accumulate. As discussed in Section 5.4, it is undesirable to allow a node to accumulate credit indefinitely, so private buckets are capped at the *private bucket limit*.

A node that has been uploading data consistently for a long time will most likely have a private bucket filled near the limit. If it reduces its upload rate to a value below the stream rate, it will begin to consume those tokens, and will eventually empty out the private bucket and have the same performance as the remaining ADSL nodes. The higher the limit, the longer this transition will take.

We ran a series of experiments similar to the experiment in Section 6.13.2 where nodes abruptly changed their upload rates half-way through the experiment. We varied the values of *private bucket limit*, beginning with the default value of 5,000 and successively doubling and halving it to study a range of values from 625 to 40,000. We measured the time it took for a set of altruistic nodes that change their behavior to mimic the Fake ADSL nodes to have their packet loss rate increase from the low level seen by the altruistic nodes to the higher level of the ADSL nodes. Once again, we repeated our experiment with 800 and 8,000 node networks.

Figure 6.22 shows the stabilization time as a function of the private bucket limit. As expected, the stabilization time increases with a higher limit and decreases with a lower limit.

It might be tempting to use a very low private bucket limit in order to quickly penalize nodes that stop being altruistic, or rewards ADSL nodes that increase their upload rates. However, there is a trade-off. If the private bucket limit is very small, nodes are more likely to be affected by small random fluctuations in upload rates. If an altruistic node's private bucket is easily depleted, it will often be indistinguishable from the nodes with a history of selfish behavior. This might lead it to compete with the ADSL nodes for scarce shared bucket tokens, leading to higher packet loss rates. Such events benefit the ADSL nodes, because when the altruistic nodes



(a) Packet Loss Rate (800 nodes)

(b) Packet Loss Rate (8,000 nodes)

Figure 6.23: The stabilization time for the system to react to changes in node behavior is determined by the private bucket limit used by neighbors.

lose their preferred treatment temporarily, that leaves more tokens in the shared bucket that they can compete for on equal terms.

Figure 6.23 shows the packet loss rates of ADSL and altruistic nodes as a function of the private bucket limit. As before, the ADSL nodes have significantly higher packet loss rates than the altruistic nodes. However, at the low end of the graph (i.e., small limit), the gap between the two classes of nodes shrinks significantly, as more bandwidth is diverted from the altruistic to the ADSL nodes.

It might seem surprising at first glance that although the packet loss rate of ADSL nodes decreases by barely 10% from 5,000 to 625, that leads to an increase of a little over 40% in the packet loss rate of the altruistic nodes. However, recall that this network has 80% ADSL nodes and 20% altruistic nodes. Therefore, the ADSL nodes outnumber the altruistic nodes 4:1, so every byte uploaded to ADSL nodes on average costs the altruistic nodes four bytes.

Even as the packet loss rate of the altruistic nodes falls to values close to zero, the packet loss rate of the ADSL nodes continues to increase with increasing values of the *private bucket limit*. This is because a higher limit allows altruistic nodes to reserve bandwidth for future use, so a larger limit will allow larger amounts to be reserved even if it is never redeemed. This leads to under-utilization of total upload capacity in the network, and is the reason we introduced this parameter to cap private bucket values.

6.15 Selective Connection

In Section 6.13.2 we showed that nodes that have a history of uploading data faster than the stream rate will not be able to take advantage of their good reputation indefinitely, and will begin to see higher packet loss rates immediately on reducing their upload rate. Thus, uploading data rapidly for a little while and then cutting back would not be an effective strategy to game the system.

In this section, we study another method that selfish nodes might attempt to use to gain an unfair advantage while limiting their upload rates. Nodes may selectively connect to high-

Percentage of ADSL-Game	Network Size	Packet Loss Rate (%)		
		Altruistic	ADSL	ADSL-Game
10	800	7.0	57.7	57.5
50	800	6.7	57.1	57.3
90	800	6.8	56.1	57.0
10	8,000	10.7	57.7	57.8
50	8,000	10.0	57.3	57.3
90	8,000	10.5	57.5	57.6

Table 6.3: A fraction of ADSL nodes, denoted by "ADSL-Game" used a strategy of connecting only to altruistic nodes in an attempt to game the system. Regardless of whether a small number of ADSL nodes (10%) or a vast majority (90%) of the ADSL nodes adopted this strategy, there was neither an improvement in their own packet loss rates, nor an adverse effect on the altruistic or normal ADSL nodes.

bandwidth nodes in an attempt to take advantage of their excess capacity, thus reducing their packet loss rate compared to other ADSL nodes in a resource-constrained system. We study resource-starved system in this section, because any gaming strategy would be redundant in a resource-rich system in our model, because all nodes receive good performance while the system is resource-rich.

In the resource-constrained system with 80% ADSL nodes, we modified 10% of the ADSL nodes to game the system. They refuse to peer with all nodes except high-bandwidth (altruistic) nodes. For brevity, we refer to these nodes that try to game the system as the ADSL-game nodes.

We observed that the strategy is completely ineffective; the performance of the ADSL-game nodes is identical to that of the rest of the ADSL nodes. The ADSL-game nodes had a mean packet loss rate of 57.5% and the regular ADSL nodes had a mean packet loss rate of 57.7%. Altruistic nodes, on the other hand, had a packet loss rate of only 7.04%.

In order to see if the strategy would be successful if large numbers of nodes adopt it, or potentially harmful to altruistic nodes, we ran additional experiments with 50% and 90% of ADSL nodes converted to ADSL-game nodes. Once again, we observed that the ADSL-game nodes gain no benefit at all.

As before, we also repeated the experiment with the 8,000 node network. Once again, we observed that the ADSL-game nodes failed to gain an advantage regardless of whether 10%, 50%, or 90% of the non-altruistic nodes adopted the strategy. The results of the experiments with 10%, 50%, and 90% of ADSL nodes converted to ADSL-game nodes with both the 800 and 8,000 node networks are summarized in Table 6.3.

The fact that ADSL-game nodes fail to gain an advantage is not unexpected. In a resource-constrained system, altruistic nodes have no unused upload capacity, so their shared buckets tend to be empty. The ADSL-game nodes compete with all other nodes for the scarce tokens in the shared bucket and receive no special benefit regardless of whether they are competing with regular ADSL nodes or other ADSL-game nodes. Moreover, altruistic nodes connected to other altruistic nodes will generally have a supply of tokens in their private buckets, thus allowing them to continue receiving bandwidth.

Even in an extreme case where the ADSL-game nodes managed to completely surround an

altruistic node by preventing it from connecting to any other types of nodes, the ADSL-game nodes would not gain an advantage, since the isolated altruistic node would have no data to offer without other nodes to receive them from. Such an attack would constitute a denial-of-service attack rather than a successful gaming of the system. Such actions do not benefit self-interested rational nodes, and is thus outside the scope of the Token Stealing protocol, and is a subject for future research. We discuss these aspects in further detail in Section 9.1.

6.16 Range of Upload Rates

In previous experiments, we used distinct classes of nodes with discrete values of upload rates (i.e., ADSL and Altruistic nodes). Although that setup is very amenable to systematic study, it is somewhat contrived; in the real world, upload rates are unlikely to fall neatly into a few distinct classes.

In this experiment, we demonstrate that our system works as expected even when upload rates are distributed across a range. Instead of being assigned to classes, nodes are randomly assigned upload rates between the 64kbps (the value used by ADSL nodes) and 320kbps (the value used by altruistic nodes). In order to allow control over the overall supply to demand ratio of the network, we assigned values using the following distribution:

$$u(x) = a + bx^{\gamma} \tag{6.2}$$

where x is a random variable $\in (0, 1)$. Using a non-linear function enables us to vary the supply to demand ratio of the system while maintaining fixed bounds on the maximum and minimum values of upload rates. The mean value of this function is:

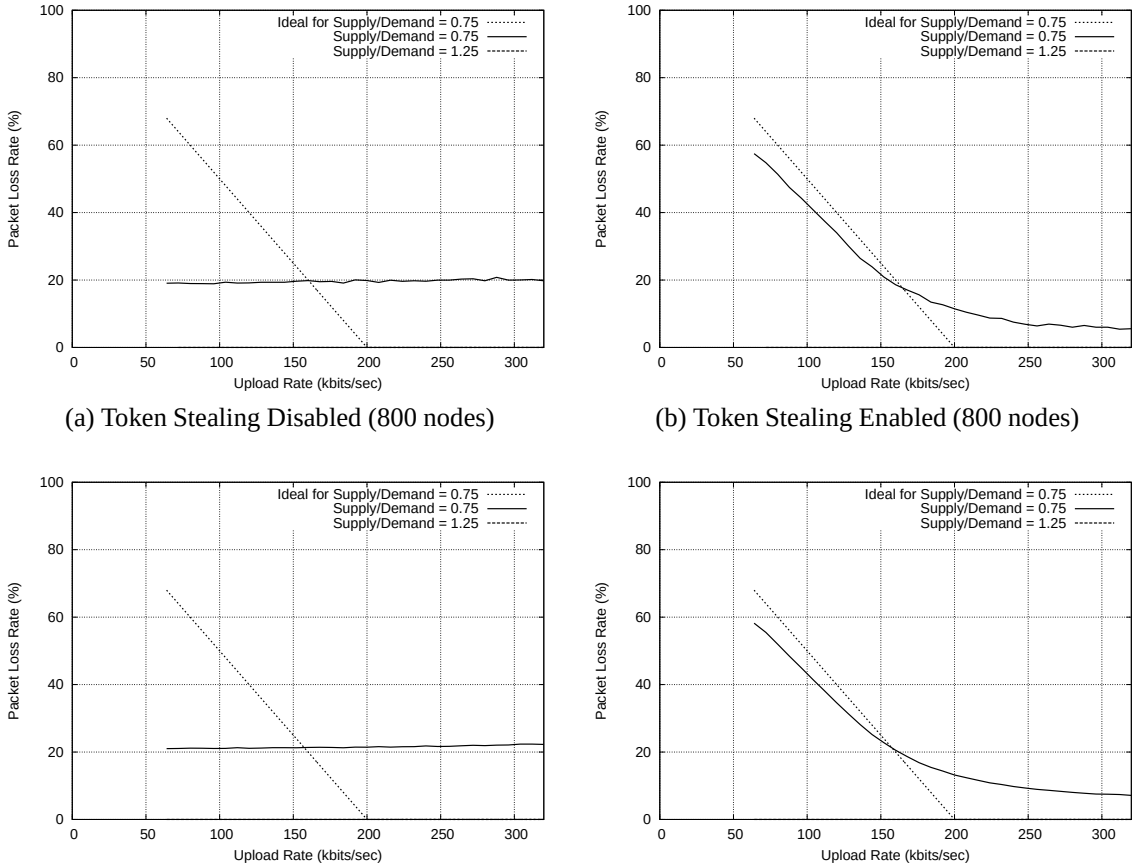
$$\bar{u} = a + \frac{b}{\gamma + 1} \tag{6.3}$$

The values of $a = 64$kbps and $b = 256$kbps, gives us the desired minimum and maximum value of 64kbps and 320kbps, respectively.

We ran two experiments: one resource rich, with a supply to demand ratio of 1.25 ($\gamma = 1.7$, $\bar{u} = 250$kbps), and one resource starved, with a supply to demand ratio of 0.75 ($\gamma = 0.35$, $\bar{u} = 150$kbps). We ran each scenario once with the Token Stealing algorithm disable, and one with the algorithm enabled. As before, we repeated each experiment with an 800 node network, and an 8,000 node network.

Figure 6.24a shows the packet loss rates of the resource-rich and resource-constrained networks with 800 nodes, as a function of a upload rate. In the resource-rich system, all nodes saw very low levels of packet loss. In the resource-constrained system, however, all nodes suffer a relatively high packet loss rate of 19.6% regardless of their upload rates. Figure 6.24c shows the corresponding results for the network with 8,000 nodes. Once again, we saw very similar results, with very low packet loss rates in the resource-rich network, and a higher uniform rate of 21.5% in the resource-constrained case.

A node would have very little incentive to increase its upload rate. In fact, as we observed before in Figure 6.13, there is a small inverse correlation between upload rate and performance, i.e., nodes that upload less see somewhat better performance. As before, this is because nodes that upload less data leave their upstream bandwidth free to make packet requests more quickly.



(a) Token Stealing Disabled (800 nodes)

(b) Token Stealing Enabled (800 nodes)

(c) Token Stealing Disabled (8,000 nodes)

(d) Token Stealing Enabled (8,000 nodes)

Figure 6.24: In this experiment, nodes picked upload rates over a distribution rather than using distinct classes. Each graph shows the packet loss rate vs. upload rate for resource-rich network (Supply/Demand = 1.25), a resource-constrained network (Supply/Demand = 0.75), and the ideal curve for the resource-constrained case. Ideal packet loss rate for the resource-rich network is zero for all values. Figure (a) shows the results for the 800 node network with the Token Stealing algorithm disabled, while Figure (b) shows the graph with the algorithm enabled. Figures (c) and (d) show the corresponding graphs for the 8,000 node network.

With the Token Stealing algorithm enabled, however, there is a strong positive correlation between upload rate and performance, as desired: nodes that upload more data see better performance.

Figure 6.24b shows the packet loss rates for the resource-rich and resource-constrained network with the Token Stealing algorithm enabled. As before, the packet loss rate for all nodes is very close to zero in the resource-rich case. In the resource-constrained case, however, there is a clear correlation between the packet loss rate observed and upload rates: the higher the upload rate, the lower the packet loss rate. The mean system-wide packet loss rate was 20.2%. Figure 6.24d shows the corresponding results with the 8,000 node network. Once again we observe very similar results, with packet loss rates near zero for the resource-rich network, and similar correlation between upload rate and packet loss rate in the resource-constrained case, with a

mean of 21.6%.

The dotted line shows the ideal packet loss curve for the resource-constrained case (ideal is of course zero for all nodes in the resource-rich case). Ideally, all packet losses would be borne by nodes that upload less than the stream rate of 200 kbits/sec. Nodes that upload less than the stream rate will suffer a higher packet loss rate, proportional to the deficit they create (i.e., the difference between the stream rate and their upload rate). Although it is not perfectly ideal, the observed curve does closely approximate the ideal curve. A node in this system would have a strong incentive to upload more if they are able to in order to reduce their own packet loss rate.

6.17 Prototype Implementation on PlanetLab

We built a prototype implementation of our system in C++ in order to validate our simulation results. The prototype consists of over 5,000 lines of code with a much of the application logic derived from the simulator, with changes to interface with the operating system's networking stack.

In order to validate our simulator, we ran several experiments on the PlanetLab [18] testbed. PlanetLab is a network research testbed with approximately 900 physical machines located at institutions (primarily research universities) around the world, and accessible to researchers from participating institutions.

Although PlanetLab is an invaluable tool for networking research, it is unfortunately a victim of its success, in some ways. PlanetLab hosts tend to be very heavily loaded, and hardware or network problems are not always addressed promptly because they are remotely managed. Therefore, at the time of writing, only 380 hosts were accessible. Moreover, the set of nodes accessible often varied from one experiment to the next. As the testbed is simultaneously shared by hundreds of researchers, and even hosts a few popular services like the CoDeeN [96] and Coral [36] Content Distribution Networks, nodes tend to be under very heavy load. Typical load averages tend to be higher than 10, with load averages over 50 quite common. Most nodes were also found to be actively swapping most of the time. Therefore, for a time-sensitive application like live streaming, we consider the PlanetLab experiments to be a stress-test rather than a typical real-life deployment.

In this section, we present the results of our experiments on PlanetLab.

6.17.1 System with All Altruistic Nodes

In this experiment we used the same parameters as we did in Section 6.3, and attempted to launch one Chainsaw node on each physical PlanetLab node.

The mean size of the network in this experiment was 350 with 3,431 nodes participating over the course of the run. Figure 6.25 shows the distribution of packet loss rates and startup times. We found that 3,209 (93%) of the nodes suffered no packet loss at all over the course of the experiment. The system-wide mean packet loss rate was 1.62%.

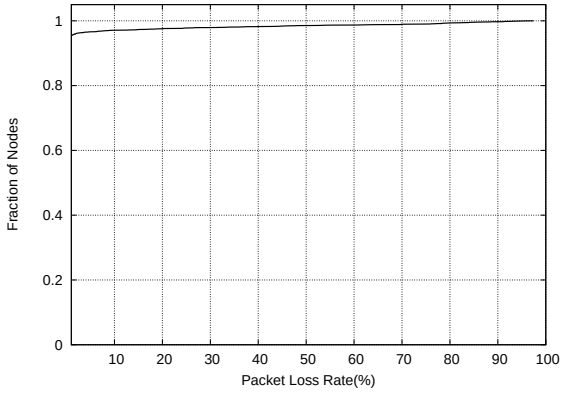
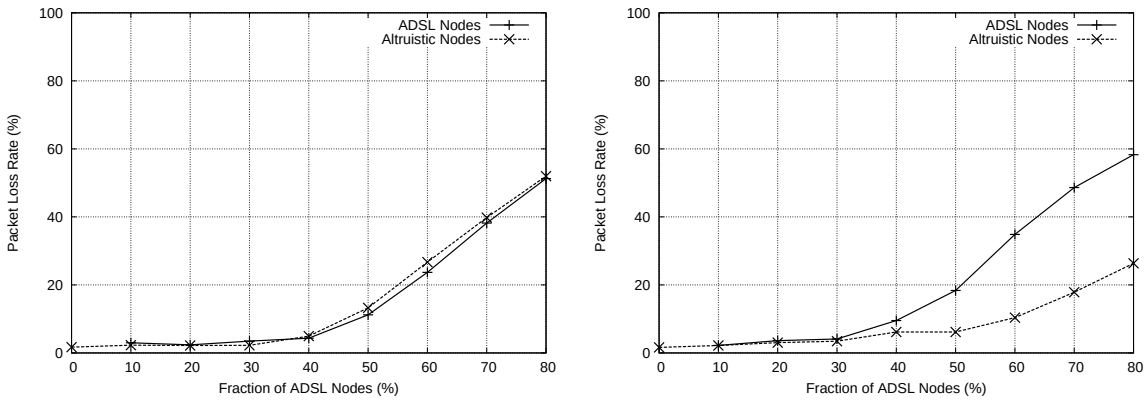


Figure 6.25: Cumulative distribution of packet loss rates on PlanetLab.

6.17.2 Resource-Constrained Systems

We then ran a series of experiments with different fractions of ADSL nodes with the same parameters as the experiments in Section 6.11. Most PlanetLab nodes actually have raw upload capacities higher than we limited their upload rate to. Therefore they can be considered to be Fake ADSL nodes. As before, we repeated the series of experiments with the Token Stealing algorithm disabled.



(a) Token Stealing Disabled

(b) Token Stealing Enabled

Figure 6.26: Performance with varying fractions of ADSL nodes on PlanetLab with token stealing enabled and disabled.

Figure 6.26a shows the relative packet loss rates of the ADSL and altruistic nodes with the Token Stealing algorithm disabled. As with the simulation results, we find that the difference in performance seen by the two classes of nodes is small. Moreover, we find that altruistic nodes have slightly higher packet loss rates than the ADSL nodes when the system is resource-constrained.

Figure 6.26b shows the relative packet loss rates of the ADSL and altruistic nodes with the Token Stealing algorithm enabled. In this series of experiments, there is a clear difference between the performance seen by the two classes of nodes. ADSL nodes have packet loss rates

more than twice as high as the altruistic nodes, 58% compared to 26%. As before, both classes of nodes have low packet loss rates while the system is resource-rich.

This series of experiments offers further evidence that the Token Stealing algorithm is capable of giving much better performance to nodes that contribute bandwidth than those that do not in a resource-constrained system, while giving good performance to all nodes in a resource-rich system, and corroborates our simulation results.

Summary

In this chapter, we presented simulation results that demonstrate that the Chainsaw streaming protocol supports high bandwidth streaming with low packet loss rates, low delay and quick startup times. We showed that the system scales well with size and stream rate, and is robust to churn. We also demonstrated that the Token Stealing algorithm achieves our goal of giving lower packet loss rates to nodes that contribute bandwidth in resource-rich systems, while taking advantage of altruistic nodes to give low packet loss rates to all nodes when possible. We showed that the algorithm is resistant to gaming, and responds quickly to changes in both an individual node's upload rates, and system-wide resource availability. Finally, we validated our simulation results with by repeating some of the experiments with a prototype implementation of PlanetLab, and demonstrating similar results.

Chapter 7

Related Work

Peer-to-peer systems have been extensively studied in recent years. In this chapter, we present related work in file sharing, incentives and streaming to place our work in context. We begin with a survey of key peer-to-peer systems in general, and then outline other approaches to streaming protocols as well as incentive mechanisms.

In Section 7.1 we give a broad overview of research in peer-to-peer networking. Section 7.2 we describe some of the key file-transfer protocols that serve as background for our work. In Section 7.3 we describe work on peer-to-peer streaming and overlay multicast and contrast it with our own work. Finally, in Section 7.4 we present related work on incentive mechanisms for file sharing as well as streaming.

7.1 Peer-to-Peer Systems

The peer-to-peer revolution was arguably started by the Napster [31] file sharing network. Napster allowed users to directly share files off of their computer with other users on the Internet and was an alternative to more traditional means of sharing content like FTP and HTTP servers. The peer-to-peer approach dramatically reduces the load on the central server compared to a traditional server-client approaches because the server only stores metadata: storage and bandwidth costs of the files themselves are borne by the end users.

Napster's network was highly centralized: users sent a list of files they wished to share to a central server which maintained an index of all shared files and the users who were sharing them. Other users could then search the database and contact the original user and download the file directly from them. However, this approach still required significant resources from the central server because it had to process every query by every participant on the system. Moreover, this centralization left Napster legally vulnerable to illegal actions of its users.

Gossip protocols [28, 29] were modeled after the spread of disease among a population. Nodes connected in a random graph sent a message to either all or a random subset of their neighbors. The neighbors in turn forwarded the message to their neighbors excluding the one they received the message from. This protocol ensured very reliable delivery of the message to every node without the need for elaborate routing algorithms.

The Gnutella [40] used a similar principle to build a system more decentralized than Napster. In Gnutella, peers merely used well-known server to locate other peers and join the system. Hav-

ing located a small number of participants, peers performed a random walk from those neighbors in order to locate a random set of neighbors to connect to, resulting in an unstructured random graph. Instead of being directed at a central server, queries were directed at individual nodes by flooding. Gnutella was tremendously popular with over a million unique participants [84], thus demonstrating the potential of peer-to-peer networks. However, Gnutella also suffered from scalability concerns because of its flooding approach.

Several researchers focused on Distributed Hash Tables [62, 80, 87, 103] to solve this problem. Distributed Hash Tables are systems that map keys to values in the same way that regular hash tables do. However, instead of pointing to values in a memory location, keys in a Distributed Hash tables help locate the node in the network that actually stores the corresponding value. These *structured* networks made searches more efficient, but were significantly more complex to build, and made join and leave operations more expensive for the network.

Other researchers focused on making unstructured networks more scalable. Lv et al. [58] proposed replacing flooding with random walks to make better use of network resources. This system was further improved upon by Adamic et al. [1] by directing walks preferentially at high-degree nodes, and by Chawathe et al. [14] by explicitly taking nodes' resource constraints into account. Kazaa [68] is a commercial network that improves scalability by directing searches at well-provisioned *super-nodes*.

7.2 File Transfer Protocols

With high-bandwidth Internet connections becoming ubiquitous, a whole range of new applications have become possible including online distribution of large software packages (ISO images of Linux distributions, demo and free games, etc.), movies, and TV shows.

The data dissemination problem can broadly be classified into *file-transfer* and *streaming* protocols. The objective of file-transfer protocols is to distribute a large, finite-length file (or collection of files) from the distributor to a large number of recipients. In general, different participants complete the download at different times, and the file is only useful once a complete copy is obtained. Streaming networks, on the other hand, aim to deliver a continuous stream of data to clients. Typical examples of streaming applications are online radio and TV stations.

Streaming systems share many of the same challenges as file-transfer protocols but impose additional bandwidth and delay constraints. Therefore, we begin by studying file transfer protocols before addressing our primary goal of building a scalable and robust peer-to-peer streaming system.

BitTorrent [19] is a peer-to-peer file sharing network that has emerged as one of the most popular peer-to-peer networks today. Some estimates suggest that BitTorrent traffic accounts for 35% of all data on the Internet today [93].

The core BitTorrent protocol is very simple. The file being shared is broken up into a number of *pieces* which are assigned sequence numbers. Nodes form an unstructured random graph and exchange piece availability information with their neighbors. Nodes then attempt to assemble a complete copy of the file by requesting missing pieces from their neighbors. Experience with BitTorrent suggests that it often takes several minutes for nodes to achieve full download speed [99]. Although this delay is not a problem when downloading a large file, a few minutes delay would be completely unacceptable to users waiting to start viewing a stream.

In 2008, Mol et al. [66] proposed and implemented an extension of the BitTorrent protocol that is more suitable for on-demand streaming, along with a new incentive mechanism. We discuss the streaming protocol in Section 7.3.3 and the incentive mechanism in Section 7.4.2.

7.3 Streaming and Multicast

In 1988, Deering proposed the Distance-Vector IP Multicast Routing Protocol (DVMRP) [22], an extension to the IP protocol to support multicast, i.e., one-to-many transmission. IP multicast naturally lends itself to applications like streaming [27, 57, 63, 81] and large-scale file distribution [10]. In DVMRP, data is delivered from the source to the recipients by constructing a tree consisting of the union of unicast paths from each recipient to the sender. This tree-based protocol prevents routing loops and ensures that each packet traverses the fewest physical links necessary to transmit to all recipients.

IP has gained wide acceptance in the research community and has been widely studied; protocols have been designed to build reliable services on top of the best-effort IP Multicast layer [56, 76, 100]. However, it requires routers to maintain membership information and violates their stateless design, increasing complexity in the network [83] and leading to scalability concerns. This, combined with the need for widespread infrastructure-level changes, has prevented the widespread deployment of IP multicast.

In 2000, Chu et al. [16] suggested that many of the deployability and scalability concerns that have prevented wide-scale deployment of inter-domain IP Multicast may be mitigated by moving to an application-layer multicast. In 2004, Sripanidkulchai et al. [86] did a measurement study that found that the network does indeed have enough resources to support large scale overlay multicast but did not propose specific protocols.

7.3.1 Tree-Based Approaches

Most of the early streaming approaches used a *tree-based* approach similar to the multicast tree constructed by IP Multicast. These are referred to as *overlay trees*, because they are a logical structure constructed on top of the underlying IP network.

In their landmark paper, Chu et al. proposed a system called End System Multicast (ESM), based on their algorithm called Narada, an adaptation of the DVMRP at the application layer. Through experiments and analysis, they show that it is possible to support multicast at the application layer. They went on to build a practical application that has enjoyed a fair degree of success and has been used to broadcast video from several events like academic conferences [17, 79]. One of the drawbacks of ESM was that it required nodes to maintain state about every other node in the system, limiting its use to networks of tens to a few hundreds of nodes.

Other researchers improved this technique by refining the tree construction techniques using DHTs [13, 35, 42, 55, 71, 104]. CoopNet [71] offers another interesting take on multicast where multicast trees are used to augment the traditional client-server model. When overloaded, the server redirects new clients to other clients it has served recently, in a model the authors call *Cooperative Networking*.

There are drawbacks inherent to networks based on simple overlay tree. Firstly, the network is fragile, because there is only one path from any node to the source (the root of the tree).

Therefore, whenever a node leaves the system, all of its descendants are affected. If the node happens to be near the root, this could include a large fraction of the network. The tree must be repaired quickly to avoid wide-scale disruption. Although this is also true of IP multicast, nodes in an IP multicast tree are core Internet routers which tend to be more stable and reliable than end hosts.

Secondly, load is distributed unfairly. Interior nodes are usually responsible for supporting multiple children, and thus need to contribute two or more times the bandwidth than they receive from their parent. Leaf nodes, on the other hands, contribute no bandwidth at all. In a balanced binary tree, for example, half the nodes are leaf nodes, placing the burden of supporting the entire network on only half the nodes.

Probabilistic Reliable Multicast [6] is an innovative system that addresses some of the reliability problems of tree-based multicast. In addition to the spanning tree, nodes forward data to a randomly chosen set of peers with small probability. Nodes that receive these broadcasts propagate them both upwards toward to root, and to their children. The authors observe that the larger the size of a disconnected subtree, the greater the probability of some member of that subtree receiving one of these random broadcasts. Thus with a small overhead (in terms of duplicate packets) it is possible to greatly increase the reliability of tree-based protocols. Although this solves the problem of interior nodes disrupting large numbers of descendants, it does not address the unfairness of the simple tree-based model.

7.3.2 Multi-Tree Protocols

Some suggested that many of the drawbacks of tree-based approaches may be mitigated by building *multiple* trees. Splitstream [11] is one such system. In Splitstream, the stream is divided into several *stripes*, and one overlay tree is built for each stripe. Splitstream uses the Scribe [12] overlay multicast system, which itself is based on the Pastry [82] Distributed Hash Table (DHT).

Splitstream mitigates many of the drawbacks of simple tree-based approaches. Nodes join as an interior node in only one of the trees, while joining as a leaf node in the rest. This ensures that all nodes in the system have similar levels of contribution. Although this protocol cannot prevent packet loss from propagating down an individual tree, stripes may be redundant, allowing nodes to recover from packets lost in individual trees. Splitstream, however, assumes that all nodes in the network contribute upload bandwidth at the source rate, and does not tackle the issue of allocating bandwidth among heterogeneous peers. However, further enhancements to Splitstream might be able to address this limitation through more complex tree structures.

Venkataraman et al. propose Chunkyspread [94, 95], an alternative multi-tree protocol. Unlike Splitstream, Chunkyspread explicitly takes heterogeneous bandwidth constraints into consideration by assigning nodes with more resources as interior nodes in more trees. This trades off some of the fairness aspects of Splitstream for shorter trees and improved performance. Although their system offers good performance when nodes contribute bandwidth, it offers no incentive for selfish nodes to do so.

7.3.3 Mesh-Based Protocols

A different approach to multi-tree protocols is to do away with trees entirely, and instead construct a *mesh*. This is the approach used in our protocol. The key benefit of a mesh-based

approach over tree-based approaches is that there are multiple paths from the source to each node. Therefore, the loss of a single node has little impact on the integrity of the network. Magharei et al. published a good survey [59] of relevant work as well as detailed comparisons between multi-tree and mesh based approaches.

Gossip or epidemic models are an example of a simple mesh-based multicast system [28, 29]. Initially designed to model the spread of infections in an epidemic, the same principle can be used to disseminate information among a group of nodes. In a gossip network, nodes forward all the data they receive to some or all of their neighbors, thus flooding the network with updates. Nodes keep track of all data they have seen recently to avoid sending the same update multiple times and causing routing loops. Nodes receive all updates with very high probability, thus making the system very reliable, but they will mostly likely receive several copies of each packet. This wastes a lot of bandwidth, making the system unsuitable for high bandwidth applications.

Bullet [51] was one of the first mesh-based protocols designed for high-bandwidth streaming. Bullet used an overlay tree as the primary path for data, but augmented that with a mesh. Nodes receive data from their parents, but that data may be incomplete due to bandwidth or other constraints. The authors proposed an algorithm called RanSub [50] to identify nodes with a largely disjoint set of packets. Nodes use the RanSub algorithm to identify and receive updates from nodes with a disjoint subset of data in order to fill in missing data. Bullet avoided per-packet routing decisions through the RanSub algorithm. However, this did result in a large amount of overhead because the RanSub algorithm is probabilistic, and nodes inevitably received duplicate copies of data. In the Bullet protocol, there is unlikely to be a bidirectional flow of data between any given pair of interacting nodes, thus making it difficult to apply pairwise incentives.

Bullet is an example of a *structured* mesh because the structure is dictated by an algorithm and data routing is dependent on the structure of the network. An alternative is an *unstructured* mesh where nodes form a random graph. This approach is commonly used by file sharing protocols, including Gnutella, Kazaa, BitTorrent and others, but early streaming protocols focused on tree-based and other structured networks in order to simplify routing decisions and minimize delay. However, we showed [72] that with a receiver-driven architecture, an unstructured mesh network can be used to support high-bandwidth streaming with both low packet loss rates and low delay. We expanded on that technique, as detailed in Chapter 4.

Biskupsi et al. proposed MeshCast [8], an extension of our Chainsaw protocol. They showed that they were able to achieve lower delay and support higher stream rates by taking node bandwidth into consideration while constructing the network topology. The also showed that their changes did not affect the resilience or other desirable characteristics of the Chainsaw protocol. In fact, their extension does not conflict with the incentive mechanism we present in Chapter 5, thus further demonstrating the benefit of decoupling data routing from network structure.

Coolstreaming/DONet [102] is another streaming protocol based on an unstructured mesh. Although they are also based on a pull-based system like Chainsaw, they have different goals. Their protocol aims to minimize overhead, at the cost of startup time, and control granularity. Although their approach does support high-bandwidth streaming, is not amenable to pairwise incentives, which was one of our key goals in developing the Chainsaw protocol.

In 2008 Mol et al. proposed an extension of the BitTorrent protocol [66] that targets on-demand streaming. On-demand streaming is a different problem that the live streaming problem

we address, in that different viewers may start watching a given program at different times. On-demand streaming poses different challenges from live streaming because different viewers will not be synchronized in time, while the entire program will be available ahead of time. In their system, peers divide the file into high, medium, and low priority packets depending on how soon the packets are needed. When requesting data from their neighbors, peers pick packets to request with different probability depending on the class the packet falls in. This is a somewhat different take on the sliding window protocol used in Chainsaw. Since it remains a receiver-driven architecture, we could apply this buffer management strategy to Chainsaw and possibly improve performance further. We discuss this and other possible enhancements in Chapter 9.

7.4 Incentive Mechanisms and Resource Allocation

Incentive mechanisms are mechanisms designed to discourage nodes from *free-riding*, i.e., taking advantage of the system without contributing resources in return. In this section, we present a survey of other incentive mechanisms and contrast them with the algorithm we presented in Chapter 5.

Incentive models can broadly be classified into cooperative and non-cooperative. The cooperative model relies on participants to follow the protocol faithfully, even when it is detrimental to their self-interest to do so. A selfish node may be able to gain an unfair advantage by misreporting information (upload rates, for instance) or by circumventing the protocol. A non-cooperative model offers a stronger guarantee either by avoiding the dependence on self-reported information, or by using various methods (cryptography, for example) to make it impossible to falsify reporting.

Incentive models can also be orthogonally classified into reputation-based systems and pairwise-systems. In a reputation system, nodes directly or indirectly receive information about their neighbors from other nodes in the system, and adjust their behavior accordingly. In a pairwise model, however, nodes rely only on direct observation, which enforces a stronger constraint while using less information. This is the approach we take in our Token Stealing algorithm.

7.4.1 File Sharing Protocols

Incentive mechanisms for file sharing protocols do not necessarily apply directly to streaming, but are an interesting point of comparison.

In addition to the basic file-transfer protocol, BitTorrent also specifies an incentive mechanism. BitTorrent clients rank their neighbors by relative upload rates and *choke*, i.e., stop sending data to all but the top few clients (typically four or so). Clients also randomly unchoke other neighbors for brief periods to probe for other neighbors who may by able to upload even faster given a chance. As a result, nodes that contribute more upload bandwidth tend to receive faster downloads.

The BitTorrent algorithm has been studied widely, and many studies have found [20, 52, 78] that the incentive model works quite well with the unmodified client, where the only option available to strategic clients are tweaking parameters like upload bandwidth and number of neighbors. However, other researchers have found [30, 47, 75, 89] that modified clients can adopt strategic behavior (without violation of the core protocol) to defeat the incentive mechanism and

gain an unfair advantage. Piatek et al. argued [75] that the performance observed in BitTorrent is not the result of rational self-interest, but due to altruism and the fact that a majority of users are not knowledgeable enough to alter their BitTorrent clients. As a social experiment, they released *BitTyrant*, a strategic BitTorrent client. Their client results in significantly higher download rates than other peers with similar upload rates when used in a network with unmodified BitTorrent clients, but greatly degraded overall performance if large numbers of peers use their client.

7.4.2 Streaming Protocols

Sung et al. proposed a solution to the bandwidth allocation problem with their *Contribution-aware* protocol [88]. They used the Taxation system [15] proposed by Chu et al. to classify nodes in a forest of End System Multicast (ESM) [16] trees. Depending on their bandwidth contribution levels, nodes are classified as *entitled* in zero or more trees, while they are classified as *excess* nodes in the remaining trees. Entitled nodes may displace excess nodes to ensure that their demand for bandwidth is satisfied first. Moreover, entitled nodes may displace other entitled nodes with lower contribution levels. They show that their protocol succeeds in ensuring a much-improved level of performance for nodes with high contribution levels.

Their protocol has the same goal as ours. However, their system assumes that nodes are non-strategic and compute and honestly report their status as entitled or excess when participating in a tree, i.e., it is a *cooperative* system.

In their paper *Considering Altruism in Peer-to-Peer Internet Streaming* [45] Chu and Zhang proposed another interesting take on cooperative resource allocation for overlay multicast. Their proposal leverages the fact that some fraction of nodes are *altruistic*, i.e., they willingly contribute resources to the system. They use Multiple-Description Codes (MDC) to allow nodes with limited upload bandwidth to take advantage of the excess bandwidth provided by altruistic nodes.

BAR Gossip [53] is a gossip-based protocol that ensures pairwise fairness among participants in the network, and is resilient to rational and Byzantine attacks. However, their model is aimed at ensuring tit-for-tat fairness, and does not easily accommodate nodes with low upload capacities, even when there are enough nodes with high-upload rates to ensure that the system is resource-rich.

Ngan et al. suggested a reputation-based approach [69] that involved periodically rebuilding trees, and eliminating nodes that do not contribute bandwidth. It might be possible to modify their approach to accommodate ADSL nodes in resource-rich systems as we do, but the periodic rebuilding of trees required by their approach means that their system cannot be as responsive as ours without very frequent rebuilding of trees, which would result in a considerable control traffic overhead.

Give-to-get [66] is the incentive mechanism proposed by Mol et al. for their on-demand streaming adaptation of BitTorrent. This is a simple reputation system that ranks neighbors by how much data they forward to other nodes. They experimentally demonstrate that this protocol works well with up to 20% free-riders in the system. However, we believe that this scheme is weaker than the Token Stealing algorithm because nodes rely on information reported by neighbor of their neighbors (i.e., nodes that are exactly two hops away), thus increasing its vulnerability to misreporting and collusion.

Chapter 8

Conclusions

Peer-to-peer networks are an effective mechanism for large-scale distribution of content. This technique allows recipients to use their upload capacity to support other nodes, thus relieving the burden from the content provider. If every node in the system contributes as much bandwidth to the system as it consumes, the system can support a large number of nodes with a constant load on the original distributor (seed). Our thesis is that a robust incentive mechanism is necessary to encourage nodes to contribute resources to the system, and a receiver-driven architecture with a pairwise incentive mechanism allows for great flexibility, simplicity, robustness, and performance.

We presented SWIFT, an incentive mechanism for file sharing networks, and showed that nodes that contribute to the system receive significantly better performance than those that do not. This system promotes system scalability by preventing nodes from consuming more resources than they contribute to the system. This system serves as a foundation for our work on peer-to-peer streaming.

We presented Chainsaw, a peer-to-peer live streaming protocol based on an unstructured mesh network. Our protocol is more amenable to pairwise incentives than traditional tree-based mechanisms. We experimentally showed that this system supports high-bandwidth data dissemination with low packet-loss and low delay. In a typical network setup, we observed the system-wide mean packet loss to be 0.0005%, or about 1 in 200,000 packets with a mean delay of 1.8 seconds. We showed that the system scaled well with increasing network size, with packet loss rate remaining low even when we increased the network size by two orders of magnitude to over 100,000 nodes. We also showed that our system was highly resistant to churn. Even when a vast majority of nodes persisted in the network for extremely short periods of time, with a mean lifetime of only 25 seconds, packet loss rates stayed low, at 0.008% or 1 in 150,000 packets.

We then presented the Token Stealing algorithm, an incentive mechanism built on top of the Chainsaw streaming protocol. Our algorithm is simple, runs locally on each node and relies only on direct observation of a neighbor's upload rates. This makes the algorithm easy to implement, immune to misreported third-hand information, and requires no additional network resources.

In a resource-constrained system, we showed that the Token Stealing algorithm preferentially directs bandwidth towards altruistic nodes that contribute more upload bandwidth to the system than they consume. This preferential allocation gives the altruistic nodes significantly better performance than the ADSL nodes that contribute less bandwidth. In a simulation with 80% ADSL nodes and 20% altruistic nodes with a supply-to-demand ratio of 0.55, the ADSL nodes

had a packet loss rate of eight times higher than the altruistic nodes. This property gives nodes a concrete incentive to contribute more bandwidth to the system; nodes that artificially capped their upload bandwidth and masqueraded as ADSL nodes would likely remove the cap in order to reduce their packet loss rate. This action benefits the system as a whole in addition to benefiting the node itself.

We showed that an ADSL node that increased its upload rate to the level of the altruistic nodes started receiving reduced packet loss rates quickly, and had performance indistinguishable from other altruistic nodes after 30 seconds. Conversely, a formerly altruistic node that reduced its packet loss rate saw increased packet loss rates that converged to the same level as other ADSL nodes after 30 seconds. This implies that nodes cannot gain an unfair advantage by uploading for a while to gain a good reputation, and then reducing their upload rate. We also showed that nodes could not game the system by connecting selectively to high-bandwidth nodes.

We presented results from a prototype implementation with experiments conducted on the PlanetLab networking testbed. The systems in these experiments showed similar behavior to that of our simulation results, thus reinforcing our results.

Finally, we discussed various future research directions, including possible performance enhancements inspired by other research in the field, and ways to thwart possible attacks by rational, self-interested peers, as well as malicious peers. We also discussed the feasibility of unifying the file sharing, on-demand, and live streaming models to allow users to seamlessly switch between the different modes on a given stream, providing great flexibility, and enabling novel media experiences.

Chapter 9

Limitations and Future Work

In this chapter we discuss potential future directions. In Section 9.1, we discuss how selfish peers may attempt to defeat the Token Stealing incentive mechanism and gain an unfair advantage. In Section 9.2 we discuss ways in which malicious attackers may try to harm the system without regard to their own performance. In Section 9.3 we consider ideas from other related research that may be used to improve the performance of our system further. In Section 9.4 we discuss how micropayments might allow nodes with limited upload capacity to receive good performance by making small payments in exchange for data. Finally, in Section 9.5 we discuss how the Chainsaw and Token Stealing protocols may be extended to unify file sharing, on-demand streaming and live streaming.

9.1 Network and OS Level Gaming Attacks

The Token Stealing algorithm does not depend on information reported by other nodes, instead relying only on direct observation, thus making it immune to attacks that involve misreporting information. In Sections 6.13 and 6.15 we studied ways in which participants may attempt to game the system within the Chainsaw and Token Stealing frameworks by tweaking upload rates or connection parameters. We showed that it was very hard for selfish nodes to cheat the system by tweaking application-level parameters.

However, further research is needed to determine if lower level attacks may prove successful. Although an ADSL node cannot achieve an improvement in performance by selectively connecting to altruistic nodes in a resource-constrained system, a low-level attack might render this strategy more fruitful. For example, if the attacker were able to prevent other nodes from connecting to an altruistic node, it will result in a local surplus of upload capacity which the attacker could then exploit. However, denial of service attacks often require considerable amounts of bandwidth or other resources in order to overwhelm their targets. Further investigation is needed to determine if such an attack could be carried out with less resources than it would take to simply upload data to that peer in the first place.

Furthermore, and attack against an altruistic node that rendered it unable to communicate with other neighbors competing with the attacker for upload bandwidth will require a fine balance between eliminating the competitors for bandwidth and isolating the targeted altruistic node completely. If the attacker disconnected the altruistic node from every other neighbor, the

altruistic node would have no source of data to send to the attacker, rendering its spare upload capacity useless.

9.2 Malicious Attacks

In this dissertation, we have only considered the case of rational self-interested peers, i.e., peers who are primarily interested in receiving content from the network. Malicious peers, on the other hand, are peers who are not interested in receiving content but rather in disrupting the network and preventing other peers from receiving content. These are called Denial of Service (DoS) attacks. An attacker with significant resources could likely overwhelm an individual participant's Internet connection with a targeted attack, but this is not a vulnerability of our system per se. We are primarily interested in attacks which disrupt a large fraction of the network, or prevent large numbers of new participants from joining.

DoS attacks could be carried out against the membership server, seed, or against normal peers. We consider each of these in turn.

9.2.1 Attacks Against the Membership Server

In our current implementation, the membership server is a single point of failure. Without the membership server, nodes are unable to join the system or find new neighbors when old ones leave. However, as mentioned in Section 4.2.2, the membership server is not an essential part of our system, and the drawbacks with our current implementation could easily be mitigated. We could easily incorporate either multiple redundant membership servers or eliminate them entirely by using random walks for peer discovery, as is done with Gnutella [40].

9.2.2 Attacks Against the Seed

In our current implementation, we have all new packets generated by a single seed node. Disruption of this node will prevent any node in the system from receiving data. Such an attack would be analogous an attack against the server in a traditional client-server system. However, we believe attacks against the seed can be mitigated in several ways.

Firstly, our system does not require just a single seed node. Data can be injected simultaneously by multiple seeds so long as all packets with a given sequence number are identical, and packets that comprise the stream are numbered sequentially. The stream could be communicated to multiple distributed seeds via a private network (which can itself be a separate Chainsaw network), and each seed could inject some or all of the packets into the network. Having multiple redundant seeds will make it harder for an attacker to disrupt the seed, and also safeguard against network or hardware failures.

Secondly, it is difficult for an attacker to even determine the identity of the seed. From the point of view of its peers, the seed behaves exactly the same as any other node. It is difficult to determine whether data is being generated by a node or if it is simply being forwarded. Even a map of the complete network will not reveal the seed unless the attacker is also able to trace the complete path of a data packet to its source, which is a formidable challenge.

Finally, since TCP connections are bidirectional, the seed can be protected behind a firewall and make outgoing connections to other peers, while refusing all incoming connections. This arrangement can guard the seed against many network and transport-layer attacks

9.2.3 Attacks Against Other Peers

It is difficult for attacks against a small number of non-seed peers to cause wide-scale disruption of the network, because unlike tree-based systems, there are many redundant paths from the seed to each node. The loss of any given peer has very little impact on the network at large, since a large fraction of edges must be disconnected to partition a well-connected random graph [43].

Section 6.7 have shown that the system is robust to high levels of churn. Therefore, the disconnection of a small number of nodes will not have an adverse effect on the network as a whole, and will quickly recover from minor local disruptions.

9.3 Performance Improvements

While we have demonstrated that the Chainsaw streaming protocol and Token Stealing incentive mechanism offer excellent performance over a wide range of parameters, there remains room for improvement

9.3.1 Network Topology

One of the key properties of our system is that the network topology, packet routing decisions, and the incentive mechanism are decoupled from each other. Therefore, we may be able to improve the performance or robustness of our system further by incorporating techniques from other related research. For example, the buffer and request management system proposed by Mol at al. [66] might help reduce the delay and drive packet loss rates further down by giving higher priority to packets that are needed sooner (i.e., packets of lower sequence number).

In resource-constrained systems, we might be able to reduce the packet loss rates of altruistic nodes closer to zero, and that of the ADSL nodes closer to the theoretical ideal by taking resource constraints and geographical factors into consideration while building the network [1, 8, 14, 88].

9.3.2 Initial Packet Loss

In Section 6.11.3 we showed that when nodes join a resource-constrained system it takes a few tens of seconds for them to ramp up their bandwidth and earn credit by uploading data to their neighbors. A major constraint is the fact that even if it is willing to do so, a node may not have data to upload to its neighbors at first.

We might be able to shorten the ramp-up period by allowing newly connected nodes to demonstrate altruistic intent by uploading an equivalent number of data in null or random bytes. This would only be permitted for a few seconds after a connection is established in order to minimize the amount of useless traffic generated.

9.3.3 Reducing Overhead

In the Chainsaw protocol, nodes must notify their neighbors of available packets so that they may request them. In our implementation this is done naively by sending notification messages on receiving a new packet. Although we have shown that this does not lead to an excessive level of overhead—on the order of 10% in most cases—the overhead bandwidth is problematic for some setups, such as with very small packets.

We might be able to improve the range of parameters over which the packet loss rates remain low by sending notifications more efficiently. For instance, notification messages may by sent in batches, rather than as individual messages in order to reduce the overhead generated per packet notification. Furthermore, since the list of packets will often be a contiguous run of sequence numbers, further savings may be achieved by transmitting differences rather than full sequence numbers, and other data compression techniques. Such improvements will require a careful balance between the overhead savings and the additional delay caused by delaying notification messages in order to send batch updates.

9.4 Micropayments as an Alternative to Uploading

We have shown that our incentive mechanism is effective at ensuring that nodes that upload data at less than the stream rate receive lower performance in return when the system is resource-constrained. This means that in effect, participants may often find themselves constrained by their *upload* capacity rather than download bandwidth. For example, a node with a 512 kbps up/6 Mbps down connection may be unable to join a 2Mbps stream and be forced to settle for a lower quality, say, 320 kbps stream.

Users who are constrained in this way by the physical limitations of their connections might be willing to pay to receive better performance. Our Token Stealing algorithm is a pairwise mechanism, which is amenable to incorporating a micropayment mechanism, as neighbors may conduct currency transactions with each other in addition to the normal data traffic. In addition to obtaining private bucket credits the usual way by uploading data, a node can purchase credits by sending a cryptographic token as a proof of payment. MuCash [67], a company founded in 2010 by the author of this dissertation is building a micropayment platform that would be well suited to this application.

On receiving proof of payment, the neighbor would transfer credits into the node's private bucket just as if it had uploaded a packet of data to it. This would allow nodes with surplus upload capacity to profit from it by offering data to other nodes that are unable to upload data at the stream rate and would ordinarily suffer from high packet loss rates. The pairwise nature of the system makes it easy for individual nodes to set their own prices and policy.

9.5 Transition Between File Sharing and Streaming

Our Chainsaw protocol is receiver-driven, i.e., the receiver decides which packets to receive. Similar protocols have been used for both file sharing [19, 89] and on-demand streaming [66]; those systems differ primarily in the buffer-management algorithms used internally by the nodes. Therefore, it might be possible to unify all three applications.

Consider a user who joins a broadcast of a sporting event after the program has already begun. The user could choose to begin receiving the very latest data, thus viewing the live coverage but missing out on the previous part of the game. Alternatively, the user may choose to watch from the beginning, as in on-demand media. If desired, the user could then jump forward periodically, catching the highlights of the earlier coverage and eventually joining the live stream. Finally, users with insufficient bandwidth to watch the stream live without packet loss (i.e., ADSL nodes) could have the option of either watching the coverage at lower quality or switching to *download mode*, and downloading the high-quality media at slower than real-time speed for later viewing. In fact, the application could even allow the user the view a low-quality version live, but fill in the missing packets later to create a high quality archive.

Two key challenges need to be addressed to enable this level of flexibility. Firstly, nodes in the Chainsaw streaming protocol rely on the fact that all neighbors are downloading packets within the same window. One possible solution would be to modify the network construction protocol to enable nodes to advertise the range of packets they are interested in, and peer with the appropriate set of neighbors interested in the same range. Secondly, as we have argued in Section 5.2.3, different incentive mechanisms are needed for live streaming and file sharing applications. Further research is needed to reconcile these differences.

Bibliography

[1] Lada A. Adamic, Rajan M. Lukose, Amit R. Puniyani, and Bernardo A. Huberman. Search in power-law networks. *Phys. Rev. E*, 64(4):046135, Sep 2001.

[2] E. Adar and B.A. Huberman. Free Riding on Gnutella. *First Monday*, 5(10), Oct 2000.

[3] E. Altman, K. Avrachenkov, and B. Prabhu. Fairness in mimd congestion control algorithms. In *Procceedings of the IEEE INFOCOM 2005*, 2005.

[4] AT&T. At&t expands new-generation ip/mpls backbone network. http://www.att.com/gen/press-room?pid=4800&cdvn=news&newsarticleid=24888.

[5] R. Axelrod. *The Evolution of Cooperation*. Basic Books, 1984.

[6] Suman Banerjee, Seungjoon Lee, Bobby Bhattacharjee, and Aravind Srinivasan. Resilient multicast using overlays. In *SIGMETRICS '03: Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 102–113, New York, NY, USA, 2003. ACM.

[7] T. C. Bergstrom. 'A fresh look at the rotten kid theorem and other household mysteries'. *Journal of Political Economy*, 1989.

[8] Bartosz Biskupski, Raymond Cunningham, Jim Dowling, and René Meier. High-bandwidth mesh-based overlay multicast in heterogeneous environments. In *AAA-IDEA '06: Proceedings of the 2nd international workshop on Advanced architectures and algorithms for internet delivery and applications*, page 4, New York, NY, USA, 2006. ACM.

[9] John W. Byers, Michael Luby, Michael Mitzenmacher, and Ashutosh Rege. A digital fountain approach to reliable distribution of bulk data. In *SIGCOMM '98: Proceedings of the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 56–67, New York, NY, USA, 1998. ACM.

[10] John W. Byers, Michael Luby, Michael Mitzenmacher, and Ashutosh Rege. A digital fountain approach to reliable distribution of bulk data. In *SIGCOMM*, pages 56–67, 1998.

[11] M. Castro, P. Druschel, A. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. Splitstream: High-Bandwidth Multicast in Cooperative Environments. In *Proceedings of the 2003 Symposium on Operating System Principles*, 2003.

[12] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in communications (JSAC)*, 20(8):1489–1499, 2002.

[13] Yatin Chawathe. Scattercast: an adaptable broadcast distribution framework. *Multimedia Systems*, 9(1):104–118, July 2003.

[14] Yatin Chawathe, Sylvia Ratnasamy, Lee Breslau, Nick Lanham, and Scott Shenker. Making gnutella-like p2p systems scalable. In *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 407–418, New York, NY, USA, 2003. ACM.

[15] Y. Chu, J. Chuang, and H. Zhang. A case for taxation in peer-to-peer streaming broadcast. In *Proceedings of the ACM SIGCOMM workshop on Practice and theory of incentives in networked systems*, 2004.

[16] Y. Chu, S. G. Rao, and H. Zhang. A case for end system multicast. In *Measurement and Modeling of Computer Systems*, 2000.

[17] Yang Chu, Sanjay Rao, Srinivasan Seshan, and Hui Zhang. Enabling conferencing applications on the internet using an overlay muilticast architecture. *SIGCOMM Comput. Commun. Rev.*, 31(4):55–67, 2001.

[18] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. Planetlab: an overlay testbed for broad-coverage services. *SIGCOMM Computer Communication Review*, 2003.

[19] B. Cohen. BitTorrent, 2001. http://www.bitconjurer.org/BitTorrent/.

[20] B. Cohen. Incentives build robustness in BitTorrent. *Workshop on Economics of Peer-to-Peer Systems*, Jun 2003.

[21] comScore, Inc. comScore releases august 2010 U.S. online video rankings. comScore.com, August 2010. `http://www.comscore.com/Press_Events/Press_Releases/2010/9/comScore_Releases_August_2010_U.S._Online_Video_Rankings`.

[22] Stephen E. Deering. Multicast routing in internetworks and extended lans. In *SIGCOMM*, pages 55–64, 1988.

[23] Brian Dessent. Brian's bittorrent FAQ & guide. `http://www.btfaq.org`, 2003.

[24] J. Douceur. The sybil attack, 2002.

[25] J.R. Douceur. The Sybil attack. *International Workshop on Peer-to-Peer System(IPTPS'02)*, Mar 2002.

[26] P.B.J. Duijkers. Performance analysis of chainsaw-based live p2p video streaming. Master's thesis, Delft University of Technology, Delft, The Netherlands, December 2008.

[27] Derek L. Eager, Mary K. Vernon, and John Zahorjan. Minimizing bandwidth requirements for on-demand data delivery. *Knowledge and Data Engineering*, 13(5):742–757, 2001.

[28] P. T. Eugster, R. Guerraoui, A. M. Kermarrec, and L. Massoulie. Epidemic information dissemination in distributed systems. *Computer*, 37(5):60–67, May 2004.

[29] P. T. Eugster, R. Guerraoui, A. m. Kermarrec, and L. Massouli. From epidemics to distributed computing. *IEEE Computer*, 37:60–67, 2004.

[30] Bin Fan, Dah ming Chiu, and John Lui. The delicate tradeoffs in bittorrent-like file sharing protocol design. In *ICNP '06: Proceedings of the Proceedings of the 2006 IEEE International Conference on Network Protocols*, pages 239–248, Washington, DC, USA, 2006. IEEE Computer Society.

[31] Shawn Fanning. Napster. http://www.napster.com/.

[32] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFCs 2817, 5785.

[33] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, 1993.

[34] Chuck Fraleigh, Sue Moon, Bryan Lyles, Chase Cotton, Mujahid Khan, Deb Moll, Rob Rockell, Ted Seely, and Christophe Diot. Packet-level traffic measurements from the sprint ip backbone. *IEEE Network*, 17:6–16, 2003.

[35] Paul Francis. Yoid: Extending the internet multicast architecture. Technical report, AT&T Center for Internet Research at ICSI (ACIRI), 2000.

[36] Michael J. Freedman, Eric Freudenthal, and David Mazières. Democratizing content publication with coral. In *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, pages 18–18, Berkeley, CA, USA, 2004. USENIX Association.

[37] E. Frost. Postmortem [RedHat 9 Binary ISO download on BitTorrent]. http://f.scarywater.net/postmortem/.

[38] Didier Le Gall. Mpeg: A video compression standard for multimedia applications. *Communications of the ACM*, 34:46–58, 1991.

[39] Lawrence Gasman. Oc-768 and beyond - technology information. *Telecommunications*, July 2000.

[40] Gnutella. http://rfc-gnutella.sourceforge.net/.

[41] P. Golle, K. Leyton-Brown, I. Minronov, and M. Lillibridge. Incentives for Sharing in Peer-to-Peer Networks. *Proceedings 3rd ACM Conference on Electronic Commerce (EC-2001)*, Oct 2001.

[42] David A. Helder and Sugih Jamin. End-host multicast communication using switch-trees protocols. In *CCGRID '02: Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, page 419, Washington, DC, USA, 2002. IEEE Computer Society.

[43] Shlomo Hoory, Nathan Linial, Avi Wigderson, and An Overview. Expander graphs and their applications. *Bulletin of the Americal Mathematical Society*, 43(4):439–561, October 2006.

[44] John Horrigan. Home broadband adoption 2009. Technical report, Pew Research Center, June 2009.

[45] Yang hua Chu and Hui Zhang. Considering altruism in peer-to-peer internet streaming broadcast. In *NOSSDAV '04: Proceedings of the 14th international workshop on Network and operating systems support for digital audio and video*, pages 10–15, New York, NY, USA, 2004. ACM.

[46] Seung Jun and Mustaque Ahamad. Incentives in bittorrent induce free riding. In *Proceeding of the 2005 ACM SIGCOMM workshop on Economics of peer-to-peer systems*, pages 116–121, New York, NY, USA, 2005. ACM Press.

[47] Seung Jun and Mustaque Ahamad. Incentives in bittorrent induce free riding. In *P2PECON '05: Proceedings of the 2005 ACM SIGCOMM workshop on Economics of peer-to-peer systems*, pages 116–121, New York, NY, USA, 2005. ACM.

[48] Sepandar D. Kamvar, Mario T. Schlosser, and Hector Garcia-molina. The eigentrust algorithm for reputation management in p2p networks. In *In Proceedings of the 12th International World Wide Web Conference (WWW 2003*, 2003.

[49] O. Kim and M. Walker. The free rider problem: Experimental evidence. *Public Choice*, 1984.

[50] D. Kostić, A. Rodriguez, J. Albrecht, A. Bhirud, and A. Vahdat. Using random subsets to build scalable network services. In *Proceedings of the 2003 USENIX Symposia on Internet Technologies and Systems*, 2003.

[51] D. Kostić, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: high bandwidth data dissemination using an overlay mesh. In *Proceedings of the 2003 Symposium on Operating System Principles*, 2003.

[52] Arnaud Legout, G. Urvoy-Keller, and P. Michiardi. Rarest first and choke algorithms are enough. In *IMC '06: Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 203–216, New York, NY, USA, 2006. ACM.

[53] H. Li, A. Clement, E. Wong, J. Napper, I. Roy, L. Alvisi, and M. Dahlin. BAR gossip. In *Proceedings of the 2006 USENIX Operating Systems Design and Implementation (OSDI)*, November 2006.

[54] Weiping Li. Overview of fine granularity scalability in mpeg-4 video standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 11(3):301–317, March 2001.

[55] J. Liebeherr and M. Nahas. Application-layer multicast with delaunay triangulations. In *Global Telecommunications Conference, 2001. GLOBECOM '01. IEEE*, volume 3, pages 1651–1655 vol.3, 2001.

[56] John C. Lin and Sanjoy Paul. Rmtp: A reliable multicast transport protocol. In *IEEE Journal on Selected Areas in Communications*, pages 1414–1424, 1996.

[57] Jiangchuan Liu, Bo Li, and Ya-Qin Zhang. Adaptive video multicast over the internet. *IEEE MultiMedia*, 10(1):22–33, 2003.

[58] Qin Lv, Sylvia Ratnasamy, and Scott Shenker. Can heterogeneity make gnutella scalable. In *In Proceedings of the first International Workshop on Peer-to-Peer Systems*, pages 94–103, 2002.

[59] Nazanin Magharei, Reza Rejaie, and Yang Guo. Mesh or Multiple-Tree: A Comparative Study of P2P Live streaming. In *Proceedings of IEEE Infocom*, 2007.

[60] Farhad Manjoo. Do you think bandwidth grows on trees? Slate.com, April 2009. http://www.slate.com/id/2216162.

[61] G. Marwell and R. Ames. Experiments in the provision of public goods: I. resources, interest, group size, and the free-rider problem. *American Journal of Sociology*, 1979.

[62] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *1st International Peer-to-Peer Symposium (IPTPS 2002)*, pages 53–65, 2002.

[63] Steven McCanne, Van Jacobson, and Martin Vetterli. Receiver-driven layered multicast. In *ACM SIGCOMM*, volume 26,4, pages 117–130, New York, August 1996. ACM Press.

[64] R. C. Merkle. Protocols for public key cryptography. *Proceedings of the IEEE Symposium on Security and Privacy*, Apr 1980.

[65] Alexander Mohr, Eve Riskin, , Richard Ladner, and Richard E. Ladner. Unequal loss protection: Graceful degradation of image quality over packet erasure channels through forward error correction. *IEEE Journal on Selected Areas in Communications*, 18:819–828, 1999.

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

[67] MuCash, Inc. Mucash micropayments platform. http://www.mucash.com/.

[68] Sharman Networks. Kazaa. http://www.kazaa.com/.

[69] T. Ngan, D. S. Wallach, and P. Druschel. Incentives-compatible Peer-to-Peer Multicast. In *Proceedings of the Second Workshop on the Economics of Peer-to-Peer Systems*, 2004.

[70] National Institute of Standards and Technology. FIPS 140-2; security requirements for cryptographic modules, November 2002.

[71] Venkata N. Padmanabhan and Kunwadee Sripanidkulchai. The case for cooperative networking. In *Peer-to-Peer Systems: First International Workshop, IPTPS 2002*, pages 178–190, Cambridge, MA, USA, March 2002.

[72] Vinay Pai, Kapil Kumar, Karthik Tamilmani, Vinay Sambamurthy, and Alexander E. Mohr. Chainsaw: Eliminating trees from overlay multicast. In *in Proc. The 4th International Workshop on Peer-to-Peer Systems (IPTPS*, pages 127–140, 2005.

[73] Vinay Pai and Alexander E. Mohr. Improving robustness of peer-to-peer streaming with incentives. In *Proceedings of the 1st Workshop on the Economics of Networks*, 2006.

[74] H. E. Peters, A. S. Unur, J. Clark, and W. D. Schulze. Free-Riding and the Provison of Public Goods in the Family: A Laboratory Experiment. *International Economic Review*, 45:283–299, Feb 2004.

[75] Michael Piatek, Tomas Isdal, Thomas Anderson, Arvind Krishnamurthy, and Arun Venkataramani. Do incentives build robustness in bittorrent? In *Proceedings of the 2007 Symposium on Networked Systems Design and Implementation*, Cambridge, MA, April 2007.

[76] Sridhar Pingali, Don Towsley, and James F. Kurose. A comparison of sender-initiated and receiver-initiated reliable multicast protocols. In *IEEE Journal on Selected Areas in Communications*, pages 221–230, 1994.

[77] J. Postel and J. Reynolds. File Transfer Protocol. RFC 959 (Standard), October 1985. Updated by RFCs 2228, 2640, 2773, 3659, 5797.

[78] Dongyu Qiu and R. Srikant. Modeling and performance analysis of bittorrent-like peer-to-peer networks. In *SIGCOMM '04: Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 367–378, New York, NY, USA, 2004. ACM.

[79] Sanjay G. Rao, Kunwadee Sripanidkulchai, Jibin Zhan, and Hui Zhang. Early experience with n internet broadcast system based on overlay multicast. In *in Proceedings of USENIX*, 2004.

[80] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, volume 31, pages 161–172. ACM Press, October 2001.

[81] R. Rejaie, M. Handley, and D. Estrin. Layered quality adaptation for internet video streaming, 2000.

[82] 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.

[83] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to end arguments in system design. *Innovations in Internetworking*, pages 195–206, 1988.

[84] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. *Proceedings of Multimedia Computing and Networking*, Jan 2002.

[85] C. Shannon and D. Moore. The Spread of the Witty Worm. http://www.caida.org/analysis/security/witty/, Mar 2004.

[86] Kunwadee Sripanidkulchai, Aditya Ganjam, Bruce Maggs, and Hui Zhang. The feasibility of supporting large-scale live streaming applications with dynamic application end-points. *SIGCOMM Comput. Commun. Rev.*, 34(4):107–120, 2004.

[87] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160, New York, NY, USA, 2001. ACM.

[88] Yu-Wei Sung, Michael Bishop, and Sanjay Rao. Enabling contribution awareness in an overlay broadcasting system. *SIGCOMM Computer Communications Review*, 36(4):411–422, 2006.

[89] Karthik Tamilmani, Vinay Pai, and Alexander E. Mohr. Incentives-compatible Peer-to-Peer Multicast. In *Proceedings of the Second Workshop on the Economics of Peer-to-Peer Systems*, 2004.

[90] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall PTR, fourth edition, 2003.

[91] Akamai Technologies. Akamai. http://www.akamai.com/.

[92] The OpenSSL Project. OpenSSL: The open source toolkit for SSL/TLS. `http://www.openssl.org`, April 2003.

[93] Iain Thomson. New BitTorrent Flows Across the Web, 2005. http://www.itweek.co.uk/vnunet/news/2126947/bittorrent-flows-across-web.

[94] Vidhyashankar Venkataraman, Paul Francis, and John Cal. Chunkyspread: Multi-tree unstructured peer-to-peer multicast. In *in Proc. The 5th International Workshop on Peer-to-Peer Systems*, 2006.

[95] 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.

[96] Limin Wang, Kyoung Soo Park, Ruoming Pang, Vivek Pai, and Larry Peterson. Reliability and security in the codeen content distribution network. In *ATEC '04: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 14–14, Berkeley, CA, USA, 2004. USENIX Association.

[97] Stephen B. Wicker. *Reed-Solomon Codes and Their Applications*. IEEE Press, Piscataway, NJ, USA, 1994.

[98] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra. Overview of the h.264/avc video coding standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):560–576, July 2003.

[99] Jin wook Seo, Dong kyun Kim, Hyun chul Kim, and Jin wook Chung. The algorithm of sharing incomplete data in decentralized p2p. *International Journal of Computer Science and Network Security*, 7(8), August 2007.

[100] X. Xu, A. Myers, H. Zhang, and R. Yavatkar. Resilient multicast support for continuous-media applications, 1997.

[101] Tingting Zhang and Youshi Xu. Unequal packet loss protection for layered video transmission. *IEEE Transactions of Broadcasting*, 45:243–252, 1999.

[102] Xinyan Zhang, Jiangchuan Liu, Bo Li, and Y. S. P. Yum. Coolstreaming/DONet: a data-driven overlay network for peer-to-peer live media streaming. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, volume 3, pages 2102–2111 vol. 3, 2005. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1498486.

[103] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, January 2004.

[104] Shelley Q. Zhuang, Ben Y. Zhao, Anthony D. Joseph, Randy H. Katz, and John D. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *Proceedings of NOSSDAV*, June 2001.