RICE UNIVERSITY

Providing Incentives to Peer-to-Peer Applications

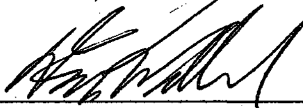
by

Tsuen Wan "Johnny" Ngan

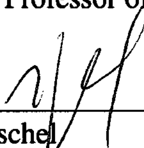
A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

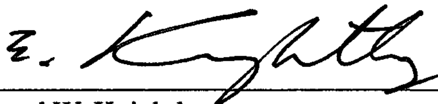
APPROVED, THESIS COMMITTEE:



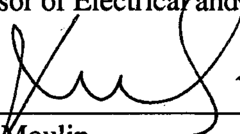
Dan S. Wallach, Chair
Associate Professor of Computer Science



Peter Druschel
Scientific Director, Max Planck Institute for Software Systems



Edward W. Knightly
Professor of Electrical and Computer Engineering



Hervé Moulin
George A. Peterkin Professor of Economics

T. S. Eugene Ng
Assistant Professor of Computer Science

HOUSTON, TEXAS
AUGUST 2008

UMI Number: 3362376

INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.
In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

UMI®

UMI Microform 3362376
Copyright 2009 by ProQuest LLC
All rights reserved. This microform edition is protected against unauthorized copying under Title 17, United States Code.

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

Abstract

Cooperative peer-to-peer applications are designed to share the resources of participating computers for the common good of all users. However, users do not necessarily have an incentive to donate resources to the system if they can use the system's resources for free. As commonly observed in deployed applications, this situation adversely affects the applications' performance and sometimes even their availability and usability.

While traditional resource management is handled by a centralized enforcement entity, adopting similar solution raises new concerns for distributed peer-to-peer systems. This dissertation proposes to solve the incentive problem in peer-to-peer applications by designing fair sharing policies and enforcing these policies in a distributed manner. The feasibility and practicability of this approach is demonstrated through numerous applications, namely archival storage systems, streaming systems, content distribution systems, and anonymous communication systems.

Contents

1 Introduction 1
- 1.1 Peer-to-peer Systems 1
- 1.2 Problem Statement 2
- 1.3 Previous Approaches and Relevant Works 4
  - 1.3.1 Discourage abuse of service 4
  - 1.3.2 Tit-for-tat 5
  - 1.3.3 Electronic currency 6
  - 1.3.4 Reputation 6
  - 1.3.5 Byzantine faults 7
  - 1.3.6 Game theory 9
  - 1.3.7 Mechanism design 9
  - 1.3.8 Economic analysis 10
- 1.4 Contribution of this Thesis 11
- 1.5 Guide to the Thesis 13

2 Background and Model 15
- 2.1 Peer-to-peer Networks 15
  - 2.1.1 Unstructured and structured networks 16
  - 2.1.2 Distributed hash table (DHT) 17
  - 2.1.3 Advantages 18
  - 2.1.4 An example: Pastry 18
- 2.2 Attacks and Threat Model 19
  - 2.2.1 Malicious attacks 19
  - 2.2.2 Freeloading "attacks" 22
  - 2.2.3 Threat model 23

3 Archival Storage Systems 24
- 3.1 Introduction 24
- 3.2 Ensuring Honest Storage 25
- 3.3 Design 26
  - 3.3.1 Usage files 26
  - 3.3.2 Attacks and audits 27
  - 3.3.3 Cheating patterns 30
  - 3.3.4 Expelling a misbehaving node 32
  - 3.3.5 Preventing content distribution abuse 32
  - 3.3.6 Handling churn 33
  - 3.3.7 Extensions 33

3.4 Experiments 35
3.5 Summary 37
3.6 Related Work 37

4 Streaming Systems **39**
4.1 Introduction 39
4.2 System Model 40
4.2.1 FairStream basics 40
4.2.2 Data and path authenticity 42
4.3 FairStream Design 44
4.3.1 Overview 44
4.3.2 Periodic tree reconstruction 45
4.3.3 Collecting information 45
4.3.4 Reciprocal requests 47
4.3.5 Selective servicing 48
4.4 Experimental Evaluation 48
4.4.1 Modification from SplitStream 49
4.4.2 Experimental setup 49
4.4.3 Tree reconstruction cost 50
4.4.4 Fraction of freeloaders 51
4.4.5 SplitStream properties 52
4.4.6 Enforcing FairStream policies 53
4.5 Summary 55
4.6 Related Work 56

5 Content Distribution Systems **57**
5.1 Introduction 57
5.2 Background: BitTorrent 58
5.3 Goals and System Model 59
5.3.1 Goals 60
5.4 Design 60
5.4.1 Relationships 60
5.4.2 Confidence 62
5.4.3 Transitive trade 63
5.4.4 Caching 67
5.5 Implementation 68
5.5.1 Node bootstrapping 68
5.5.2 Finding credit paths 69
5.5.3 Bounding lengths of credit paths 71
5.6 Experimental Results 72
5.6.1 Workload model 73
5.6.2 System performance 74
5.7 Summary 85

5.8 Related Work 85

6 Anonymous Communication Systems **88**
6.1 Introduction 88
6.2 Background 90
6.3 Design 91
6.3.1 Design space 91
6.3.2 The proposed design 95
6.4 Experiments 98
6.4.1 Experimental apparatus 98
6.4.2 Experiment 1: Unincentivized Tor 101
6.4.3 Experiment 2: Score measurements 103
6.4.4 Experiment 3: Gold stars 105
6.4.5 Experiment 4: Pair-wise reputation 110
6.5 Discussion 111
6.5.1 Strategic users 111
6.5.2 The audit arms race 112
6.6 Summary 113
6.7 Related Work 114

7 Design Discussions **116**
7.1 Pairwise Exchanges 116
7.2 Reputation 117
7.3 General Primitives 118
7.4 Other Design Choices 119
7.5 Common Design Principles 120

8 Future Work **123**
8.1 Conclusions 124

List of Figures

3.1 A peer-to-peer system with nodes showing their local and remote lists. . . . 27
3.2 A cheating chain. 29
3.3 Overhead with different number of nodes. 35
3.4 Overhead with different number of files stored per node. 36
3.5 Overhead with different average node lifetime. 36

4.1 A hash chain. 42
4.2 Average tree reconstruction cost for one tree. 50
4.3 Fraction of multicast data received by obedient nodes with different types of freeloaders. 51
4.4 Cumulative distribution of negative confidence for obedient nodes and freeloaders. 52
4.5 Cumulative distribution of parental availability with only obedient nodes. . 53
4.6 Fraction of multicast streams successfully received when applying FairStream mechanisms. 54
4.7 Fraction of multicast data received by obedient nodes with different fraction of freeloaders after applying my mechanisms. 55

5.1 How a node can use a credit path to leverage a chain of credit to obtain content directly from a non-neighbor node. 64
5.2 The stages in the transitive trade protocol. 64
5.3 Success rate with only obedient nodes. 74
5.4 Cumulative distribution of the number of retries to find a debt-based path. . 75
5.5 Cumulative distribution of debt-based path lengths for different system sizes. 75
5.6 Success rate with 5% freeloaders that do not serve objects. 78
5.7 Success rate with 50% freeloaders that do not serve objects. 78
5.8 Success rate with a higher churn rate. 79
5.9 Success rate with the worst-case scenario where every obedient node gives a high initial confidence to all freeloaders. 80
5.10 Success rate with freeloaders that participates in transitive trades but do not fetch objects for the first 20 time units. 81
5.11 Success rate with freeloaders that serve objects only for the first 20 time units. 82
5.12 Number of objects served and fetched with freeloaders that serve half of the object requests. 82
5.13 Success rate and number of objects served and fetched with freeloaders that aim at 50% success rate. 83

5.14 Success rate and number of objects served and fetched with freeloaders that switch between cooperation and freeloading every 20 time units. 84

6.1 Average download and ping time over time when no incentive scheme is in place. 101
6.2 Cumulative download and ping time when no incentive scheme is in place. . 102
6.3 The average scores measured for different relays. 103
6.4 The average scores measured when there are alternating relays. 105
6.5 Cumulative download and ping time with the gold star scheme and no background traffic. 106
6.6 Cumulative download and ping time with the gold star scheme and light background traffic. 106
6.7 Cumulative download and ping time with the gold star scheme and heavy background traffic. 106
6.8 Cumulative download and ping time with the gold star scheme, cooperative reserve relays, and no background traffic. 108
6.9 Cumulative download and ping time with the gold star scheme, cooperative reserve relays, and light background traffic. 108
6.10 Cumulative download and ping time with the gold star scheme, cooperative reserve relays, and heavy background traffic. 108
6.11 Average download and ping time with relays that alternate between being cooperative and selfish. 109
6.12 Cumulative download and ping time with the pair-wise reputation design and heavy traffic. 110

Chapter 1

Introduction

To the general public, p2p may be a synonym for unauthorized distribution of copyrighted music and movies. This unfortunate association is primarily due to the popularity of many file-sharing applications, such as the early versions of Napster, Kazaa [Kaz], and, more recently, BitTorrent [Coh03], that make use of *peer-to-peer* technology. Peer-to-peer technology is not limited to sharing files; in fact, it is merely a series of networked computers, comprising of individual users' computing resources, to run distributed applications.

1.1 Peer-to-peer Systems

Peer-to-peer, or more precisely peer-to-peer computer networking, is a paradigm for connecting computers in a decentralized fashion to build scalable systems. Unlike the more common client-server paradigm, in peer-to-peer networks the application service is not provided by dedicated servers. Instead, each participant, or *peer*, contributes some of their resources to collectively provide the service. Ideally, peer-to-peer systems can aggregate resources of all peers to provide service with very high levels of availability, reliability, scalability, and autonomy. With this promising upside, large number of co-

operative peer-to-peer systems have been proposed and developed recently, providing a general-purpose network substrate [MM02, RD01a, RFH+01, SMK+01, ZHS+04] suitable for sharing files [DKK+01, DR01, MMGC02], storage space [CMN02, RD01b], and bandwidth [BBK02, CDK+03, CDKR02, CRZ02], and for anonymous communication [DDM03, DMS04, FM02], among other applications.

1.2 Problem Statement

Peer-to-peer applications are usually designed to work in open networks, where any computer can freely join and leave at any time. Early applications typically assumed that most users are cooperative and are willing to altruistically contribute resources to the network. In practice, however, with the widespread of applications such as Napster, Gnutella, and Kazaa (see, e.g., Oram [Ora01] for an overview of these systems), it has been observed that in peer-to-peer systems, many users choose to consume the resources of other users without providing any of their own resources in return. These self-interested users are often referred to as "*free riders*" or "*freeloaders*." The study on Gnutella by Adar and Huberman [AH00] in 2000, for example, showed that upwards of 70% of the users enjoy the benefit of the system without contributing to its content, and nearly half of all service responses are performed by the top 1% of the users. Similar uncooperative behavior had been observed by Saroiu et al. [SGG02]. A later study in 2005 found freeloaders have further increased to 85% of all Gnutella users [HCW05]. This uneven service provision clearly shows that if non-cooperative behavior has no downside, most users will not con-

tribute their resources, and only a very small fraction of potential resources would become available. User selfishness significantly limits the scalability and performance of the resulting systems. If insufficient number of users contribute, in the extreme, these systems may even fail to function properly. This conflict over resources between individual interests and the common good leads to a classic "tragedy of the commons" situation [Har68].

The inherent cause of this conflict is that users have no natural *incentive* to provide services to their peers if it is not somehow required of them. Participants in peer-to-peer systems, after all, are human users with their own agendas. While some people may contribute because of the software's default settings, negligence, altruism, or other reasons, most people join simply because they expect to benefit from these systems by receiving services. Instead of sharing, they may prefer to reserve their resources for their own consumption or they may worry about unfavorable consequence of sharing (e.g., legal consequence of sharing copyrighted materials or monetary costs imposed by some Internet Service Providers when usage goes beyond certain bandwidth threshold). Therefore, to operate open peer-to-peer systems to their full potential, system designers have to pay attention to ensuring fair sharing of resources.

Resource management in distributed systems is not a new problem. Common solutions are typically centralized, where some designated trusted authority would give a user "permission" to consume resources. For example, in a distributed file system of a campus network, an administrator may assign a disk usage quota to each user. However, such notions are hard to create in a network of decentralized, ungoverned peers. Why should some

peers be placed in a position of authority over others? What if they abuse their authoritative power? Furthermore, how could the central authority ever scale to support millions of concurrent users, and provide services efficiently and reliably? Are there alternatives to this centralized design?

1.3 Previous Approaches and Relevant Works

Many approaches have been proposed to study and solve the incentive problem in distributed systems. This section serves as a brief survey of these approaches and other relevant works. Further discussions and comparisons to the mechanisms proposed in this dissertation can be found in Chapter 7.

1.3.1 Discourage abuse of service

Resource consumption in peer-to-peer systems are typically "free" (i.e., *gratis*) in the sense that users do not have to pay for any service they receive, even though the service itself has value. To discourage abuse, one can associate a cost with the service. Dwork and Naor [DN92] suggested to require users to compute a moderately hard, but not intractable, function in order to gain access to the resource. It has been implemented as a countermeasure for spam and denial-of-service attacks [Bac02]. Abadi et al., noting the sharp disparities in computational power across computer systems, proposed the use of memory-bound functions instead [ABMW05].

While these methods may be successful in limiting the service received by freeloaders,

it is a complete waste of resources — these mathematical puzzles only serve as a deterrence to use of service and solving them do not directly benefit anyone.

1.3.2 Tit-for-tat

A straightforward way to ensure fairness is to resort to barter economy, where two or more parties simultaneously exchange service with each other. Tangler [WM01], a censorship-resistant publication system, maintains fairness by requiring new servers to provide storage space for a period of time before they are allowed to publish. For file-sharing, BitTorrent [Coh03] allows peers to exchange pieces of a large object that everyone wants, but would "choke" (i.e., stop uploading to) peers that do not return the favor, effectively encouraging peers to exchange uploading slots. In spite of this design intent, Piatek et al. [PIA+07] showed that all BitTorrent nodes contribute resources that do not directly improve their performance, and selfish peers can significantly reduce their contribution and yet improve their download performance. Samsara [CN03] considered peer-to-peer storage systems and proposed equal exchange of storage space between peers.

The downside of tit-for-tat is that it only works when two peers are interested in the resource possessed by each other at the same time. The difficulty in finding exchange partners increases with network size and the likelihood decreases as the varieties of resource increase.

1.3.3 Electronic currency

A natural improvement to barter economy is to leverage some form of electronic currency as a medium of exchange. A number of micropayment schemes have been proposed to support lightweight transactions over the Internet (see Wayner [Way97] for a survey). These schemes can be applied to peer-to-peer systems for peers to exchange for services [Moj]. In particular, efforts have been spent on building such notion in peer-to-peer systems [VCS03] and anonymous communication networks [RWW05]. SHARP [FCC+03] is a framework for distributed resource management, where each user can issue its own currency, known as *claims*, and trade resources like bandwidth with trusted peers.

Trading and payments architectures may be too expensive for many peer-to-peer distribution systems, as each operation would incur cryptographic operations and additional communication. Moreover, implementing micropayments either requires a centralized authority to issue currencies, or uses distributed trust and currency, which still remains as largely unresolved.

1.3.4 Reputation

An alternative to payments is to keep track of each user's past behavior, either everything they have done before (reputation) or only when they deviate from acceptable behavior (accountability). Dingledine et al. [DFM01] surveyed many schemes for tracking nodes' reputations. In particular, if obtaining a new identity is cheap and positive reputations have value, negative reputation could be shed easily by leaving the system and

rejoining with a new identity.

There have been several proposals on building a centralized reputation system for peer-to-peer applications [AD01, DDPS03, GJA03, NT04]. Furthermore, Blanc et al. [BLV05] considered applying a reputation system to the more fundamental peer-to-peer routing incentive problem. An important example of a somewhat successful reputation management is the online auction system eBay [eba], where buyers can rate sellers after each transaction, and eBay acts as the centralized system to store and manage these ratings.

While the aforementioned reputation systems require a centralize trusted authority, EigenTrust [KSGM03] computes reputation in a fully distributed and secure fashion. In EigenTrust, the global reputation of each peer is given by the local trust values assigned to that peer by other peers, weighted by the global reputations of the assigning peers. The security and scalability of the reputation system is achieved by disallowing a peer to compute its own trust value and computing each trust value by more than one peer.

A potential threat for all reputation systems, including EigenTrust, is that colluding peers that are otherwise cooperative can boost up the reputation of each other, giving them an unfair advantage over other peers.

1.3.5 Byzantine faults

A common way to abstract computers' failure to follow prescribed behavior is to employ a Byzantine failure model [LSP82]. This model encompasses any arbitrary fault that occurs in a distributed system. In general, mechanisms to prevent or mitigate malicious behavior, such as Byzantine quorums [MR97] and Byzantine state machines [CL99], may be

employed as defensive measures. Castro et al. [CDG+02] described techniques that make peer-to-peer substrates robust to collusions of a minority of malicious nodes in the overlay who attempt to compromise the overlay. These measures are typically very expensive and feasible only under the assumption that most nodes will behave correctly.

More recently, Aiyer et al. [AAC+05] introduced the *BAR* (Byzantine, Altruistic, and Rational) model as a foundation for reasoning about cooperative services. Under this model, they defined a replicated state machine protocol that can tolerate both Byzantine users and an unbounded number of rational users, and demonstrated the architecture through a cooperative backup service. Li et al. [LCW+06] combined the BAR replication protocol with the Gossip algorithm [DGH+88] to create a peer-to-peer data streaming application.

PeerReview [HKD07] is a system to provide accountability in distributed systems. It works by maintaining a secure record of messages sent and received by each node. These records are then used to automatically detect when a node's behavior deviates from that of a given reference implementation. This ensures that Byzantine faults are eventually detected, and a correct node can defend itself against false accusations.

These general primitives are alternate approaches to deal with freeloading in peer-to-peer systems. A comparison of these approaches against the methods used in this dissertation can be found in Chapter 7.

1.3.6 Game theory

Game theory is a field to use mathematics to capture behavior in strategic situations, in which an individual's success in making choices depends on the choices of others. It is suitable for analyzing peer-to-peer systems, where individual users interact with each other in a selfish fashion, and there are often no central authorities to oversee and govern activities of each user. Axelrod [Axe81] investigated the necessary and sufficient conditions under which cooperation will emerge in these situations. A more recent overview of the entire field can be found in Young [You98]. These studies explain why ungoverned peer-to-peer systems can function at all, and what properties are necessary for these systems to function well and be collectively stable.

1.3.7 Mechanism design

A closely related field to designing incentives into peer-to-peer systems in economics is *mechanism design*. Mechanism design is concern with the design of rules of games that involves multiple self-interested agents to achieve a specific system-wide outcome (see Mas-Colell et al. [MCWG95] or Varian [Var92] for references). Nevertheless, mechanism design may be considered "hyper-rational" for human behavior. It may be a more appropriate model for computers (software agents), as software agents generally have better computational powers than human beings [Var95].

Nisan and Ronen's seminal work on *algorithmic mechanism design* combined theoretical computer science's traditional focus on computation tractability with incentive com-

patibility in economics [NR01]. Feigenbaum et al. further extended the model to a setting where relevant information and computation are all inherently distributed [AFK+04, FKSS03, FPS01].

Mechanisms design guarantees incentive-compatibility, but they usually demand coordination efforts from some central authorities to collect users' preferences and to define policies. Moreover, all the aforementioned works require electronic currency to facilitate payment between peers. These requirements make them impractical for many peer-to-peer systems.

1.3.8 Economic analysis

There are also many economic studies and analyses that provide insights to the art and science of designing peer-to-peer systems. Friedman and Resnick [FR01] noted that the Internet allows one to easily erase his reputation by changing his name. This creates a situation where positive reputations are valuable, but negative reputations do not stick. Their study concluded that this model does not sustain complete cooperation, and a natural convention is to distrust or even mistreat strangers until they establish positive reputations. For resource-sharing peer-to-peer applications, it is therefore necessary for peers to limit the resources available to newcomers, even at a cost to the system as a whole.

Fuqua et al. [FNW03] considered a game-theoretic model to study the economic behavior of peer-to-peer storage networks. Their study showed how to model users' utility in a peer-to-peer system. This provides insights to system administrators on how system parameters can be chosen by soliciting votes from users, and how users with similar preferences

might cluster into multiple systems.

Fehr and Gächter's study considered an economic game where selfishness was feasible but could easily be detected [FG02]. When their human test subjects were given the opportunity to spend their money to punish selfish peers, they did so, resulting in a system with less selfish behaviors. This result justifies that users will be willing to pay some costs to ensure fairness.

1.4 Contribution of this Thesis

This thesis is concerned with solving the incentives problem in peer-to-peer systems. By studying the incentives of the users, resource management and policy enforcement can be designed directly into peer-to-peer systems. More specifically, instead of leaving resource contribution as a user option and relying on users' goodwill and altruism, it should be made as a necessary condition for one to receive good services from the systems. To this end, if good services are only provided to cooperative peers, freeloaders would find their quality of service drops sufficiently that they may prefer to either leave the system for good or start contributing resources back.

A distributed resource enforcement scheme is usually preferred even when a centralized solution is feasible. By incorporating incentive into the design, it removes reliance of a centralized authority for systems to function correctly, therefore improving their availability and scalability.

To provide incentives to promote cooperations in peer-to-peer system, one has to first

understand user incentives, which vary greatly for different applications. For instance, in an archival storage systems, users want to have reliable long term storage space; whereas in content distribution systems, users want to retrieve contents quickly. In this thesis, I propose to solve the incentives problem in peer-to-peer applications by:

1. Identifying the incentives of users;
2. Defining a fair sharing policy; and
3. Designing a mechanism to enforce the policy in a (mostly) distributed manner.

My thesis research is focused on applying this approach to different applications. The major contributions of this thesis include the followings.

- Define selfishness as a separate attack model in peer-to-peer systems and compare it against the traditional adversarial model.
- Propose a general approach to solve the incentive problem in peer-to-peer systems.
- Using the proposed approach, show how the incentive problem can be addressed in peer-to-peer archival storage systems, streaming systems, content distribution systems, and anonymous communication systems.
- Discuss general conditions for incentive schemes to work in peer-to-peer systems.
- Generalize several common design principles for large-scale peer-to-peer systems.

1.5 Guide to the Thesis

The rest of this thesis is organized as follows.

Chapter 2 provides the background, advantages, and taxonomy of peer-to-peer systems. It discusses the assumptions and the model of peer-to-peer systems that this thesis is focused on.

Chapter 3 describes the incentive scheme for the first application, peer-to-peer archival storage systems, which allows participants to exchange disk space to store their backup data. It proposes a novel policy enforcement scheme by allowing peers to probabilistically audit each other.

Chapter 4 considers the applications of peer-to-peer streaming systems, which utilize the collective bandwidth of users to forward multimedia streams to each other. To ensure fairness, the streaming network layout changes over time and peers can monitor each others' contributions to the system to decide whether to provide services to them.

The third application, peer-to-peer content distribution systems, is discussed in Chapter 5. In those systems, users can download objects of their interest from other computers possessing the objects. The problem is solved by tracking "debts" between peers and providing a scheme for leveraging debts to retrieve objects.

Peer-to-peer anonymous communication systems are the last application considered. Chapter 6 presents an incentive scheme for systems that help users to communicate anonymously on the Internet. Fairness is enforced by relying on some trusted authorities to measure the level of contribution of each peer and to decide the level of service it can

receive.

Chapter 7 summarizes the techniques used to introduce incentives for different applications. It also discusses the conditions for these incentive schemes to work and when they should be used. Finally, Chapter 8 discusses future work and concludes.

Chapter 2

Background and Model

This chapter gives an overview of peer-to-peer systems and provides the definitions of some of the terminologies used throughout this thesis. It also discusses security problems in peer-to-peer systems and defines the threat model.

2.1 Peer-to-peer Networks

In a peer-to-peer network, all computer nodes have identical or similar roles. Each node may be a service provider (server) as well as a service consumer (client). Unlike the more common client-server paradigm, there is usually no dedicated server to provide services; instead, the functionality of providing services is distributed among all participating computers. Peer-to-peer networks are usually *self-organizing* and do not need much administrative setup or maintenance. Most peer-to-peer networks are also *overlay* networks, where connections between nodes are built on top of an existing network, typically the Internet.

In the purest form, a peer-to-peer network should be completely decentralized and every single node in the network should share the exact same responsibilities. However,

many peer-to-peer networks rely on some kind of centralized components to provide limited functionality, such as to provide a point of entry or to assign a unique identifier number to a joining node. Some systems also allow nodes to serve different roles according to their level of trust and resources. For example, some utilize the concept of *supernodes*, where a small subset of more stable, resource-rich nodes provides routing and discovery services to the rest of the network.

Decentralized system design has been around for decades. Early examples include Usenet news server, email transmission protocol (SMTP), and Internet Relay Chat (IRC). However, most more recent applications followed the client-server paradigm because of the simplicity of its design. This remained until recent years, when the enormous growth of the Internet called for more scalable system designs and better utilization of otherwise unused resources at end points.

2.1.1 Unstructured and structured networks

A peer-to-peer network can either be *structured* or *unstructured.* In an unstructured peer-to-peer network, there is no specific rules to govern the connections between nodes; each node has the complete freedom of choosing its neighbors. As a consequent, nodes tend to choose other nodes that are in close proximity or share the same interest, if not entirely randomly.

A major drawback for unstructured peer-to-peer networks is the lack of guarantee in efficiency and scalability. For instance, searching is usually inefficient. Since there is no central directory, a node could appear in any part of the network. Without flooding the

entire network, there is no guarantee in locating a particular node or resource. This problem can be solved by exerting a graph structure in the network. Generally, in a structured peer-to-peer network, each node has a unique identifier from a large numeric space, referred to as a *nodeId*. A node selects its neighbors from the set of nodes whose nodeId satisfy certain constraints relative to its own nodeId. The constraints differ for different neighbor set slots and some are more restrictive than others. With this constraint in place, a search for a particular node can be done by progressively moving closer to that node in the nodeId space. This guarantees that, under normal network conditions, any node can be reached from anywhere in the network within a maximum route length, usually logarithmic to the network size.

2.1.2 Distributed hash table (DHT)

One common functionality provided by structured peer-to-peer networks is as a *distributed hash table* (DHT). Similar to a traditional hash table, a DHT provides lookup services from a *key* (or name or identifier) to a value (a resource or a resource location). However, the responsibility for the maintenance of the association between keys and values is distributed among the nodes in the network. This is usually done through a *key-based routing service* [DZD+03], where given a key, a request will be routed to the node that is responsible for that key. Similar to searching for a node, any object in the network can be found with a number of network hops logarithmic to the size of the network.

2.1.3 Advantages

Since the peer-to-peer paradigm does not require any centralized component, it has many advantages over the client-server paradigm. First, it is more *available* and *reliable*, since the system can still function when part of the network has failed or is under maintenance. Second, while demands for services grow with the network size, so will the aggregated resources provided by the new peers, making peer-to-peer networks fundamentally *scalable*. Peer-to-peer networks are also *autonomous*, as the lack of centralized component means that there is no need for dedicated management and maintenance of the network.

2.1.4 An example: Pastry

There are many different structured peer-to-peer networks proposed in recent years, including CAN [RFH+01], Chord [SMK+01], Kademlia [MM02], and Tapestry [ZHS+04]. While the designs in this thesis are mostly independent of the peer-to-peer substrate, *Pastry* is discussed here as a reference.

Pastry [RD01a] is a structured peer-to-peer overlay network that provides a key-based routing service. Each Pastry node has a unique, 128-bit nodeId. Given a message and a key, Pastry can normally route the message to the live node whose nodeId is numerically closest to the key in less than $\lceil \log_{2^b} N \rceil$ hops, where N is the number of nodes in the network and b is a configuration parameter with a typical value of 4. A Pastry node's routing table is organized into $\lceil \log_{2^b} N \rceil$ rows with $2^b - 1$ entries in each row. The jth entry in row i refers to a node whose nodeId shares the first i digits with the current node but with its $i+1^{\text{th}}$

digit being j. An entry may not be defined if no node with suitable nodeId is known. This definition of a routing table enables *prefix routing*, where each routing hop can usually match at least one more digit in nodeId.

In addition to a routing table, each Pastry node also maintains a *leaf set*, which is defined as the $\ell/2$ nodes with numerically closest nodeIds in each direction in nodeId space. The value of ℓ is usually 16. The inclusion of the leaf set improves the connectivity of the network under high churn rate.

FreePastry [Fre] is an open source implementation of Pastry in Java. It provides the peer-to-peer substrate Pastry and several applications built on top, including anycast primitive in Scribe [CDKR02], archival storage utility in PAST [RD01b], and content streaming and distribution system in SplitStream [CDK$^+$03]. FreePastry is used in some of the simulations in this thesis.

2.2 Attacks and Threat Model

This section discusses attacks in peer-to-peer systems and the threat model considered in this thesis.

2.2.1 Malicious attacks

Malicious attacks are defined as any deviation from the prescribed behavior by an adversary with malicious intent. The objective of the adversary may include obtaining unauthorized access to content, corrupting or censoring content, or denying or degrading services

to other users.

Sybil attacks

An inherent weakness of open peer-to-peer networks is the lack of central identity management. When a new node joins a network, it needs to create an identity. While each identity in the network corresponds to a unique node, and thus a user, a user may join the network under multiple identities. Enforcing a one-to-one mapping between nodes and identities is made difficult by the distributed nature and openness of many peer-to-peer systems. In fact, one simple attack on an open peer-to-peer network is to join the network, using the same node or a handful of nodes, many times under different identities. This is known as the Sybil attack [Dou02]. Once controlling a sufficiently large fraction of the network, the adversary would be able to eavesdrop traffic and censor messages. It could also cut off nodes from the network if they are only connected to nodes controlled by the adversary. Worse, if the peer-to-peer network relies on consensus, these results could easily be manipulated by the adversary.

Several suggestions have been made on defending or limiting the effects of Sybil attack, including "hashcash" [Bac02], bounding number of neighbors of each node [SNDW06], and relying on social networks [DLLKA05, MPDG08, YKGF06, YGKX08]. Cheng and Friedman [CF05] proposed a framework for assessing a reputation mechanism's robustness to Sybil attacks. However, to date, the only known robust method is to require some form of a centralized authority to assign identities and maintain a *public key infrastructure* (PKI). Under this infrastructure, any user would have a fixed identifier to be used in only one node.

Because the centralized authority's sole purpose is to assign identifiers, it is not involved in regular peer-to-peer transactions. This reduces the cost to implement the centralized authority, as it needs not provide highly available, scalable, or redundant service.

As a side benefit, a PKI also gives each node in the network the capability of digital signature. Any node can digitally sign any data such that any other node can verify the origin, yet it is computationally infeasible for other nodes to forge. This allows nodes to freely contact and negotiate with their peers without worrying about the authenticity and integrity of the communication.

Other malicious attacks

Traditional threat models for distributed systems usually assume an adversary that can take over at most a small fraction of the servers. Even if a solution for the Sybil attack is assumed, there are still numerous attacks the adversary can mount on the network.

At the lowest level, an adversary may deviate from the prescribed protocol, say by not forwarding messages. Most peer-to-peer systems are already engineered to be robust against traffic loss due to network failures. In the extreme case of a node refusing to properly forward low-level traffic, that nodes' neighbors could flag the node as unresponsive and would likely remove the node from the network. Regardless, these types of deviation from prescribed behavior can be classified as Byzantine faults and can be handled as such, as discussed in Section 1.3.5.

2.2.2 Freeloading "attacks"

While traditional threat model assumes an adversary may attack the system in any arbitrary way, even if the attack is costly to the adversary, the predominant form of misbehavior observed in peer-to-peer systems is freeloading.

In a sense, freeloaders more closely resemble economically "rational" agents who will behave correctly, but only if good behavior maximizes their "utility" from the peer-to-peer network. While any deviation from correct or desirable behavior can be treated as a security attack and handled accordingly, freeloading in peer-to-peer systems raises a somewhat non-traditional security model. Users of peer-to-peer systems are not motivated to attack the system; they only care about maximizing their own welfare. For instance, in a file-sharing system, if uploading files to peers has no effect on a user's download rate, the user may choose to not upload. However, users have no incentive to upload incorrect data or otherwise try to damage other nodes in the network.

Moreover, freeloading is relatively easy. In Kazaa, for example, a client configured to have minimum upload bandwidth suffices to freeload. Similar settings also exist in many variations of BitTorrent. This is unlike a malicious attack, which would require considerable technical expertise. Thus, the fraction of users who have the motivation and ability to freeload is likely to far exceed those that are intent and able to mount a malicious attack.

2.2.3 Threat model

Since malicious attackers and freeloaders have completely different intent and behavior, the two threats call for different mechanisms. A defense against malicious behavior can, and often must, assume that malicious behavior is limited to a minority of users. A defense against freeloading must be effective and efficient even when a large fraction of participants attempt to freeload. These defenses should be complementary to each other and should be used in conjunction. As a result, the adversarial model in this thesis is limited to simple freeloading behavior on the application layer, where the only objective of the adversary is to obtain services without contributing a reasonable amount of its resources to the system.

Chapter 3

Archival Storage Systems

3.1 Introduction

The first application to consider is peer-to-peer archival storage systems. In these systems, users store their data remotely on other peers but rarely retrieve their data. Thus, storage space (i.e., free disk space) becomes the limited commodity. An example of a system like this is a remote backup service, where only the writer of a given data block might want to read it. The data may go unread until the writer suffers some kind of catastrophic equipment failure, which could be relatively rare in modern systems. A simple fair policy is to require each node to provide at least the same amount of storage space it uses from the system.

> **Design objective:** The amount of storage space a user consumes from the network should not exceed the amount of space it is providing back.

Note that the ability to consume resources, such as remote disk storage, is a form of commodity, as remote resources have different values to a user than its local storage. When users exchange their local storage for others' remote storage, the trade could benefit both

parties, giving an incentive for them to cooperate. As such, there is no need for cash or other forms of real-world money to exchange hands; the economy can be expressed strictly in terms of single-good barter schemes in storage.

This chapter is based on a paper published in the Second International Workshop on Peer-to-Peer Systems (IPTPS) [NWD03]. This was a joint work with Dan S. Wallach and Peter Druschel.

3.2 Ensuring Honest Storage

For any untrusted network storage system, it is imperative to ensure the remote nodes are honestly storing the files they are claiming credit for. This is guaranteed by the following *challenge* mechanism.

For each file a node, say Alice, is storing in the network, she periodically picks a node, say Bob, that stores a replica of the same file as a target. Alice notifies all other replicas holders of the file that she is challenging Bob. Then she randomly selects a few blocks of the file and a random key, and queries Bob for a keyed hash of those blocks. Bob can answer correctly only if he has the file. Bob may ask another replica holder for a copy of the file, but any such request during a challenge would cause Alice to be notified, and thus restarts the challenge for another file.

A potential problem is that colluding nodes storing replicas of the same file could just store one copy instead of two, however this would be unlikely to occur. The nodes responsible for replicating a given file are typically constrained; in PAST [DR01], for example,

they are the set of adjacent nodes with nodeIds closest to the object handle. With random nodeId assignment, few colluders would be adjacent to each other in nodeId space.

3.3 Design

Instead of directly enforcing quota or employing any capability in the system, this design requires nodes to maintain their own records and publish them, so that other nodes can audit those records. Of course, nodes have no inherent reason to publish their records accurately. This section describes how to create natural economic disincentives to nodes lying in their records.

3.3.1 Usage files

Every node maintains a *usage file*, digitally signed, which is available for any other node to read. The usage file has three sections:

- the *advertised capacity* this node is providing to the system;
- a *local list* of tuples (file handle, file size, nodeId), containing information of all files that the node is storing locally on behalf of other nodes; and
- a *remote list* of handles of all the files published by this node (stored remotely), with their sizes.

Together, the local and remote lists describe all the credits and debits to a node's account. Note that the nodeIds for the peers storing the files are not stored in the remote list,

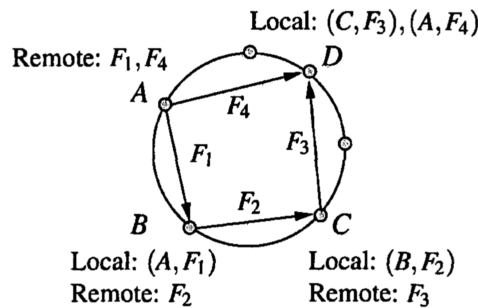


Figure 3.1: A peer-to-peer system with nodes showing their local and remote lists. For simplicity, the size of the files are not shown.

since this information can be found using mechanisms in the storage system (e.g., PAST). A node is said to be "under quota," and thus allowed to write new files into the system, when its advertised capacity minus the sum of the files in its remote list multiplied with the number of replications, is positive. Since the entries in local/remote lists have to be matched, all usage files have to be balanced. By increasing the advertised capacity, a node can store more files on the system, but it also has to make an equal amount of space available. By adding matched pairs in the local list of one node and the remote list of another, the credit is transferred from the latter node to the former.

When a node A wishes to store a file F_1 on another node B, B must first fetch A's usage file to verify that A is under quota. Then, two records are created: A adds F_1 to its remote list and B adds (A, F_1) to its local list. This is illustrated in Figure 3.1. Of course, A might fabricate the contents of its usage file to convince B to improperly accept its files.

3.3.2 Attacks and audits

It is necessary to provide incentives for A to tell the truth. To game the system, A might normally attempt to either *inflate* its advertised capacity or *deflate* the sum of its remote

list. If A were to increase its advertised capacity beyond the amount of disk it actually has, this might allow A to consume more space than it is actually providing when the system is significantly under capacity. This is not the probable case, since providing additional space creates additional bandwidth overhead in storing files and challenges. Moreover, if the system is underutilized, nodes consuming unfair amount of space is not a serious issue. When the utilization becomes high, A will attract storage requests that it cannot honor. A might compensate by creating fraudulent entries in its local list claiming the storage is used. To prevent fraudulent entries in either list, there is an auditing procedure that B, or any other node, may perform on A.

If B detects that F_1 is missing from A's remote list, then B can feel free to delete the file.[1] After all, A is no longer "paying" for it. Since it would be possible for A to provide a tailored version of the usage file if it knew the identity of its auditor, anonymous communication is required, which can be accomplished using intermediate nodes to relay the request, a technique similar to Crowds [RR98]. As long as every node that has a relationship with A is auditing it at randomly chosen intervals, A cannot distinguish whether it is being audited by B or any other node with files in its remote list. This process is referred to as a *normal audit.*

Random audits. Normal auditing — alone — does not provide a disincentive to inflation of the local list. For every entry in A's local list, there should exist an entry for that file in another node's remote list. An auditor could fetch the usage file from A and then connect

[1]In practice, B should give A a grace period as A might be facing a transient failure and actually need the backup.

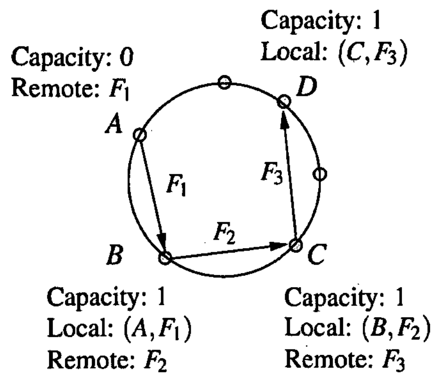


Figure 3.2: A cheating chain, where node A is the cheating anchor with unbalanced usage file but pushed its debt along the chain to node D.

to every node mentioned in A's local list to test for matching entries. This would detect inconsistencies in A's usage file, but A could collude with other nodes to push its debts off its own books. To fully audit A, the auditor would need to audit the nodes reachable from A's local list, and recursively audit the nodes reachable from those local lists. Eventually, the audit would discover a *cheating anchor* where the books did not balance (see node D in Figure 3.2). Implementing such a recursive audit would be prohibitively expensive. Alternatively, all nodes in the peer-to-peer overlay are required to perform *random auditing*. With a perhaps lower frequency than their normal audits, each node should choose a node at random from the peer-to-peer overlay. The auditor fetches the usage file, and verifies it against the nodes mentioned in that file's local list. Assuming all nodes perform these random audits on a regular schedule, every node will be audited, on a regular basis, with high probability.

To see how frequently each node would be audited, consider a system with N nodes, where $c \ll N$ nodes are conspiring. Assume the c conspiring nodes build a cheating chain, where there is only one cheating anchor. The probability that the cheating anchor is not

random audited by any non-conspiring node in one period is

$$\left(1 - \frac{1}{N-1}\right)^{N-c} ,$$

which approaches to $1/e \approx 0.368$ for large N. In other words, the cheating anchor would be discovered in three periods with probability over 95%.

Recall that usage files are digitally signed by their nodes. Once a cheating anchor has been discovered, its usage file is effectively a *signed confession* of its misbehavior. This confession can be presented as evidence toward ejecting the cheater from the peer-to-peer system. Unlike reputation systems, the proof is non-repudiable, regardless of the credibility of the auditor. With the cheating anchor ejected, other cheaters who depended on the cheating anchor will now be exposed and subject to ejection, themselves. This would not affect non-conspiring nodes, however, as they can simply delete the involved files and make the space available for other storage requests.

Note that this design is robust even against bribery attacks, because the collusion will still be discovered and the cheaters ejected. Also note that since everybody, including the auditors, benefits when cheaters are discovered and ejected from the peer-to-peer system, nodes do have an incentive to perform these random audits [FG02].

3.3.3 Cheating patterns

While cheating chains can be easily discovered and ejected from the system, a possible attack is for the cheating node to push back its debt to another node which, when audited,

will push back its debt to another node, eventually forming a cycle instead of a chain. This might be attempted either with or without increasing the advertised capacity. The following considers both cases.

Without increasing the advertised capacity, a cheating node can only balance its usage file by removing some entries from its remote list. By forcing nodes to maintain logs of changes made to their usage files and marking each update with the (logical) timestamps of all involved parties [Lam78], the cheating anchor would be forced to give a total order on all its changes. If a node claims to remove an entry from its remote list after it has exceeded its quota, its log would be a signed confession that it has cheated. Otherwise, it has to withdraw a file from the system before storing new files. This eliminates any benefit the node might get, for itself, by participating in a cheating chain.

By increasing its advertised capacity without bound, a node might be able to gain additional storage credit in the system, hoping to take advantage of the system operating far below its actual capacity. While a freeloading agent might temporarily benefit from abusing other nodes' excess capacity, these freeloaders will be discovered and ejected as the system's free space decreases and they eventually cannot support the local storage demand of them. Thus, when legitimate nodes need space, freeloaders will be naturally pushed out of the way.

If the system is operating at or near its capacity, then any report of increased capacity on one node will immediately attract more files to be stored on that node. A node might try to simultaneously increase its published capacity and its list of locally stored files, claiming

to be full; the nodes responsible for replicating these objects will notice the discrepancy.

3.3.4 Expelling a misbehaving node

Once a node is discovered cheating, it needs to be ejected from the network. Note that there is already a signed confession of misbehavior, for which any node in the network can verify. Therefore, the ejection can be done by presenting this evidence to the node's current neighbors (in Pastry, for example, this means its leaf set and nodes with nodeIds that are best fit for its routing table).

The node that discovered the misbehavior would also have obtained the usage file of the offender during the process. Since the offender's files are likely included in the remote list of the usage file, nodes storing those files can be notified and have the files removed from the network. For those files that are not in the remote list, they would not survive through the next normal audit done by the storer anyway.

To prevent ejected nodes from rejoining the network, these evidence should be made available for any node to query before they agree to provide storage space. This can be achieved by creating a special *evidence file* for each expelled node and have these files also stored in the network. The handle of these files can be a hash of the nodeId of the expelled nodes, so that it is trivial to query for such evidence, if existed.

3.3.5 Preventing content distribution abuse

We have assumed that these storage systems are strictly for backup data. However, it is conceivable that users may abuse the service for content distribution, in which case the

network bandwidth becomes the limited resources. This type of abuse can be prevented by setting a limit on the number of times each file can be retrieved over a period of time and requiring file owners to digitally sign a receipt for each retrieval. These receipts can be stored in a similar fashion to the evidence files and can become grounds for expelling a node from the network if the retrieval limit is exceeded.

3.3.6 Handling churn

Nodes leaving and rejoining is an unavoidable part of peer-to-peer networks, since these nodes are often not dedicated servers and are not maintained as such. However, a storage network needs to have high stability and availability, and any churn could be very costly since it could trigger relocation of files in the network. The hard question is whether file replications should be delayed in the hopes that a failed node will come back alive later on.

A rule of thumb is that nodes with transient failures should not be removed from the network, but nodes with poor availability should be expelled. Consequently, there should be a minimum level of availability that each node agrees to provide. Neighboring nodes (e.g., those in the leaf set in Pastry) could monitor the availability of each other, and only when a node is unable to sustain this level of availability should its neighboring nodes arrive to a consensus and expel that node from the network.

3.3.7 Extensions

Selling overcapacity. Using this mechanism, a node cannot consume more resources from the system than it provides itself. However, it is easy to imagine nodes who want

to consume more resources than they provide, and, likewise, nodes who provide more resources than they wish to consume. Naturally, this overcapacity could be sold, perhaps through an online bidding system [CGM02a], for real-world money. These trades could be directly indicated in the local and remote lists. For example, if D sells 1GB to E, D can write (E, 1GB trade) in its remote list, and E writes (D, 1GB trade) in its local list. All the auditing mechanisms continue to function.

Reducing communication. Another issue is that fetching usage logs repeatedly could result in serious communication overhead, particularly for nodes with slow network connections. Three optimizations can be used to reduce this overhead. First, rather than sending the usage logs through the overlay route used to reach it, they can be sent directly over the Internet: one hop from the target node to the anonymizing relay, and one hop to the auditing node. Second, since an entry in a remote list would be audited by all nodes replicating the logs, those replicas can alternately audit that node to share the cost of auditing. Third, communication can be further reduced by only transmitting the differences between usage logs, since the logs change slowly. Note that this must be done carefully to ensure that the anonymity of auditors is not compromised. For instance, a node could provide different sets of version numbers to different auditors to try to relate the sources between audits. To address this, the auditor needs to, from time to time, request the complete usage logs.

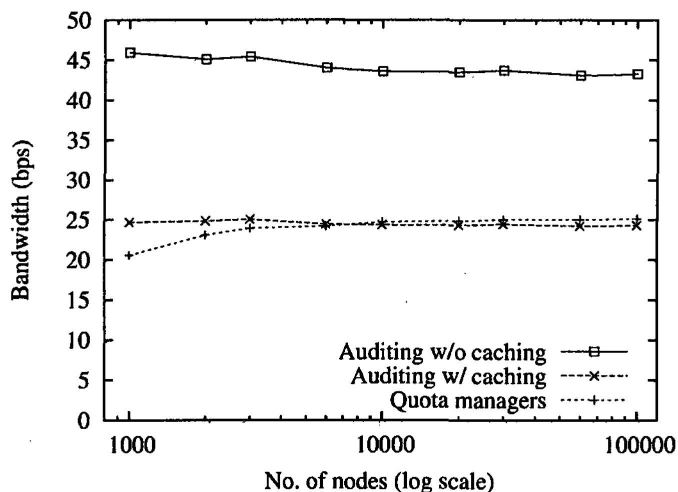


Figure 3.3: Overhead with different number of nodes. The overhead remains at the same level with increasing number of nodes, and auditing with caching costs similar to quota managers.

3.4 Experiments

We implemented a simulator to measure the communication cost of the design. The cost is compared against the following *quota manager* approach: Each node have a set of other nodes denoted as its *manager set*. Each manager must remember the amount of storage consumed by the nodes it manages and must endorse all requests from the managed nodes to store new files. To be robust against minority collusion, a remote node would insist that a majority of the manager nodes agree that a given request is authorized, requiring the manager set to perform a Byzantine agreement protocol [CL99].

Figure 3.3 shows the average upstream bandwidth required per node, as a function of the number of nodes (the average required downstream bandwidth is identical). The per-node bandwidth requirement is almost constant, thus all systems scale well with the size of the overlay network. Moreover, with proper caching, the cost for auditing can remain low and be comparable to the quota manager approach.

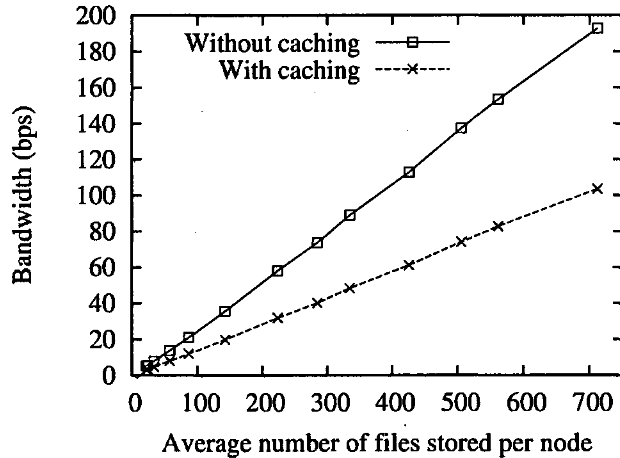


Figure 3.4: Overhead with different number of files stored per node. The overhead grows linearly with the number of files stored per node.

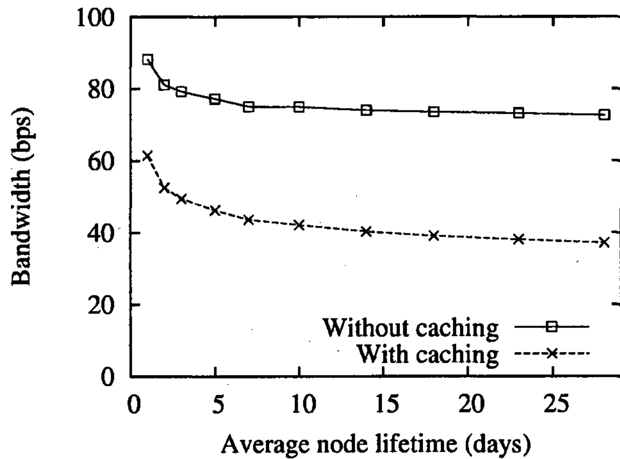


Figure 3.5: Overhead with different average node lifetime. The overhead remains at the same level with increasing number of nodes, and auditing with caching costs similar to quota managers.

Figure 3.4 shows the bandwidth requirement as a function of the number of files stored per node. The overheads grow linearly with the number of files, but for auditing without caching, it grows nearly twice as fast as auditing with caching. If the system is used for large files, such as might be expected for backup systems, this overhead would be inconsequential. Nodes can also be incentivized to store a smaller number of larger files by defining a system-wide minimum file size.

Figure 3.5 shows the overhead versus average node lifetime. As node lifetime increases, the overhead rapidly stabilizes at a low level. While most Internet-based file-sharing systems have relatively short node lifetimes, it is expected that a storage system would have see a much longer node lifetimes. Long lifetimes will be necessary, in any case, to avoid having normal use of the system dominated by the bandwidth costs of maintenance and replication [BR03].

3.5 Summary

In summary, auditing with caching has a very low overhead, linear in the number of files stored and scaling well as the number of nodes in the system grows. Relative to the bandwidth required for storing and retrieving files, the auditing overhead is of the order of tens of bps (bits per second), only a small fraction of a typical participating peer-to-peer node's bandwidth on an archival system. Auditing provides a practical and bandwidth-efficient mechanism to ensure fair sharing in storage-constrained systems, providing resistance to malicious nodes and scalability to large peer-to-peer systems.

3.6 Related Work

In a storage network, nodes share spare disk capacity for applications such as distributed backup systems. Tangler [WM01] is designed to provide censorship-resistant publication over a small number of servers (i.e., less than 30), exchanging data frequently with one another. To maintain fairness, Tangler requires servers to obtain "certificates" from other

servers which can be redeemed to publish files for a limited time. A new server can only obtain these certificates by providing storage for the use of other servers and is not allowed to publish anything for its first month online. As such, new servers must have demonstrated good services to the peer-to-peer system before being allowed to consume any system services.

Palimpsest [RH03], Tangler, and earlier systems like Gnutella and FreeNet all provide *ephemeral* storage. These systems make no guarantees that a file will be available indefinitely. While popular files can be widely replicated, unpopular files will disappear for lack of interest and would then need to be reinserted. This contrasts with my work, where inserted files can live forever.

Samsara [CN03] enforces fairness by charging peers storage space in the form of a claim, which can be replaced by real data when needed. The transfer of claims doubles the space and bandwidth required to store data. Claims can be forwarded to form chains or cycles to improve space efficiency. Since cycles can only be formed by chance, chains can grow to $O(N)$, where N is the size of the overlay network (they witnessed a chain consisting of over $3/4$ of the nodes in their experiment). In this case, a single failure could cause a majority of nodes to lose data, casting doubts on the scalability of the system.

Cooper and Garcia-Molina considered peer-to-peer trading for a small number of storage sites and propose algorithms on peer selection in order to increase global reliability [CGM02b]. Subsequently they described how auction and bidding could work, and examined policies for deciding when to call an auction and how much to bid [CGM02a].

Chapter 4

Streaming Systems

4.1 Introduction

In a media streaming system, a central node broadcasts a video and/or audio stream to subscribers. It is often necessary to leverage the subscribers' bandwidth to help disseminating the content. The participating nodes usually form a tree structure, with streaming content sent from the root through interior nodes to leaf nodes. Peer-to-peer multicast systems [CDK+03, CGN+04, JGJ+00, KRAV03, ZZJ+01] have demonstrated that when participating nodes are cooperative, media streaming applications can scale to reliably support large numbers of nodes without the need for the costly server and network infrastructure.

However, if a node was to refuse to transmit data to its downstream peers, or to accept any downstream peers, it could freeload on the system. If every node were to follow a similar policy, the system as a whole would collapse. This chapter considers incentive issues in the context of peer-to-peer multicast streaming services. This work was done in collaboration with Peter Druschel and Dan S. Wallach, and a preliminary version of this work was published in the Second Workshop on the Economics of Peer-to-Peer Systems [NWD04].

4.2 System Model

The system described in this chapter, called *FairStream*, is designed to run on tree-based peer-to-peer multicast systems that concurrently use multiple trees. For concreteness, the description of the design is based on *SplitStream* [CDK+03].

SplitStream is an application-level multicast system built on top of Pastry. The key idea behind SplitStream is to split the original content stream into k stripes and to multicast each stripe using a separate multicast tree. Nodes subscribe to k different trees. Every node will (most likely) be an interior node in exactly one tree and will be a leaf node in the remaining $k-1$ trees. If each node supports a fan-out to k children, then the total in-degree and out-degree would be equal.

This splitting strategy in SplitStream is usually used in conjunction with a multiple description coder [Apo01, AW01, MRL00, PWCS02], so that the receivers' stream quality is proportional to the number of stripes they receive. Together, they provide robustness against packet loss, and also ensure a uniform sharing of the multicast transmission costs across participants.

4.2.1 FairStream basics

Similar to SplitStream, FairStream creates up to k concurrent multicast trees to stream the k parts of the multicast data. The stripe bandwidth is chosen such that every Internet user with a reasonable connection can receive and forward at least one stripe. In this way, users with weak Internet connections can still participate, albeit with a smaller number

of stripes. Moreover, nodes with more bandwidth would have an incentive to serve more children, because that will earn them more quality. The objective of the design is therefore to reward nodes that stream more content.

Design objective: The more stream content a node forwards to other nodes (in a timely fashion), the more stream content it should receive from its peers.

FairStream leaves the construction of multicast trees to the underlying multicast system. To avoid spending bandwidth to forward data, freeloaders may employ the following two strategies:

Choking strategy A freeloader can refuse to forward any data to its children after a multicast tree is constructed by claiming that it could not receive any data. Downstream peers could not easily tell whether their parents were lying.

Rejection strategy A freeloader can also refuse to accept children in the first place. A parent could do this by, perhaps, falsely claiming that it has already accepted enough children and its outgoing bandwidth is fully utilized.

In the latter case, the problem cannot be solved simply by requesting a list of children from a possible freeloader; such second-hand information could be easily falsified. A freeloading parent might, for example, claim that some conspiring nodes are its children. Such children would happily vouch that their parent is giving them services while it might, in fact, not be providing them any service at all. To avoid these issues, FairStream was designed to only gather information by direct observation. If a correct packet is received,

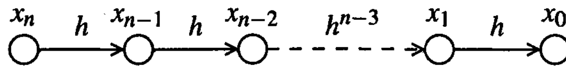


Figure 4.1: A hash chain, where each value of x is a hash of the value on its left.

every node from the parent to the root must have forwarded it; on the other hand, if a packet is not received, any of the nodes from the parent to the root might be responsible for the loss.

4.2.2 Data and path authenticity

FairStream relies on the knowledge of the path to the roots in the multicast trees. A freeloader, of course, would not necessarily cooperate to provide such information correctly. False information might allow a freeloader to stay hidden or even to falsely deflect blame to a well-behaved peer. This problem can be alleviated by the use of common cryptography primitives. While a number of techniques would suffice to authenticate a data stream and verify the integrity of its path, FairStream borrows ideas from hash chains [PCST01] and path authentication in Ariadne [HPJ02], which uses hash chains to achieve similar security semantics to digital signatures without requiring the overhead of expensive public-key cryptographic operations. In general, each packet will contain information that can be used to validate the previous packet.

First, the source creates a hash chain by randomly generating a value x_n and iteratively computing $x_{n-1}, \ldots, x_0$ by $x_i = h(x_{i+1})$ with a cryptographically secure one-way hash function h (e.g., SHA-1), as shown in Figure 4.1. The source will reveal x_0 initially and subsequent x_i's, one per packet. As such, n should be relatively large to avoid the need to

redistribute new x_0's very often. An important property of one-way hash functions is that while it is cheap to compute a hash, it is computationally infeasible to find its inverse. Thus, given x_{i+1}, it is trivial to verify that it hashes to x_i, but it is infeasible to find x_{i+1} from x_i.

When the source sends the ith packet, it also computes the packet's message digest $d_i = h(\text{data}_i, x_i)$. Whenever an interior node sends the packet, it hashes the message digest it receives from its parent with the receiving node's nodeId. Thus, the message digest received by the source's child A would be $h(d_i, A)$ and that by A's child B would be $h(h(d_i, A), B)$ and so on. Each packet will also include the base hash chain value used in the previous packet, i.e., the $i+1^{\text{th}}$ packet contains x_i. Upon receipt of x_i, each node can confirm that $x_{i-1} = h(x_i)$. Each node can then verify the integrity of the previous packet by reconstructing the message digest using x_i and the path it was told by its parent.

In case of lost packets, a node only needs to hash the value multiple times until it matches the last seen x_i. Likewise, a node joining an ongoing streaming session only needs to keep hashing the value until it matches x_0. When multiple multicast trees are being used, each multicast tree can use a separate hash chain so that the functioning of one tree would not be interfered by another.

Under this scheme, nodes cannot fake the path from the root to itself without knowing x_i, which would not be revealed until after the packet becomes obsolete. In particular, it is not possible for a freeloader to remove any node in the path to the root. It should be noted, however, that it is possible for conspiring nodes to include additional nodes in the path.

4.3 FairStream Design

This section describes the design of FairStream. FairStream focuses on mechanisms that individual nodes can follow, based strictly on information they observe about their peers, as well as information they can authenticate about nodes between themselves and the root of any given tree.

4.3.1 Overview

When a node participates in multicast streaming, it simultaneously interacts with many other nodes; for a system with k trees, it will typically have k parents and up to k children, not to mention its grandparents and other ancestors. If the trees are constantly rebuilt, the number of peers it has interacted with would keep growing over time. Nodes can easily account for services they have provided to and received from each peer. More importantly, nodes can also easily account how many times their peers have *failed* to provide them with good services. A peer that consistently fails to provide services is more likely to be a freeloader. Nodes can provide preferential services to peers that have proven cooperative history than peers that they barely know, and deny services to peers with suspicious tracks. Well-behaved nodes will be preferred by most nodes and receive near-perfect services, while freeloaders will be disliked by most nodes and denied services.

4.3.2 Periodic tree reconstruction

When multicast trees are constructed, some nodes may be in unfair or unfavorable positions. A lucky node might happen to be a leaf in all trees, whereas an unlucky node might happen to be downstream from a freeloader that is refusing to forward data. By periodically reconstructing the multicast tree, a node will only benefit or suffer from such situations for a fixed time period at most. To avoid affecting the stream, new multicast trees can be constructed concurrently while existing trees are in use. In this way, at least $k-1$ out of the k trees will be available for streaming data at any time. Of course, it is imperative that the trees are constructed with certain degree of randomness, so that the new multicast tree will be sufficiently different from the old one. There will remain a tradeoff between the bandwidth overhead of tree reconstruction and the desire for smaller time steps. Smaller time steps allow nodes to respond more rapidly when they detect that a node is being selfish.

4.3.3 Collecting information

FairStream collects two measurements through first-hand observations, namely *ancestor rating* and *parental availability*.

Ancestor rating. A node can monitor the services provided by its ancestors and decide how much *confidence* it has on them. The confidence values, denoted by c, are initially 0. Whenever a node receives a correct packet, it considers all the nodes in the path to the root, i.e., all its ancestors, have forwarded the packet and thus increments its confidence value of each node. Note that if the confidence values are incremented irrespective to the path

length, it would be possible for a node to help its conspiring peers to gain confidence from the downstream peers simply by claiming that the conspiring nodes are all its ancestors. The solution is to decrease the increment value for longer paths: The increment value is set to $1/2^{\ell}$, where ℓ is the depth of the node from the root. Under this scheme, the confidence value a node earns will be halved for each additional node introduced, and the sum of the confidence values gained by conspiring nodes, if added as ancestors, would be no more than the confidence value gained by a single node if none of those conspiring nodes were added.

Whenever an expected packet is not received, the node decrements the confidence value of each ancestor, blaming them all equally, for the lack of any more specific information. When the trees are reconstructed, any blame assigned falsely or due to lost packets would average out as nodes are later observed to behave correctly. Freeloading nodes, on the other hand, would be consistently blamed for their misbehavior. The decrement value is $w/2^{\ell}$, for some system parameter w. Thus, decrements are w times faster than increments.

Furthermore, positive confidence values are decayed over time, multiplied by a decay factor λ after each time step. As a result, nodes will forget how good they peers have been, but they will remember how bad they were. Nodes are thus forced to continue providing services to maintain their peers' confidence.

Parental availability. When a node joins a multicast tree and is refused services by its prospective parent, it has no way to determine if the prospective parent is genuinely at capacity or is freeloading. A freeloader can always claim to be serving its conspiring

peers. If a prospective parent has demonstrated a history of refusing to accept children, it is probably a freeloader. The likelihood that a peer is making itself available as a parent is defined as *parental availability*. The ability of a child to measure this parental availability will depend on the specific details of how multicast trees are constructed in any given system.

In FairStream, the parental availability p is initialized to a default value p_{init} at the beginning. Each time when the node requests the peer to become its parent, this value is updated as a weighted average of its previous value and $v \in \{0, 1\}$, a value denoting whether its peer accepts to become its parent, by $p' = \lambda p + (1 - \lambda)v$. In this way, a node has to constantly accept its peers as children if it wants to maintain a high parental availability among its peers.

4.3.4 Reciprocal requests

While well-behaved nodes would accept children in accordance with the prescribed protocol, freeloaders might regularly refuse to accept children. When a node A asks some prospective node B to be its parent, B needs a way to judge whether A has had a history of behaving selfishly. To address this, FairStream encourages B to occasionally break the join protocol and instead attempt to make A its parent by requesting to join directly under A for a multicast tree where A is supposed to be an interior node. This would allow B to determine whether A is misbehaving, and thus have a stronger basis for ignoring A in the future.

4.3.5 Selective servicing

FairStream makes use of both the confidence value c and parental availability p to decide whether to serve a peer. In particular, nodes should only accept children that have high confidence ($c > c_+$) or with both sufficient confidence and sufficient parental availability ($c > c_{\text{th}}$ and $p > p_{\text{th}}$). Note that more emphasis is put on the confidence value because it is more effective in distinguishing misbehaving nodes, as to be shown in Section 4.4. Once a node believes that a peer is a freeloader, it should cease to provide services to the peer by refusing to accept those peers as their children. Moreover, when it gets request from a peer it has good relationship with ($c > c_+$ and $p > p_{\text{th}}$), it should make space for it by dropping children with mediocre relationship.

Nodes should also be wary in choosing a parent, since a freeloader may accept children but refuse to forward data, which will in turn prevent them from forwarding data and affect their descendants' confidence. Specifically, a node should not request a peer to become its parent if its confidence on the peer is less than a minimum threshold ($c < c_{\text{min}}$).

4.4 Experimental Evaluation

FairStream is implemented on top of SplitStream. This section evaluates FairStream through simulations.

4.4.1 Modification from SplitStream

Other than the design changes as described in Section 4.3, the only modification we made in FairStream is the join protocol. In SplitStream, when a node joins a tree, it sends a subscribe message to a potential parent. If the potential parent does not accept the joining node, it will forward the subscribe message to other nodes in the tree to search for another parent. This prevents the joining node from knowing which nodes have refused to accept it as a child in the process. Therefore, we modified the join protocol so that the potential parents will always reply directly to the joining node whenever it refuses to accept the joining node, with a list of other possible parents. The joining node will then send subscribe messages to those nodes itself. While this would add a constant factor to the communication overhead, it is necessary for maintaining parental availability.

4.4.2 Experimental setup

All simulations are based on a peer-to-peer multicast system with 500 nodes. The simulations include *obedient* nodes that follow the prescribed protocol, and both choking and rejection freeloaders. The system is built on top of a transit-stub topology model generated by Georgia Tech random graph generator [ZCB96]. The model has 1050 routers, 50 of which are transit routers. End nodes are assigned to the routers with uniform probability. The link delays between the routers are computed by the graph generator. The delay on intranet links is set to 1 ms. No end nodes are attached to transit routers.

In all experiments, each node attempts to subscribe to all $k = 16$ trees, and each obedient

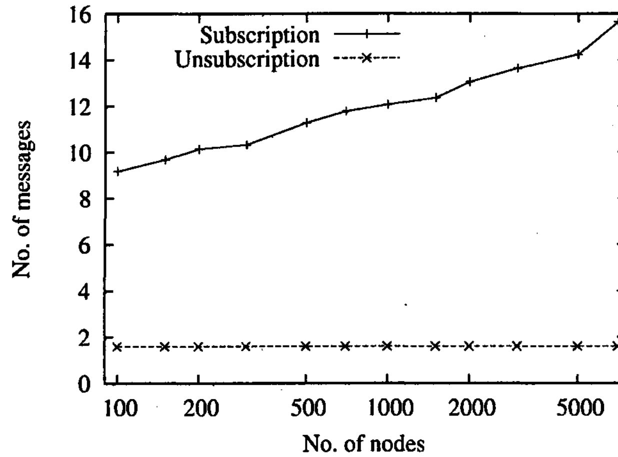


Figure 4.2: Average tree reconstruction cost for one tree.

node will accept up to 16 children. The multicast source acts as the root of each tree and will blindly accept up to 16 children. At every time step, the source transmits one "data unit" to each multicast tree and then all trees are reconstructed.

4.4.3 Tree reconstruction cost

Tree reconstruction would not be feasible if the overhead was prohibitively expensive. The first experiment measures the cost of reconstructing and discarding trees. Figure 4.2 shows the average number of messages sent by every node on a per tree basis. Since subscribing to a tree involves simply sending a subscribe message to a specific nodeId, the cost is proportional to the logarithmic of the number of nodes. These messages are very small in size, so the overhead is minimal relative to typical data rates for streaming video. To unsubscribe from a tree, a node only needs to notify its parent to stop forwarding data, therefore the cost is constant. Moreover, this cost can be saved if all the nodes simultaneously discard the tree after finished forwarding all data for the period.

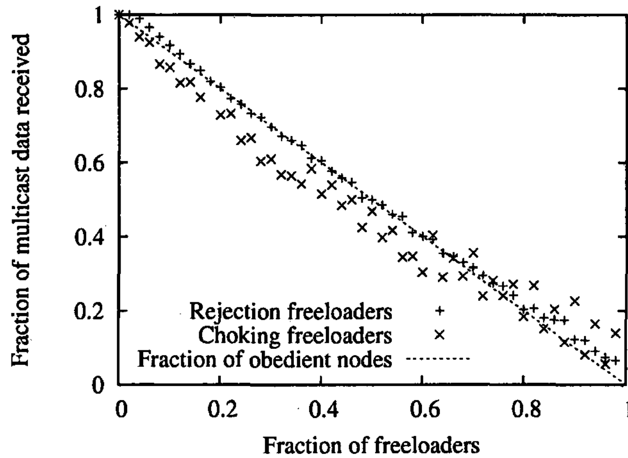


Figure 4.3: Fraction of multicast data received by obedient nodes with different types of freeloaders.

To estimate the overhead in practice, consider video streaming to 500 nodes. Assume that the video is streaming at 128Kbps, the typical upstream bandwidth for a DSL user. Figure 4.2 shows that on average each node needs to send 11.5 messages to reconstruct one tree. Assume that each message is 128 bytes and all 16 multicast trees are reconstructed every two minutes, the total overhead would only be 1.2% of the stream.

4.4.4 Fraction of freeloaders

Next experiment studies the effect of freeloaders to the system. The fraction of stream data received by obedient nodes, after 64 tree reconstructions, is shown in Figure 4.3. Observe that the fraction of data received by obedient nodes is roughly equal to the fraction of obedient nodes in the system. This is because each additional freeloader essentially takes away one share of the bandwidth without contributing any back.

Note that choking freeloaders hurt the system more than rejection freeloaders. This is because when a node is rejected by a freeloader, it can still look for another parent, while a

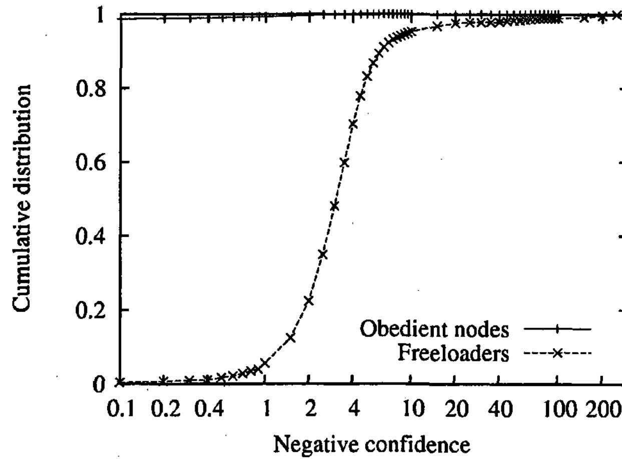


Figure 4.4: Cumulative distribution of negative confidence for obedient nodes and freeloaders.

node that is accepted by a choking freeloader will be stuck under the freeloader and receives no data.

4.4.5 SplitStream properties

In order to find suitable parameters for FairStream, we run the following experiments for 256 time steps to measure typical confidence and parent availability value distributions.

Confidence. This experiment consists of 5% choking freeloaders. Figure 4.4 shows the distribution of negative confidence. While obedient nodes rarely have negative confidence, around 90% of freeloaders have confidence values less than -4. It shows that the confidence values can effectively distinguish selfish nodes. For example, by setting a threshold of -0.22, a choking freeloader can be positively identified by more than 99% of nodes in the system with only 1% false positives.

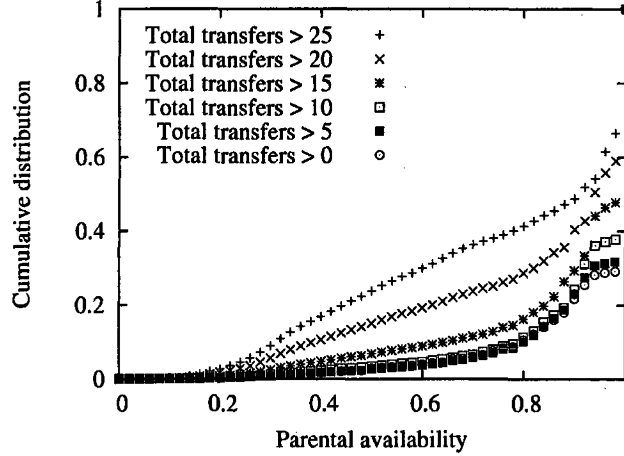
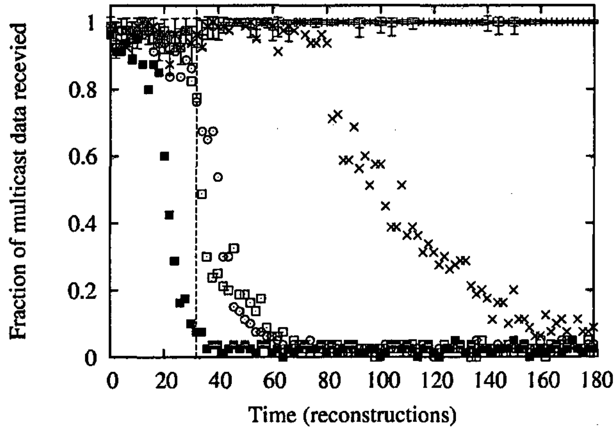


Figure 4.5: Cumulative distribution of parental availability with only obedient nodes.

Parental availability. In this experiment, we try to understand parental availability (a child's rating of how likely a given parent was to accept it as a child). The simulation consists of only obedient nodes. Figure 4.5 shows the distribution of parental availability when all nodes are willing to accept children. While most of the time the parental availability is quite high and more than 80%, it is becoming more common to have lower parental availability as the number of transfers increases. If a node was a rejection freeloader, its parent availability would be zero. To cut off parents with low availability, we must be careful to avoid false positives, particularly given that many legitimate parents have low ratings. For example, a cutoff of 0.8 might normally reject 10% of the legitimate parents.

4.4.6 Enforcing FairStream policies

Finally, the last experiments evaluate the effectiveness of enforcing FairStream policies. The following parameters are chosen based on empirical testing: $c_{+} = 0.1$, $c_{\text{th}} = -0.22$, $c_{\text{min}} = -0.32$, $w = 3.07$, $p_{\text{th}} = 0.37$, and $\lambda = 0.9$.



Type	Count	Strategy
+	480	Obedient
×	5	Rejection after time 32
⊡	5	Rejection from start
⊙	5	Choking after time 32
■	5	Choking from start

Figure 4.6: Fraction of multicast streams successfully received when applying FairStream mechanisms.

This experiment consists of 480 obedient nodes and 20 freeloaders. Ten of these freeloaders begin freeloading immediately while the other ten start freeloading only after 32 reconstructions. Half of freeloaders use the choking strategy, while the other half use the rejection strategy. The results are shown in Figure 4.6.

The figure shows that the fraction of data received by obedient nodes is always close to 100%, without being affected by freeloaders. The errorbars for obedient nodes show the standard deviations. The ranges are very small, which suggests that the variance experienced by obedient nodes is very low. Meanwhile, freeloaders cease to receive much data after they start to freeload. Note that this fraction never goes to 0, since there is a slim chance ($16/500 = 3.2\%$) that a freeloader can become a child of the source, which does not use any mechanism to track freeloading behaviors. It can also be observed that initial cooperation followed by freeloading behavior has only a limited effect in the short term and no effect in the long term.

The last experiment shows the effects of different fractions of freeloaders. The result,

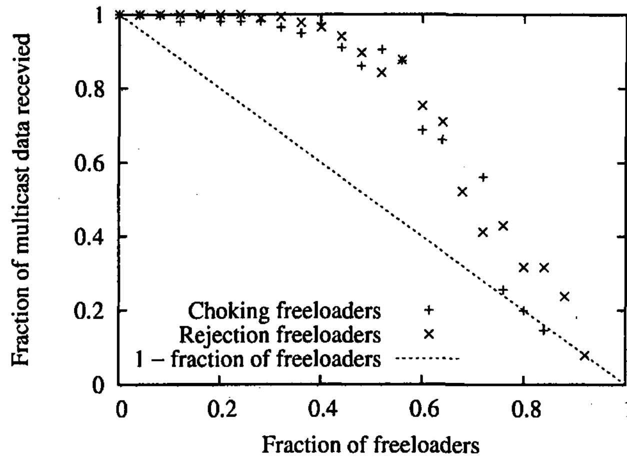


Figure 4.7: Fraction of multicast data received by obedient nodes with different fraction of freeloaders after applying my mechanisms.

after 64 time units, is shown in Figure 4.7. Comparing to Figure 4.3, the data received by obedient nodes is almost not affected when the system has less than 40% freeloaders. Beyond that, the fraction drops more quickly. Regardless, the fraction is still significantly higher than if the mechanisms were not used, up to the extent of 70% freeloaders.

4.5 Summary

This chapter describes the design and simulation results of FairStream, a system with fully decentralized mechanisms to incentivize cooperations in peer-to-peer multicast streaming. By regularly rebuilding multicast trees and having nodes to only track their first-hand observed behavior of their peers, nodes can easily distinguish freeloaders from well-behaved nodes. Based on that, nodes can deny service to freeloaders. Experimental results show that FairStream greatly reduces the quality of service received by freeloaders and improves that for well-behaved nodes. The network and computational overhead of

FairStream is low, making it practical to be deployed to current systems and make cooperative peer-to-peer applications more robust.

4.6 Related Work

Nicolosi and Mazières [NM04] proposed a technique for the sender of multicast data to confirm message delivery to all receivers. While this method can help the sender to learn the identity of nodes refusing to forward data, it does not prevent nodes from refusing to accept children.

Habib and Chuang [HC04] considered a model where a node makes a request and selects to receive services from a set of candidate suppliers. Since each individual node makes its own requests, it is similar to general file sharing. Moreover, they assumed a reliable peer-to-peer trust system, which is still an active research area.

Chu et al. [CGN+04] have deployed an operational Internet broadcast system based on overlay multicast, which has a peak user size of a few hundreds and provides reasonably good performance. However, since the content are usually conference/lecture-type broadcast, they are generally less susceptible to freeloading behaviors comparing to what might be in a general system.

Chu et al. [CCZ04] considered a taxation model, where resource-rich peers are required to contribute more bandwidth to the system to subsidize for the resource-poor peers. While their simulation showed that taxation can improve social welfare, it is unclear whether requiring the publisher to enforce taxation can effectively scale with the system size.

Chapter 5

Content Distribution Systems

5.1 Introduction

This chapter considers the incentive problem in the context of general content distribution systems. Unlike existing systems like BitTorrent [Coh03], where peers are interested in obtaining the same objects, a general content distribution system consists of both popular and unpopular objects. A fair policy, as used in BitTorrent, is "tit-for-tat," preferring to transmit content to other nodes who are willing to return the favor. This policy has the desirable property that no strategy can do any better than the population average [Axe81]. However, in this more general setting, a simultaneous swap of content is rarely possible.

This chapter describes a system called Scrivener. Scrivener encourages uploading content by allowing peers to accumulate pairwise credit that can be redeemed at a later time, for unrelated content from unrelated peers.

> **Design objective:** The more content a node serves to other nodes, the more easily it should be able to retrieve content from the system.

This work was done in collaboration with Animesh Nandi, Atul Singh, Peter Druschel,

and Dan S. Wallach, and was published in the ACM/IFIP/USENIX Sixth International Middleware Conference [NNS+05].

5.2 Background: BitTorrent

BitTorrent is a peer-to-peer file sharing communication protocol, tailored for distributing large amounts of data widely. In BitTorrent, users share a file or group of files via a *torrent* file. This torrent file contains metadata about the files to be shared and about the *tracker*, which is a central computer that acts as a rendezvous point. The set of peers that are interested in downloading a particular tracker is known as a *swarm*. Once a node has completely downloaded the file, it becomes a *seed* and uploads the file to all requesters.

BitTorrent provides incentives for users to upload content through a tit-for-tat strategy [Coh03]. Each BitTorrent node has a fixed number of uploading slots (default is four), and preferentially spends its uploading slots on peers which currently provide it with good downstream bandwidth. To discover if currently unused connections are better than the ones being used, a BitTorrent node also uses one uploading slot optimistically on other peers in the swarm in a round-robin fashion, regardless of its download rate.

Despite the measures BitTorrent has taken to induce incentives, more recent studies show that they are not strategyproof. First, BitTorrent nodes can download without uploading by collecting and connecting to more peers in the swarm [LMSW06]; sometimes they can even achieving better download rates than a compliant BitTorrent client [SPCY07]. Piatek et al. propose a strategic BitTorrent client called BitTyrant [PIA+07] that provides a

median 70% performance gain over a compliant BitTorrent client.

5.3 Goals and System Model

Scrivener is designed for cooperative content distribution systems where participants wish to obtain content stored on other participants' computers. Content is assumed to be published by its owner and disseminated into the system for distribution. Scrivener is designed under the assumption that, at least for popular objects, the owner has insufficient bandwidth to service every possible request and wishes to leverage the bandwidth available among other nodes in the system.

The set of participating nodes is assumed to form an overlay network. Scrivener is based on mechanisms that in principle can be applied to both unstructured [Gnu, Kaz] and structured overlay networks [RD01a, SMK+01], as long as they meet the following minimal requirements:

1. Each node in the overlay communicates directly with only a bounded (i.e., constant or logarithmic in the size of the overlay) number of overlay *neighbors*;

2. The overlay has a mechanism to discover new overlay neighbors; and

3. The overlay supports a search primitive that discovers, when given a valid content identifier, one or more overlay paths to a node that stores content associated with that identifier.

5.3.1 Goals

Scrivener's goal is to achieve fair sharing of bandwidth in content distribution systems. The key aspects of this goal are summarized below.

- *Fairness.* The system must ensure that participants receive a quality of service that is proportional to the amount of bandwidth they are actually contributing to the system. Furthermore, no participant should be permitted to perpetually consume resources in excess of their contributions at the expense of another participant. This provides an incentive for nodes to not freeload.

- *Low overhead.* The overhead imposed by the mechanisms used should be modest. Moreover, the marginal cost related to ensuring fairness when downloading an object should be low, to ensure efficiency despite small object sizes.

- *Robustness.* The system should retain the above properties even in the presence of large numbers of freeloaders and in the presence of modest churn.

5.4 Design

This section provides a more detailed description of Scrivener's design.

5.4.1 Relationships

Each Scrivener node maintains relationships with a small number of other nodes, typically its overlay neighbors, as selected by the overlay protocol. More precisely, any two

nodes in the overlay network form a relationship if and only if at least one of them has the other in its overlay neighbor table. A Scrivener node A grants a small initial confidence value (and thus a small credit limit) to any node that A has chosen as a neighbor, but it assigns an initial confidence of zero (and thus no credit) to any node that has invited A to be a neighbor. This prevents freeloaders from obtaining a large credit limit by initiating many relationships with many nodes, perhaps pretending that its normal neighbors have failed.[1]

The small initial credit limit allows neighbors chosen by A to request contents from A, and it allows A to request content from legitimate nodes who have chosen A as a neighbor. As contents are exchanged, the parties gain more confidence in each other and gradually grant each other larger credit limits. This scheme puts newcomers at a disadvantage; they need to initiate relationships, forcing them to grant credits and offer services while receiving little in return initially. This is the price for defending against freeloaders in any reputation-based system [FR01]. However, as below shows, the initial sacrifice is rewarded quickly as the node establishes confidence and gains credits with its neighbors.

When a Scrivener node A finds that one of its neighbors B has accumulated debt in excess of its credit limit, it ceases to accept requests from B. Regardless, A continues to make requests to B in order to give B the opportunity to pay back its debt. Likewise, A may find that the confidence value of one of its neighbors B has dropped to zero, perhaps because B has repeatedly failed to fulfill requests from A even though A is in good standing

[1]Overlay network systems are generally engineered to assume a high rate of node failure and include elaborate mechanisms to locate previously unknown nodes and form new relationships in order to preserve important invariants, including the degree of node-to-node connectivity and of file replication. As a result, Scrivener needs to limit the benefits automatically granted to a node solely because it happens to be a peer.

with B. In this case, A ceases to make requests via B or to accept requests from B. From A's perspective, B might as well not be a part of the overlay network. A then uses existing mechanisms provided by the overlay network to replace B with a different, and hopefully more cooperative, neighbor.

In principle, a Scrivener node must maintain a record of its past overlay neighbors indefinitely. Erasing a negative record would amount to forgiving debt, and would enable freeloading. In practice, it is acceptable to delete records of nodes that have been offline for long periods, perhaps a year, thus seriously inconveniencing freeloaders who wish to exploit the resulting loophole. Storing a year's worth of records is reasonable as these records are very compact: only a nodeId and two integer values, the credit and confidence values, are required. Such concise records could easily scale to track the millions of neighbors that a node might see in a year's time.

Note also that due to the pairwise relationships, freeloader cannot benefit from collusion. While colluding freeloaders may be able to convince legitimate nodes to shift credit from one freeloader to another, the total credit will remain unchanged.

5.4.2 Confidence

Scrivener nodes keep a confidence estimate for each of their overlay neighbors. The confidence value serves two purposes: (1) it determines the magnitude of the credit limit granted to a neighbor and (2) it can be used to bias overlay routing decisions towards cooperative neighbors.

The confidence assigned by a node to its neighbor is based on the history of their rela-

tionship. The confidence estimate has the following properties:

1. As nodes exchange content, the confidence increases slowly;
2. The confidence drops rapidly once a neighbor starts to misbehave; and
3. The confidence is bounded to limit the damage caused by a node that plays by the rules for an extended period and then starts to freeload.

An additive increase, multiplicative decrease (AIMD) strategy offers a simple implementation of these properties.

5.4.3 Transitive trade

In peer-to-peer content distribution systems with a large content set, the odds that a desired object can be found on an immediate overlay neighbor of the node wishing to fetch that object are small. In general, there is a need for nodes to trade their credits and debts with one another, preferably at the same time avoiding the overhead of digital cash or other cryptographic schemes. Scrivener employs an incremental trading strategy called *transitive trade*, which works by identifying a *credit path* from a source node to a node that has the desired object. In a credit path, each node in the path either has credit with the next node, or its debt is below the next node's credit limit. A scheme to locate such paths is described in Section 5.5.2.

Conceivably, once a credit path has been identified, it is possible to rearrange all the credits in the path such that the destination node now owes something not to its predecessor

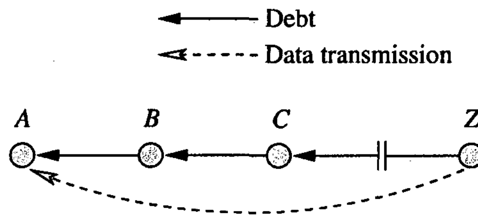


Figure 5.1: How a node can use a credit path to leverage a chain of credit to obtain content directly from a non-neighbor node.

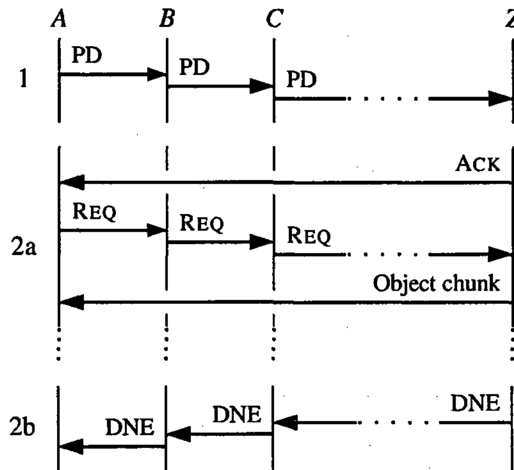


Figure 5.2: The stages in the transitive trade protocol.

in the route, but instead to the source of the route. This is illustrated in Figure 5.1. A series of debts, where *B* owes *A*, *C* owes *B*, and so forth until *Z* owes its predecessor could all be replaced with a direct debt from *Z* to *A*. *Z* can now cancel this debt by providing *A* with the desired content.

To make debt swapping work, a protocol that is robust against any node in the trading chain cheating is needed. For example, a node could attempt to cancel a debt that it owes without giving up the debt owed to it by the successor in the trading chain. Rather than resorting to a complex cryptographic commitment protocol, Scrivener takes a straightforward, incremental approach. The protocol is depicted in Figure 5.2.

1: Credit path discovery: A first routes a "path discovery" message (PD) towards Z. As a side effect, A "pays" B for this message, B pays C, and so forth until Z is paid. At the same time, each node reduces its confidence in its successor as if the request had failed (even though it may be working perfectly well). This design avoids the need to maintain timeouts to detect and react to failures. The credit path discovery might fail for a number of reasons, ranging from a freeloader dropping the message to network failures (see Section 5.5.2). The effect is that every node that forwarded the request will have reduced confidence in its successor. Furthermore, the last node in the chain effectively keeps the credit originally transferred from A.

2a: Object exists: Upon receiving the request, Z transmits a confirmation message (ACK) directly to A. A now routes a request message (REQ) for a chunk of the content object along the existing credit path, paying for the chunk as a side-effect of the message transmission. Z transmits the requested object chunk directly to A. A repeats this step until it has obtained the last chunk of the object. A final message, announcing A's success, causes each node to adjust the confidence value of its successor to compensate for the reduction in step (1), plus an additional confidence gained as a result of the trade.

2b: Object does not exist: Upon receiving the request, Z routes a "does not exist" message (DNE) along the reverse credit path. The message contains the addresses of the complete set of nodes that would store replicas of the content if it existed. Intermediate nodes can contact a member of this set to verify that the object does not exist.

If they are convinced that the object really does not exist, they restore the confidence of the successor node to compensate for the reduction taken in step (1).

Each participating node has an incentive to follow each of the protocol steps: Node A wants to receive all the chunks, node Z wants to be credited for transmitting all the chunks, and all nodes wish to maintain the confidence of their predecessors along the credit path. When a node defects from the protocol at some stage, it can collect credit without providing the corresponding service. However, the price is a drop in the confidence of the node's predecessor. Also, the damage is limited to the size of a single chunk, which can be made appropriately small.

In general, for any failure, the client A is charged for at most a single chunk — a modest loss. The charge can be interpreted as the price for imposing load on the overlay by issuing a request that could not be satisfied. Such a charge also discourages flooding requests into the system; the client must pay for each and every request it makes. The client can minimize the loss associated with a failure when it begins with a small chunk and gradually increases the request size as its confidence in the path increases.

Over the long term, transitive trading tends to balance credit and debt among a node's overlay neighbors, maximizing the chances that the node will be able to obtain content in the future. Moreover, participation in a transitive trade is beneficial because it increases the confidence of each node along the path in its successor.

At the same time, nodes have a disincentive to refuse participation in a transitive trade. Such a refusal leads the predecessor along the credit path to reduce its confidence in the

node. While the failure of a neighbor adversely affects a node, if it happens repeatedly, the node quickly reduces its confidence in that neighbor, and avoids routing messages through that neighbor in the future. As a result, failing nodes are avoided by the neighbors and become isolated.

It is important that nodes are not penalized for being offline. When a node is offline, other nodes merely suspend their relationship with the node until it returns. A related question is whether a node has an incentive to swap credit from an established neighbor to a newcomer as part of a transitive trade. In practice, having credit with a large and diverse set of neighbors maximizes the chances that a node will be able to successfully locate a credit path for a future request.

5.4.4 Caching

In general, objects in a content distribution system have a highly skewed popularity distribution [GDS+03]. To avoid load imbalances as a result of such skew, caching is used in these systems to dynamically adjust the number of nodes serving a content object according to its popularity. Typically, once a node has obtained some content for itself, it serves the content to other interested clients from its local cache. Thus, popular objects tend to be replicated widely.

In Scrivener, dynamic caching is required to address an additional form of imbalance caused by skewed popularity. Without caching, nodes serving popular objects would tend to accumulate a huge amount of credit. Nodes that serve less popular objects would tend to accumulate debt and lack the "earning potential" to ever repay the debt. Simulations

(see Section 5.6) will demonstrate this effect in action and show how caching addresses the problem. Moreover, nodes have an incentive to cache objects, because it increases their earning potential. Caching popular objects allows a node to earn the credit needed to satisfy its own future needs.

5.5 Implementation

This section describes an implementation of a prototype of Scrivener. It is implemented using FreePastry [Fre] with a distributed hash table service called PAST [RD01a, RD01b]. Scrivener uses only the key-based routing (KBR) API [DZD+03] exported by FreePastry. Thus, this implementation will also work with any structured overlay that supports this interface, e.g., Chord [SMK+01].

5.5.1 Node bootstrapping

Recall that when a new node joins the system, it has no credit or debt. To earn credit, it needs to obtain some initial content that it can serve to other nodes. In our implementation, PAST's normal content placement and replication policy provides a node with its initial set of content objects.

When a PAST node joins the system, it is *required* to store a set of objects based on its position in the identifier space. The node obtains these initial objects from its neighbors in the id space for free; they form the new node's initial content offering and allow it to acquire credit with its overlay neighbors, which forward requests for these objects to the

node as part of PAST's normal lookup operation. The simulation results show that this simple mechanism suffices for a node to quickly bootstrap itself.

5.5.2 Finding credit paths

A key implementation issue is how to efficiently discover credit paths. The Pastry routing primitive finds an overlay path to a node that stores the requested content object, given the object's identifier. Finding a credit path introduces the additional constraint that each node along the path must be in good standing with its successor.

We use a randomized, greedy algorithm to discover credit paths. To determine the next hop, a Scrivener node first selects the set of neighbors that satisfy the Pastry routing constraint. These nodes either have identifiers that match the requested object handle in a longer prefix than the present node's id, or their ids match as long a prefix as the present node's id but are numerically closer to the object handle. Forwarding the request to a node in this set guarantees that the route is loop-free and will end at a node that has the desired content, assuming the content exists in the overlay.

Next, any neighboring nodes where the present node is not in good standing is subtracted from the candidate set. These neighbors would refuse requests from the present node because it had exceeded its credit limit. Because all of the information used by nodes to rate their neighbors is available equally to both parties, nodes can easily track their standing with their neighbors.

Among the set of remaining candidate nodes, a biased random choice is made, based on the following criteria:

- *Length of the neighbor's prefix matches with the object handle.* Choosing a neighbor with higher prefix match than the present node reduces the latency and path length, and therefore also increases the chance to find a working path.

- *Confidence in the neighbor.* Neighbors with higher confidence values have been more helpful in the past, and are thus more likely to be helpful this time.

- *Amount of credit with the neighbor.* Choosing neighbors with higher credit helps the present node to balance credit and debt and therefore increases flexibility in handling future requests.

Scrivener strongly biases the forwarding choice toward neighbors with a prefix match (minimizing the number of overlay routing hops), while also trying to balance credit and debt, and gives preference to neighbors with high confidence values. More precisely, let $\mathscr{C}$ denote the set of candidate nodes. Scrivener assigns a *score* to each node x in $\mathscr{C}$, which is calculated as $\text{score}(x) = e^{\ell(x)} \times t(x) \times [c(x) - c_{\min} + 1]$, where $\ell(x) \geq 0$ is the number of additional digits that the neighbor x shares with the object handle relative to the present node, $c(x)$ and $t(x)$ are the credit and confidence value of neighbor x, and $c_{\min} = \min_{i \in \mathscr{C}} c(i)$. Then the probability that peer x is chosen ($p(x)$) is its score divided by the total score of all candidate peers, i.e., $p(x) = \text{score}(x) / \sum_{i \in \mathscr{C}} \text{score}(i)$. The quality of a node's prefix match figures exponentially in its score to give a significantly greater weight to shorter routes. Note also that both confidence and credit/debt are measured in the same units, i.e., the number of objects or bytes transferred.

This randomized, greedy algorithm is not guaranteed to discover a credit path even

if one exists. A request could end up at a node that has no neighbor that satisfies the Pastry routing constraints and with which the node is in good standing. In such case, the request cannot be forwarded and the client will need to retry the request through a different neighbor.

Our simulations shows that the success rate of finding a credit path is very high and the number of retries typically necessary to discover a credit path is very low in practice. There are several reasons for this. First, the Pastry overlay is richly connected and many redundant paths exist between a client and a node holding the required content. Second, dynamic caching effectively balances the "earning power" of nodes, avoiding strong imbalances in the credit available to different nodes. Third, the bias in the forwarding policy against nodes with low confidence tends to isolate freeloaders, causing requests to be effectively routed around such nodes. Lastly, the bias in the forwarding policy based on credit tends to balance the available credit a node has with its different neighbors. These various self-stabilizing forces reduce the probability that a credit path search might fail, either due to lack of credit or because a freeloader refuses to honor it.

5.5.3 Bounding lengths of credit paths

Unlike the native Pastry routing policy, Scrivener does not always choose a neighbor with a longer prefix match, even if such a neighbor exists. As a result, Pastry's logarithmic bound on the expected path lengths does not strictly hold. Note that shorter path lengths are desirable for two important reasons: (1) shorter path lengths ensure low delay and network utilization, and (2) shorter paths are more robust against node failures. Since the routing

policy of Scrivener may occasionally lead to long paths, it is resort to another mechanism to bound the path length.

In our implementation, Scrivener artificially bounds the credit path length to be logarithmic in the overlay size. When the search for a credit path has reached this bound, the request is dropped. A rough estimate of the size of the overlay N suffices to determine the bound. Since nodeIds are assigned at random, the overlay size can be extrapolated from the local density of nodeIds with sufficient accuracy. When a search exceeds this boundary, the request is dropped. Our simulation results, presented in Section 5.6, show that the impact of this restriction on the ability to locate credit paths is minimal, while it ensures deterministic bounds on the system's resource consumption.

5.6 Experimental Results

This section presents simulation results to evaluate our implementation. In the simulation, network messages are delivered instantaneously. Objects are replicated using PAST's replication strategy, storing an object on the k nodes with nodeIds closest to the identifier for that object. When requesting an object, client nodes perform at most 10 queries, each time attempting to discover a credit path using the randomized greedy algorithm. The initial credit limit is set to one object, and increases linearly with the confidence the node has in its peer. The credit paths are limited to $\lceil 3 \log N \rceil$ hops. Each node also has a fixed sized, 1024-object soft cache to retain objects it has previously obtained to satisfy future requests. A least recently used (LRU) cache replacement policy is implemented to replace entries

from the cache when it is full.

A node's peers maintain their credit and confidence values for a node that is temporarily offline. Also, the Pastry routing tables are persistent, i.e., a node remembers its table while it is offline. Inappropriate entries are simply replaced by the existing overlay maintenance mechanisms, but biased towards peers with which the node already has a relationship. As a last resort, the node initiates a new relationship. Also, for each entry in the routing table, a node maintains at most three neighbors but uses only the one with the highest confidence value. (Confidence estimation is described in Section 5.4.2.)

5.6.1 Workload model

The workload model is generated according to Gummadi et al. [GDS+03]. This model, derived from Kazaa traffic observations, captures the fetch-at-most-once behavior and the importance of new object arrivals in typical peer-to-peer file sharing applications. Based on this model, we choose the following parameters: Number of nodes online $C = 800$, number of objects $O = 40,000$, request rate per node $\lambda_R = 50$, object arrival rate $\lambda_O = 12$, and node arrival rate $\lambda_C = 5$ (the units are nodes or objects per simulation time unit). The node departure rate is the same as the arrival rate, keeping the number of active nodes constant. Each object is initially replicated to $k = 3$ nodes. There is a fixed pool of 1,000 distinct nodes, out of which 800 are online at any time. As a result, during the first 40 time units all arriving nodes are fresh, but after time 40 all arriving nodes are those that were online once before. Nodes that go offline are chosen randomly from the currently live nodes.

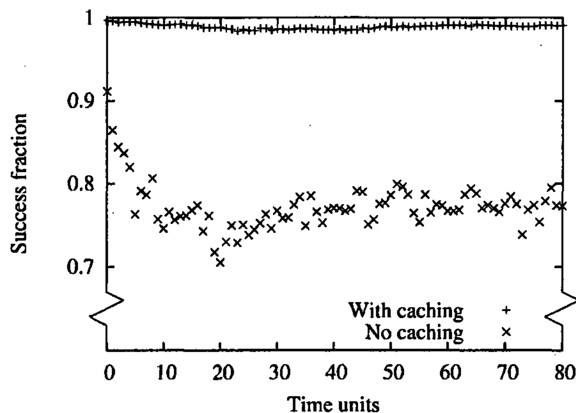


Figure 5.3: Success rate with only obedient nodes.

5.6.2 System performance

The first experiment studies how the mechanisms affect the performance of the underlying cooperative content distribution system in the absence of freeloaders. In particular, it investigates how much overhead is added to the system.

Success rate. Figure 5.3 shows the fraction of successful requests, both with and without caching. Without caching, the success rate stabilizes around 80%. This is because object popularity is so uneven that nodes around the replicas of popular objects become indebted to the replica holders, making it sometimes impossible for a node to find a credit path to the replicas. Many requests to popular objects fail despite retries. However, allowing nodes to serve cached objects eliminates this problem and the success rate approaches 100%. The stability of the success rate suggests that the system balances out nicely and obedient nodes do not build up debt over time.[2]

[2]Another experiment makes use of *speculative caching*, where nodes observe the requests they have forwarded and actively fetch objects that they consider popular. However, the improvements observed in terms of success rate were insignificant.

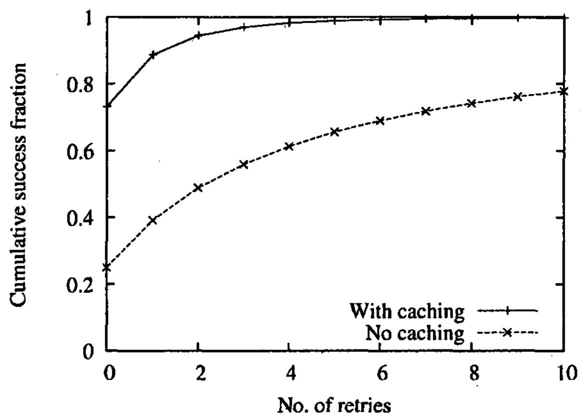


Figure 5.4: Cumulative distribution of the number of retries to find a debt-based path.

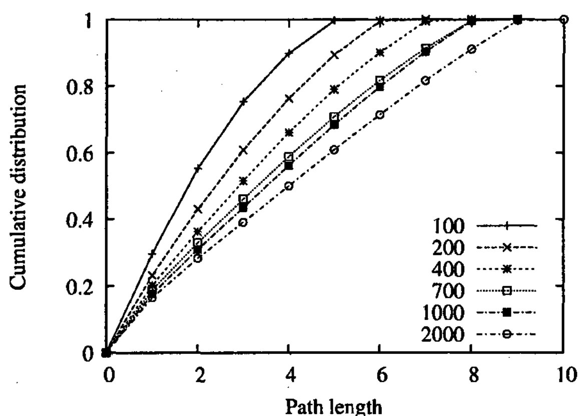


Figure 5.5: Cumulative distribution of debt-based path lengths for different system sizes.

Figure 5.4 shows the number of retries required to successfully find a credit path. When caching is enabled, over 73% of queries succeed on the first attempt, and three attempts are sufficient to achieve over 95% success rate. We conclude that the policy enforcement in Scrivener with bounded paths does not seriously affect object fetch reliability in the absence of freeloaders.

Path efficiency. Scrivener's randomized greedy routing strategy attempts to use Pastry's routing mechanism to achieve logarithmic-length paths, when possible, and falls back

to less efficient mechanisms, when necessary, that are artificially capped to preserve an $O(\log N)$ expected path length (see Section 5.5.3). A cumulative distribution of path lengths at different overlay sizes is shown in Figure 5.5. By observing horizontal slices through this graph, the growth in path length follows roughly the log of the number of nodes. The simulations show that common case routes are quite efficient and the worst case routes are only twice as long as common-case routes.

Due to limitations of the simulation environment, we cannot run simulations for overlay sizes larger than 2000. In order to emulate the effect of larger overlay sizes, simulations with 1000 nodes are used, but with Pastry's routing base set to $b = 2$ instead of 4. The results show that the median Scrivener path lengths is around 5, close to the expected Pastry path length ($\log_{2^2} 1000 \approx 4.98$). Note that when $b = 4$, the expected path length for a Pastry overlay with one million nodes is 5. given destination. This result suggests that Scrivener's greedy routing strategy easily scales to much larger overlay sizes than it was simulated.

These longer paths, which also occur as the number of nodes in the overlay increases, raise concerns about path usability, particularly if the system is experiencing high node churn. More nodes in a path increases the odds that one of these nodes will fail while a transitive trade is in progress. However, the system provides incentives for nodes to stay online until a transitive trade in which they are involved completes (see Section 5.4.3). If a path fails, the original requesting node can restart the trading protocol, find a new path to the source of the data (or a replica), and resume downloading the missing data.

The total overhead for Scrivener to fetch an object is the product of the average number of attempts to discover a credit path (≈ 2) and the average credit path length ($< \lceil 3 \log N \rceil$). Among competing systems that use auditor sets, KARMA [VCS03] is the most efficient known system. KARMA's asymptotic message overhead is comparable to Scrivener's, but requires expensive public-key cryptographic operations and additional means of incentivizing auditors.

Introducing freeloaders. Next, we introduce freeloaders into the simulations. Freeloaders issue requests like obedient nodes, but they may refuse to serve objects. In a deployed system, freeloaders can be expected to attempt a variety of strategies. We consider a number of freeloading strategies, and show that in all cases there are no sustainable benefits to freeloading. The simulations consist of 800 nodes with 5% freeloaders. We assume that freeloaders forward requests and participate in transitive trades, as this allows them to earn confidence with minimal traffic overhead. While obedient nodes undergo churn as specified in the model, freeloaders are always online throughout the entire simulation period. Recall that routing tables are persistent, ensuring that freeloaders cannot neither escape a bad reputation by periodically departing from the system nor by repeatedly exploiting the limited credit granted by obedient nodes looking to establish relationships.

Freeloaders that never serve. The first experiment considers freeloaders that never serve any object. Figure 5.6 shows that their success rate drops to below 5% within a few time units, yet that of obedient nodes is unaffected. Note that the success rate for freeloaders

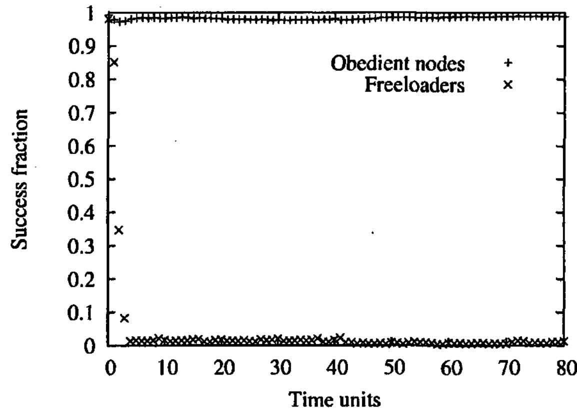


Figure 5.6: Success rate with 5% freeloaders that do not serve objects.

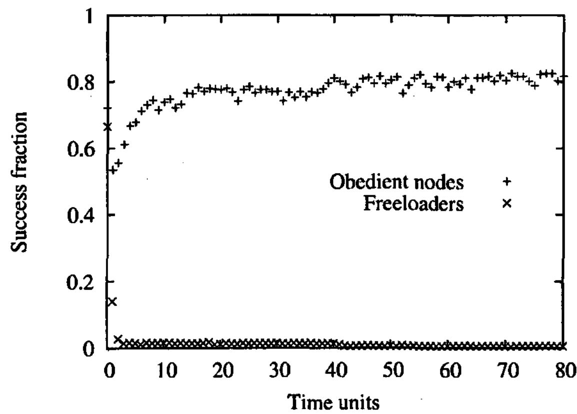


Figure 5.7: Success rate with 50% freeloaders that do not serve objects.

never goes to zero. This is because freeloaders can still get the objects that they themselves are storing "for free."

To determine Scrivener's sensitivity to the size of the soft cache, the cache size is reduced. The success rate remains virtually constant down to a cache size of 320 objects, and gradually decreases to 91% at 128 objects. This shows that Scrivener does not require a large soft cache to work efficiently.

The next experiment increased the fraction of freeloaders to 50%, with results shown in Figure 5.7. The success rate of freeloaders again drops quickly to near zero, while that

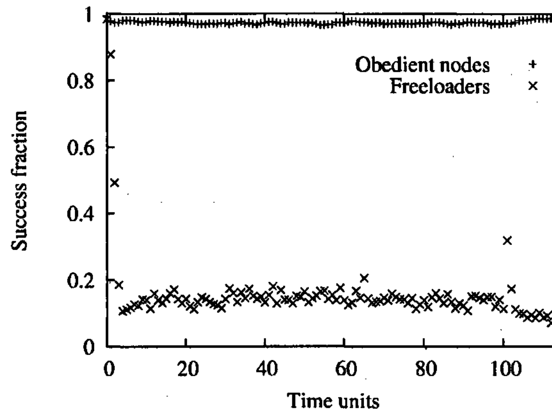


Figure 5.8: Success rate with a higher churn rate.

for obedient nodes starts below 60% and plateaus at 80%. Note that with 50% freeloaders and a replication factor $k = 3$, it is expected that 12.5% of the objects are only stored by freeloaders and will thus never be served. This suggests that a more expensive search may increase the success rate somewhat, but with diminishing returns.

To test the system under extreme conditions, we further increased the fraction of freeloaders to 80%. At this point, more than half of the objects are stored only by freeloaders and, unsurprisingly, the success rate for obedient nodes is only 30%. Also, as a result of more transitive trading failures, it takes longer for the success rate of obedient nodes to stabilize. Scrivener does continue to function remarkably well, despite the extreme freeloading rate. Given that these freeloaders receive no benefit from being present in the network, one would expect them to depart, allowing the remaining obedient nodes to operate more efficiently.

Since it takes time for obedient nodes to recognize freeloaders, one concern is that a high churn rate might enable freeloaders to get a satisfactory success rate by exploiting

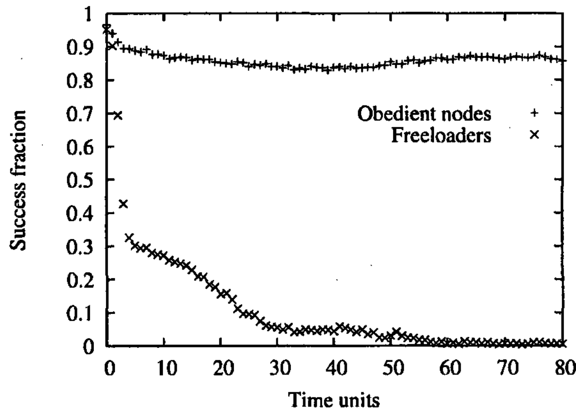


Figure 5.9: Success rate with the worst-case scenario where every obedient node gives a high initial confidence to all freeloaders.

newly arrived nodes. To simulate that effect, we increased the churn rate λ_C to 50 nodes per time unit and with fresh nodes arriving for the first 100 time units. After time 100, the arriving nodes have all previously been part of the network and gone offline. Figure 5.8 clearly shows that with this higher churn of fresh nodes, the success rate for freeloaders stabilizes at around 15%, dropping after time 100 when the returning nodes remember previous freeloaders. Thus, while freeloaders can exploit newcomers, the benefit is limited. More importantly, the success rate for obedient nodes is unaffected. While obedient nodes waste some effort handling requests from freeloaders, they give clear priority to serving each other.

Recall that a Scrivener node grants an initial credit to its chosen neighbors. The next experiment considers an attack where a freeloader somehow convinces an obedient node to choose it as a neighbor, thus granting it an initial credit. It considers a worst-case scenario where freeloaders can always manipulate obedient nodes into choosing them as neighbors. With such an attack, freeloaders could now exploit the initial credit from each obedient

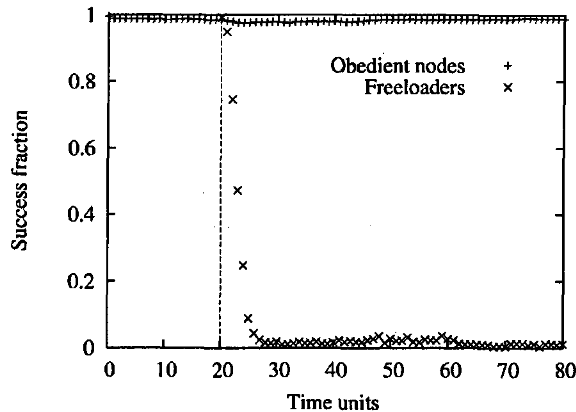


Figure 5.10: Success rate with freeloaders that participates in transitive trades but do not fetch objects for the first 20 time units.

node. Figure 5.9 shows that, indeed, freeloaders get a better success rate initially. However, the success rate drops to 30% quickly and gradually goes down as obedient nodes refuse to serve freeloaders after their debts built up. This simulation shows that, even with such a hypothetical attack, freeloaders would have little benefit and obedient nodes would observe no significant change in their own success rate.

Short-term cooperation. Participation in transitive trades, alone, can earn confidence and increase credit limits without actually serving any object. An interesting question is whether it is possible for freeloaders to build up confidence simply by participating in transitive trades, and then exploit that confidence. Figure 5.10 shows a simulation where freeloaders participate in transitive trades for 20 time units before fetching any object. The success rate for freeloaders drops to below 0.1 within ten time units. Thus, participation in transitive trades does have a benefit, but only a small one.

The next simulation shows nodes that were obedient for 20 time units and then began

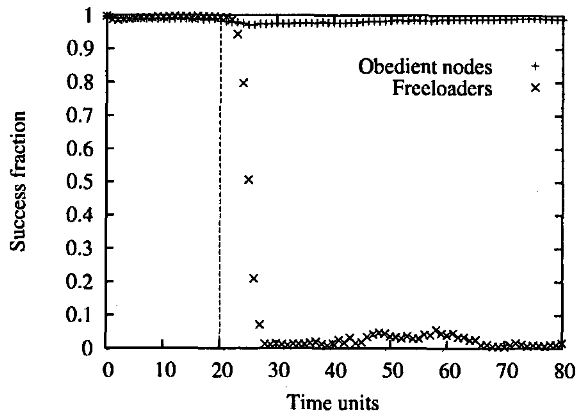


Figure 5.11: Success rate with freeloaders that serve objects only for the first 20 time units.

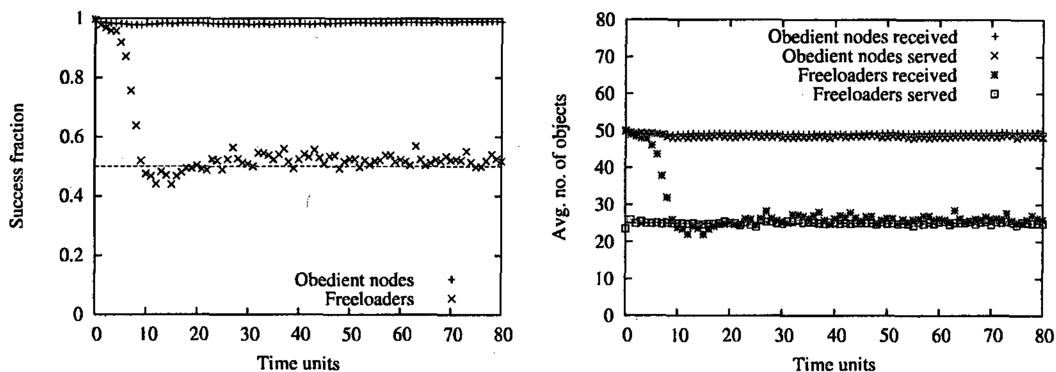


Figure 5.12: Number of objects served and fetched with freeloaders that serve half of the object requests.

freeloading. As shown in Figure 5.11, the freeloader's success rate now takes seven time units to drop below 0.1. The freeloader does benefit from its earlier obedience. However, once freeloading behavior begins, the success rate remains high for only two time units, then falls quickly.

These experiments demonstrate that short-term cooperation is not an effective strategy for freeloaders to exploit the system; once they start to freeload, obedient nodes will quickly refuse to serve them.

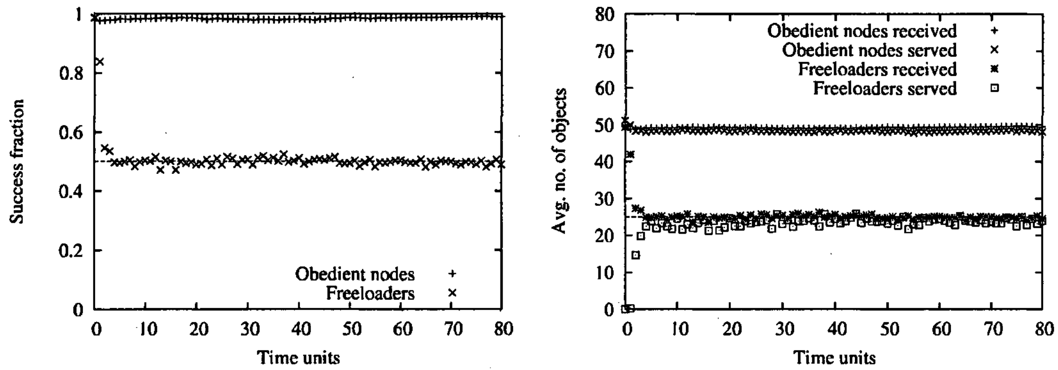


Figure 5.13: Success rate and number of objects served and fetched with freeloaders that aim at 50% success rate.

Providing partial service. Another possible freeloading behavior is to serve objects at a reduced rate. The first experiment considers freeloaders that arbitrarily serve half of their requests. Figure 5.12 shows that the success rate for freeloaders drops to and remains at roughly 50% — the same rate at which they are providing service. Note also that the number of objects received by freeloaders also approaches and stabilizes at the same level as the number they serve.

Another potential strategy is to have a target quality of service. This freeloading behavior serves only enough requests to maintain a desired success ratio. The simulation considers freeloaders that target a 50% success rate. Figure 5.13 shows that the resulting success rate oscillates around 50%. As before, the number of objects served by the freeloader quickly dictates the number of objects the freeloader is allowed to consume.

Finally, this experiment considers a strategy that alternates between obedience and freeloading, changing behaviors every 20 time units. Figure 5.14 shows that the success ratio quickly tends toward 1 and 0 whenever these nodes switch to cooperation and to freeloading, respectively, with the peak success ratio dropping over time. Also, during the

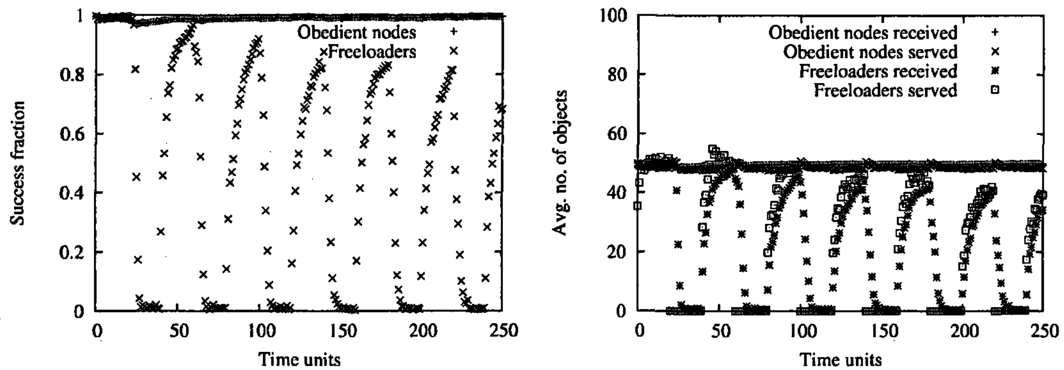


Figure 5.14: Success rate and number of objects served and fetched with freeloaders that switch between cooperation and freeloading every 20 time units.

cooperation periods, the former freeloaders service more requests, effectively making up for the debts they previously accumulated. On average, this alternation strategy performs worse, from the freeloader's perspective, than the previous 50% service strategy.

Other experiments. In this simulation, a node requests 50 objects per time unit. If each object is 64 Kbytes, this translates into roughly 3MB of data per time unit — about the size of a typical MP3 file or digital photograph. If users attempt to download 100MB of data per day, their success rate would drop to zero in about an hour. Increasing the download rate does not help, since its merely accelerates the decline in success rate.

To test Scrivener's sensitivity to the size of the downloaded content, the next experiment divides large objects into smaller chunks that were stored and downloaded separately. The success rate of obedient nodes improved relative to the earlier experiments. When downloading smaller chunks, smaller credits were necessary, increasing the success rate of transitive trading. Also of note, freeloaders experienced an even lower success rate. Because a desired object may now be spread over several chunks, the odds of successfully

obtaining all of a file's chunks diminished. Of course, breaking a file into chunks will increase the overhead rate, as each chunk will need to be separately located and fetched.

Simulations are also carried out where obedient nodes have diverse bandwidth capacities. The success rate for both types of nodes are very close to 100%, although the success rate for high-end nodes drops slightly. This shows that Scrivener can accommodate modest imbalances in the demands and "earning potentials" of participating nodes gracefully. Other approaches, including treating a high-end node as several virtual nodes, may also be applicable.

5.7 Summary

This chapter evaluates mechanisms to make bandwidth-limited peer-to-peer content distribution networks robust against freeloaders. Obedient nodes experience modest additional overhead, and over a variety of freeloading behaviors, freeloaders achieve only the level of service that they are willing to provide to others in the network, even for large numbers of freeloaders in the system. The simulations demonstrate that the obedient strategy maximizes a node's service received.

5.8 Related Work

SLIC [SGM04] considered the query nature of unstructured peer-to-peer systems like Gnutella [Gnu]. It proposed giving nodes service levels proportional to their contribution, so as to provide nodes incentives to share more data and handle more traffic. BitTor-

rent [Coh03] facilitates large numbers of nodes all trying to acquire exactly the same file, with an emphasis on very large files (e.g., software distributions, digital movies, and so forth). Every BitTorrent node will have acquired some subset of the file and will trade blocks with other nodes until it has the whole file. In order to bootstrap new nodes, nodes reserve one-fourth of their bandwidth for altruistic service. Nodes that fairly trade their bandwidth will experience a higher quality of service. Anagnostakis and Greenwald [AG04] suggested that performance can be improved if exchanges are extended to allow multiple parties involvement. Scrivener solves the more general problem, where nodes are interested in more diversified contents of potentially much smaller sizes. Scrivener allows nodes to acquire credits from the files they serve to obtain any other files they desire in the future. Thus, they have an incentive to serve, even when they themselves do not require any content at the moment.

GNUNET [Gro03] used the idea of locally-maintained debit/credit relations in a similar fashion to Scrivener. It also used debt relationships across nodes, comparable to the debt-based routing. As GNUNET is more concerned with anonymity than network efficiency, it does not support transmitting objects directly across the network. All traffic goes through the overlay, forcing intermediate nodes to carry the bulk traffic of the object transfer while giving them no particular incentive to do this, save for maintaining their own anonymity. For a path with N nodes, GNUNET transfers the object $O(N)$ times. Scrivener, on the other hand, finds efficient routes and transmits bulk data directly over the Internet, yielding higher performance, but lacking GNUNET's anonymity features. Scrivener also provides

mechanism to locate and fetch objects, leveraging its existing credit/debit framework.

This chapter is particularly relevant to Axelrod's study [Axe81] on cooperations emerged under selfish individual users' interaction. Under certain assumptions, if everyone is using the reciprocal cooperative strategy of tit-for-tat, no strategy can do any better than the population average. Moreover, his study discusses how cooperation can emerge from a small cluster of discriminating individuals even when everyone else is using a strategy of unconditional defection. This explains why our proposed system can converge even when most users may be selfish and not fully cooperative.

Chapter 6

Anonymous Communication Systems

Distributed anonymous communication networks depend on volunteers to donate resources to relay traffic. In the case of Tor [DMS04], one of the most popular and widely used anonymity systems, the efforts of volunteers have not grown as fast as the demands on Tor. This increasingly disparity is limiting the system's performance. This chapter explores techniques to incentivize Tor users to establish Tor relays through measuring performance of Tor relays by the central directory authorities.

> **Design objective:** Users that provide more relay bandwidth should receive better service in return, both in terms of anonymity and performance.

This work was in collaboration with Roger Dingledine and Dan S. Wallach.

6.1 Introduction

Anonymizing networks such as Tor [DMS04] and Mixminion [DDM03] aim to provide protection from traffic analysis on the Internet. Traffic analysis focuses on who is communicating with whom, which users are using which websites, and so on. These networks work by bouncing traffic around a network of relays operated around the world, and strong

security comes from having a large and diverse network. To this end, Tor has built a community of volunteer relay operators. It suffers if too few people choose to operate relays to support the network's traffic.

In fact, Tor is heading in exactly this direction. The number of users keeps growing, while a variety of factors discourage more people from setting up relays; some want to save their bandwidth for their own use, some cannot be bothered to configure port forwarding on their firewall, and some worry about the possible consequences from running a relay. This growing user-to-relay ratio in turn hurts the service received by all users.

Worse, not all users are equal; while Tor was designed for web browsing, instant messaging, and other low-bandwidth communication, an increasing number of Internet users are looking for ways to anonymize high-volume communications. An informal measurement study, performed by running a Tor exit relay, found that the median connection coming out of the relay looked like HTTP traffic, but the median *byte* distribution looked like file-sharing traffic.

This issue is worsening over time. Because of the threat of legal action from the entertainment industry, some users of peer-to-peer file-sharing applications are starting to tunnel their traffic through Tor. The Azureus BitTorrent client, one of the most popular BitTorrent clients, has built-in support for using Tor. Even though the default Tor exit policy rejects the default BitTorrent ports, enough users are using non-standard ports for their file-sharing that this additional load on an already overloaded network makes the service bad for all users.

The most straightforward way to attract more people to run relays is to provide them with better service. Tor users care and value privacy and anonymity, and better anonymity means better service. Anonymity aside, most users would prefer a faster network, so performance (available bandwidth) is another important metric for service evaluation.

This chapter proposes a solution for Tor where the central directory authorities measure the performance of individual relays and use this information to decide the level of service one can get. This is made possible by constructing multiple disjoint Tor networks and by differentiated traffic treatment. Through simulation, this design is shown to improve the service for cooperative relays, even as traffic from other users increases. This approach incentivizes end users to establish new Tor relays, improving Tor for everybody.

6.2 Background

The Tor network is an overlay network of volunteers running *Tor relays* that relay TCP streams for *Tor clients*. Tor aims to let its users connect to Internet destinations like websites while making it hard for (1) an attacker on the client side to learn the intended destination, (2) an attacker on the destination side to learn the client's location, and (3) any small group of relays to link the client to her destinations.

To connect to a destination website or other service via Tor, the client software incrementally creates a private pathway or *circuit* of encrypted connections through several Tor relays, negotiating a separate set of encryption keys for each hop along the circuit. The circuit is extended one hop at a time, and each relay along the way knows only the immedi-

ately previous and following relay in the circuit, so no single Tor relay knows the complete path that each fixed-sized data packet (or *cell*) will take. Thus, neither an eavesdropper nor a compromised relay can see both the connection's source and destination. Clients periodically rotate to a new circuit, to complicate long-term linkability between different actions by a single user.

The client learns which relays it can use by fetching a signed list of Tor relays from one of the *directory authorities*. Each authority lists the available relays along with a set of opinions or recommendations for each. Clients make their decisions based on the authority opinions. A more detailed description of the Tor design can be found in its original design document [DMS04] and its specifications [DM].

6.3 Design

This section discusses the possible design space and our proposed solution.

6.3.1 Design space

Below is a list of design options we have considered.

Measurement methods We need a way to find out which users are contributing. There are three ways to make measurements:

(1) *Individual measurements:* If each node kept its observations strictly to itself (as with BitTorrent's tit-for-tat measurements), then a given relay node would only be aware of a few peers' current behavior. Stale or absent knowledge of remote peers' behavior might then

lead to incorrect decisions on whether to prioritize those nodes' traffic. It might also lead to *partitioning attacks* [DDM03]; when users are not all acting on the same information and being given the same treatment, an observer may be able to distinguish one user from others. An active attacker can even manipulate network views to induce these attacks.

(2) *Distributed measurements:* This approach allows relays to report their own observations about other relays to the directory authorities. The directory authorities can for example use the median vote [SB08]. However, this requires relays to reveal sensitive data that a compromised directory authority may use to deanonymize traffic.

(3) *Central measurements:* Since Tor already has globally trusted directory authorities, we can leverage them to actively measure the performance of each individual relay. By measuring through the Tor network itself, the directory authorities can hide their identity and intent from the Tor relays. This method of anonymously auditing nodes' behavior is similarly used in other systems [DS02, NWD03, SNDW06].

Network size The anonymity for Tor and similar systems comes from "blending into a crowd" [RR98]. The core idea is that an observer cannot pinpoint the origin and destination of traffic if it is as likely to be any one in a crowd. One way to measure the degree of anonymity one enjoys is k-anonymity [Swe02], where k is the size of the crowd. The k-anonymity increases with the size of the network. All other things being equal, users generally prefer to join a larger network.

Performance improvement A direct incentive for contribution is to reward good users with better performance. Contribution can be measured in the form of bandwidth and/or latency of the relayed traffic for that relay. The tricky part is to decide which users to be rewarded, since tracking users and keeping statistics can introduce new anonymity attacks. After all, anonymizing networks are specifically designed to make it hard to identify the origin of a connection, so any sort of accounting schemes seems to be at odds with preserving anonymity. If we rely on Tor users to report their experience, they could indirectly reveal the circuits they used, aiding attacks on anonymity. If we ask the relays to report their experience, they might strategically lie about their results [ARS+08], or they might reveal information that could violate users' anonymity. Any use of "hearsay" evidence that cannot be validated is an opportunity for fraud. For example, if saying good things about a peer can increase its reputation, then we now have an incentive for Sybil attacks [Dou02], creating an army of nodes whose purpose is to speak admiringly of a given node to improve its reputation.

Differentiated treatment suffers from a major drawback: It reduces the *k*-anonymity of the users. In particular, to provide special treatment, traffic from those users needs to be marked differently. If only a small fraction of users is treated preferably, this would reduce their levels of anonymity, which contradicts to our goal of rewarding them.

Maintaining and publishing "debts" Scrivener in Chapter 5 uses no central authorities, and instead relies on nodes maintaining their relative bandwidth debts and credits, which are then used to identify paths in the "debt space." These debt paths are an essential way to

overcome the otherwise limited direct relationships that may be observed between nodes. Publishing data like this would be devastating for anonymity as it would allow observers to piece together Tor's data circuits, piece by piece, by observing the bandwidth debts changing in synchrony from one relay node to the next.

Pairwise relations Another option is for nodes to directly measure their peers' performance in a fashion analogous to the tit-for-tat trading strategies used in BitTorrent [Coh03] or the peer auditing in Chapter 5. However, a Tor relay may have information on only a limited number of other relays, so any benefit from this approach may also be limited.

Electronic cash Another alternative is an anonymous digital cash scheme where relays earn cash for relaying traffic from users, but there are still traffic analysis attacks when users go to the bank to deposit or withdraw coins (these attacks may be done by an observer or also by a colluding bank). There would also be a need for a secondary protocol for resolving disputes when one side fails to hold up its end of the bargain.

Leverage social network A final method is to leverage social networks' trust relationships, which have been used in a variety of past peer-to-peer systems to improve robustness (see, e.g., SPROUT [MGGM04] and SybilGuard [YKGF06]). Unfortunately, if any reputation system included a mechanism for relays to determine that they are friends with the originator of the circuit, those mechanisms could be leveraged to attack users' anonymity. Using reputation systems without compromising anonymity may be possible, but it would be difficult to do properly.

6.3.2 The proposed design

Out of the aforementioned design choices, the most effective one is to ensure contributing users to have better anonymity by assigning them to larger networks. As a complement solution, we can also improve their performance, but this must be done carefully so as to not reduce their anonymity. The rest of this section describes our design.

We employ central measurements on the network, i.e., we use central directory authorities to perform measurements on each individual relays. This gives us an idea on how much bandwidth each relay is contributing to the system. These results are then used to decide the level of service received by each relay, through running a premium version of Tor and assigning gold stars.

Part I: Premium Tor

First, we improve the k-anonymity of contributing users by allowing them to join a larger network. The Tor administrators can create two disjoint Tor networks, one is called *premium* Tor, and the other one called *probational* Tor. All new users first join the probational Tor. Relays with acceptable performance would receive signed permission from the central authorities, which would allow them to join the premium Tor. Users in premium Tor must still provide satisfactory level of service to stay.

To guarantee that the users in the premium Tor have better anonymity than their counterparts in the probational Tor, the performance threshold must be set to be reasonably low. This means we probably need to promote all users who are running relays at a minimal

bandwidth.

With this setting, most users, except the few freeloaders, would be able to join the premium Tor. The premium Tor should be comparatively larger in size, thus providing a better k-anonymity. If there are still too many relays with unacceptable performance, the directory authorities can also break the probational Tor into several smaller ones, in a sense to punish those in the probational Tor with artificially poorer anonymity.

Note that because the premium Tor consists of mostly cooperative relays and no freeloaders, the performance of its relays should naturally be better already. This would also encourage most users to run relays, just so that they can join the premium Tor and enjoy the better anonymity and service. If performance remains as a concern, the directory authorities can opt to enable the second part of the design: Assigning gold stars.

Part II: Assigning gold stars

The second part attempts to further improve the performance for the more cooperative users through priority treatment to their connections, thus providing an incentive for users to run faster relays. Note that to provide proper treatment for traffic from different relays, an intermediate relay does not need to know the identity of the origin; in fact, it suffices for the relay to only know the priority of the cell. The problem now reduces to how the intermediate relay can reliably obtain this information. If it relies on the predecessor relay, a selfish relay could always claim its own traffic as high priority and enjoy the benefit.

The proposed solution for this problem is to give "gold star" status to relays that provide good service to others. A gold star relay's traffic is given high priority by other relays, i.e.,

they always get relayed ahead of other traffic. Furthermore, when a gold star relay receives a high priority connection from another gold star relay, it passes on the gold star status so the connection remains high priority on the next hop. All other traffic gets low priority. If a low priority node relays data through a gold star relay, the traffic is relayed but at low priority. Traffic priority is circuit-based. Once a circuit is created, its priority remains the same during its entire lifetime.

Due to variations of the network conditions and the multi-hop nature of Tor, it may take multiple measurements to get accurate results. Therefore, a "k out of n" approach is used, where a relay has to have satisfactory performance for k times out of the last n measurements to be eligible for gold star status. At this point, it becomes a policy issue of who gets a gold star. For example, one could assign a gold star to the fastest $7/8$ of the nodes, following the current Tor design in which the slowest one-eighth of Tor relays are not used to relay traffic at all. The directory authorities can then distribute the gold star status labels with the relay information they presently distribute.

In this way, no only do users with slower relays receive worse performance, they also have less anonymity, since there are only a smaller fraction of them sending traffic without the gold star.

Design properties

As measurements are performed centrally, peers need not have any trust in one another. Likewise, the system will respond quickly when the central authority publishes a finding. Best of all, none of the published information would compromise the anonymity of other

Tor traffic. The only information ever measured or published is whether a given node passed an audit for properly relaying its traffic.

The effectiveness of this approach depends on the accuracy of the measurements, which in turn depends on the measurement frequency. Frequent measurements increase confidence, but they also place an increasing burden on the overlay network and limit the scalability of the measuring nodes.

6.4 Experiments

This section shows simulation results of Tor networks under different scenarios. The goal is to evaluate the effectiveness of the two measurement schemes against a variety of different scenarios, including varying amounts of load on the Tor network, and varying strategies taken by simulated nodes (e.g., selfish vs. cooperative).

6.4.1 Experimental apparatus

We built a packet-level discrete event simulator that models a Tor overlay network. The simulator, written in Java, was executed on 64-bit AMD Opteron 252 dual-core servers with 4GB of RAM and running RedHat Enterprise Linux (kernel version 2.6.9) and Sun's JVM, version 1.5.0.

The simulator simulates every cell at every hop. Each node, particularly simulated BitTorrent clients, can easily have hundreds of outstanding cells in the network at any particular time. Unsurprisingly, the simulations are slow and memory-intensive. In fact,

in some larger scale simulations, the simulated time is slower than the wall clock time. Likewise, memory usage is remarkable. Simulating 20 BitTorrent clients and 2000 web clients consumes most of the available memory. To keep the client-to-relay ratio realistic, we limit the simulations to Tor networks with around 150 relays.

For simplicity, we assume that the upstream and downstream bandwidth for all relays is symmetric, since the forwarding rate of any relay with asymmetric bandwidth will be limited by its lower upstream throughput. We also assume relays take no processing time. The cooperative relays (which reflect the altruists in the current Tor network) have a bandwidth of 500KB/s. We assume the latency between any two nodes in the network is fixed at 100 ms.

The simulations use different numbers of simplified web and BitTorrent clients to generate background traffic. The web traffic is based on Hernández-Campos et al. [HCJS03]'s "Data Set 4," collected in April 2003 [The]. The simplified BitTorrent clients always maintain four connections and will upload and download data at the maximum speed Tor allows. They also periodically replace their slowest connection with a new one, much like the real BitTorrent seeks to maximize the download rate from its available connections. We assume that the external web or BitTorrent servers have unlimited bandwidth. The different relay traffic types are:

Cooperative. These nodes will use their entire 500KB/s bandwidth to satisfy the needs of their peers, and will give priority to "gold star" traffic when present. (If sufficient gold star traffic is available to fill the entire pipe, regular traffic will be completely

starved for service.)

Selfish. These nodes *never* relay traffic for others. They are freeloaders on the Tor system with 500KB/s of bandwidth.

Cooperative slow. These nodes follow the same policy as cooperative nodes, but with only 50KB/s of bandwidth.

Cooperative reserve. These nodes have 500KB/s bandwidth, just like cooperative nodes, but cap their relaying at 50KB/s, unless they are currently using a connection for their own traffic, in which case they do not cap that connection.

Adaptive. These nodes will behave just like cooperative nodes until they get a gold star. After this, they will change to the selfish policy until they lose the gold star.

All of the simulations use ten directory authorities. Every minute each directory authority will randomly build a circuit with three Tor relays and measure its bandwidth by downloading a small 40KB file from an external server. The bandwidth measurement is recorded and attributed to only the middle relay in the circuit. To obtain a gold star, we required Tor relays to successfully relay traffic at least two times out of the last five measurements (i.e., $k = 2$ and $n = 5$ in Section 6.3.2).

When reporting the simulation results, the observed network performance will be described in terms of "download time" and "ping time." The former describes the necessary time for each node to download a 100KB file from an external server. The latter describes the roundtrip latency for that same external server. (For the simulations, this external server

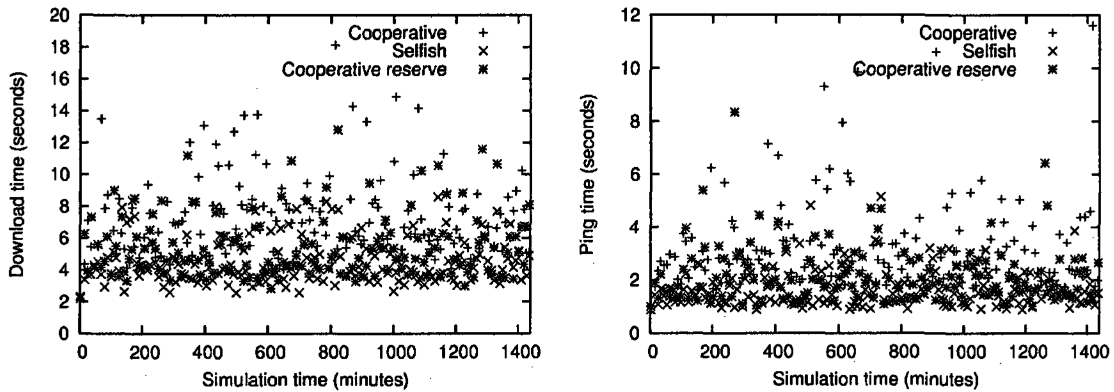


Figure 6.1: Average download and ping time over time when no incentive scheme is in place and heavy traffic (20 BitTorrent clients and 2000 web clients). Both download and ping time show significant variation, regardless of relay type.

is assumed to have infinite bandwidth and introduce zero latency of its own.) Both measures are important indicators of how a Tor user might perceive the quality of the experience when web surfing. For contrast, a Tor user running file-sharing software or downloading large files will be largely insensitive to latency.

6.4.2 Experiment 1: Unincentivized Tor

The first experiment is to understand how Tor networks behave when demand for the network's resources exceeds its supply. This experiment simulates 50 cooperative relays, 50 selfish relays, and 50 cooperative reserve relays, with heavy background traffic (20 BitTorrent clients and 2000 web clients).

Figure 6.1 plots the *average* download and ping time for each relay type. Even after averaging for 50 relays, the data points are still highly fluctuating, suggesting that the network performance is variable (and appears to be a long-tailed distribution). This is largely due to the BitTorrent traffic, as it sometimes dominates the available bandwidth, starving

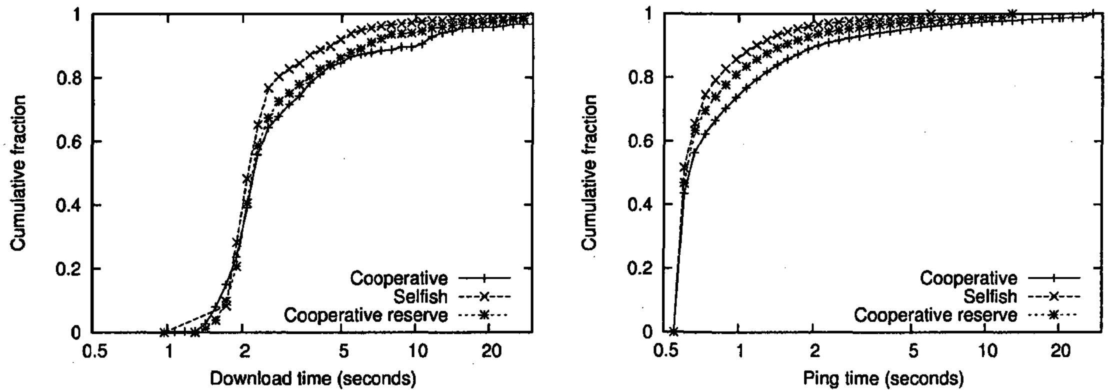


Figure 6.2: Cumulative download and ping time when no incentive scheme is in place and heavy traffic (20 BitTorrent clients and 2000 web clients). Performance for all relay types is similar, although selfish relays do somewhat better in the worst case.

other circuits sharing the same relays for bandwidth.

To get a better view of the distribution of download times and ping times, cumulative distribution functions (CDFs) are used. Figure 6.2 represents the same data as Figure 6.1, albeit without any of the averaging. The x-axis represents download time or ping time and the y-axis represents the percentage of nodes who experienced that particular download or ping time or less.

While the ideal download time for all relay types in this experiment is 0.8 second (six network roundtrip hops plus bandwidth time), all relay types rarely achieve anywhere close to this number. Figure 6.2 clearly shows that roughly 80% of the attempted downloads take more than two seconds, regardless of a node's policy. Cooperative relays have approximately 10% of attempted downloads taking longer than ten seconds. Less than 5% of the selfish nodes see such poor performance. Selfish nodes, in general, do better in the worst case than cooperative nodes, but observe similar common-case performance.

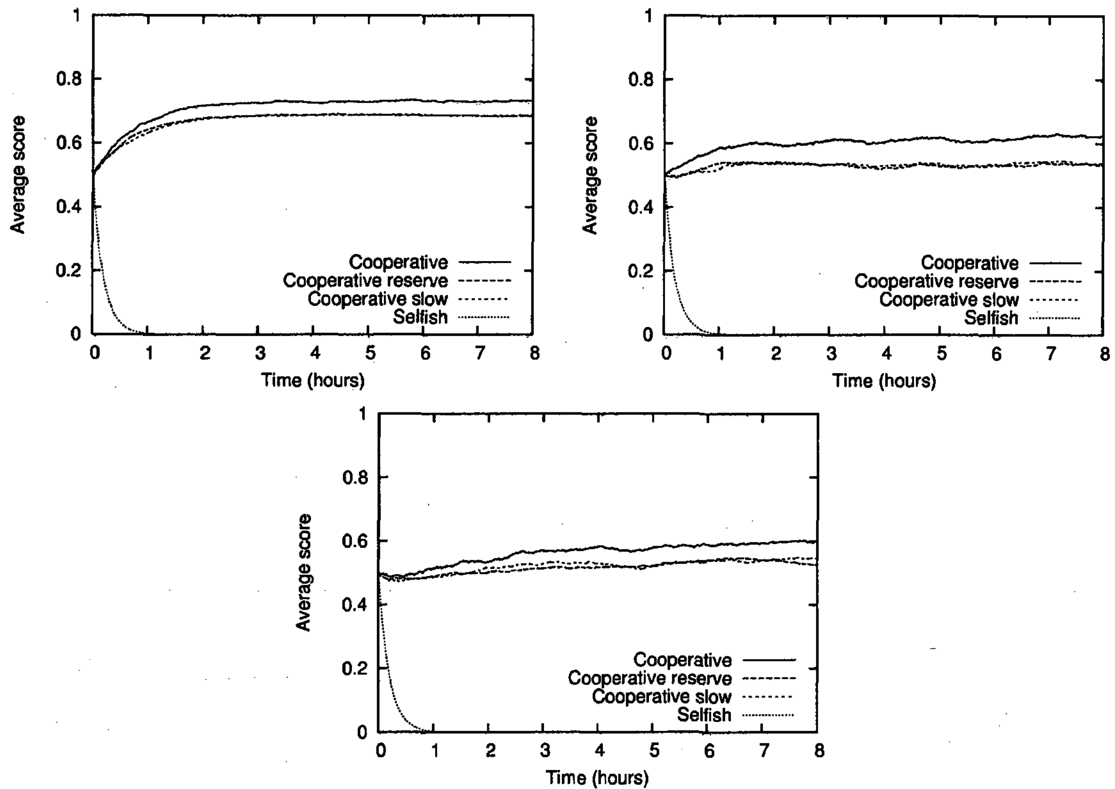


Figure 6.3: The average scores measured for different relays, with no background traffic (upper left), light background traffic (upper right), and heavy background traffic (bottom).

6.4.3 Experiment 2: Score measurements

This set of experiments verifies the feasibility to measure the performance accurately in a probational Tor network for promoting cooperative relays to the premium network, as described in Section 6.3.2.

The first experiment shows 40 of each of four types of relays (cooperative, cooperative reserve, cooperative slow, and selfish) under no background traffic, light background traffic (10 BitTorrent clients and 1000 web clients), and heavy background traffic (20 BitTorrent clients and 2000 web clients). Each directory server periodically measures the download time and gives a score to the middle relay relative to an "ideal" time. Figure 6.3 shows the

results over time.

First, observe that in all cases, the average score for selfish relays drops to zero very quickly, because they never relay traffic. On the opposite, cooperative relays attains and remains at a relatively high score, although less smoothly with increased background traffic. The scores for cooperative reserve and cooperative slow relays are very similar. This shows that a simple measurement can provide a coarse partition of cooperative relays from selfish ones, and this is sufficient for promoting more or less cooperative relays to a premium network.

Alternating relays

The second experiment investigates on how reactive the measurements are to relays' changing behavior. This experiment has 50 of each of three types of relays (cooperative, alternating, and selfish). Alternating relays change their behavior every two hours, between fully cooperative and fully selfish. Figure 6.4 shows the results over time.

From the figures, the score patterns for cooperative and selfish relays are similar to before. For alternating relays, their scores quickly approached to that of the cooperative or selfish relays, depending on which mode they are in. This shows that the measurement scheme is not only accurate, but it is also dynamic enough to find out the recent performance of the relays. Therefore, it is a very effective way to track relay performance for assigning them to a proper Tor network.

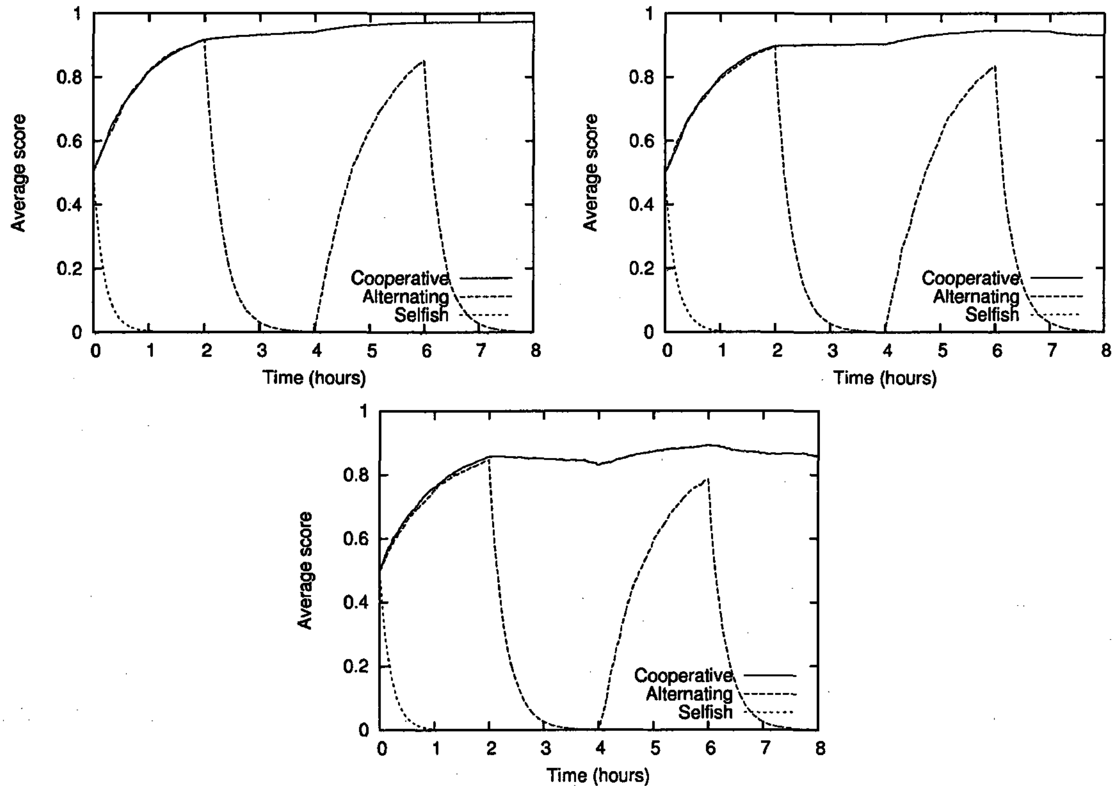


Figure 6.4: The average scores measured when there are alternating relays, with no background traffic (upper left), light background traffic (upper right), and heavy background traffic (bottom).

6.4.4 Experiment 3: Gold stars

This experiment set measures the effectiveness of the gold star mechanism described in Section 6.3.2. The simulation consists of 40 cooperative relays, 40 selfish relays, 40 cooperative slow relays, and 40 adaptive relays. These variations show whether slower cooperative nodes still get the benefits of a gold star, and whether adaptive nodes can be more effective than purely selfish nodes. Figures 6.5, 6.6, and 6.7 show the cumulative download and ping time with no background traffic, light background traffic, and heavy background traffic, respectively.

The results are striking. Cooperative nodes maintain their performance, regardless of

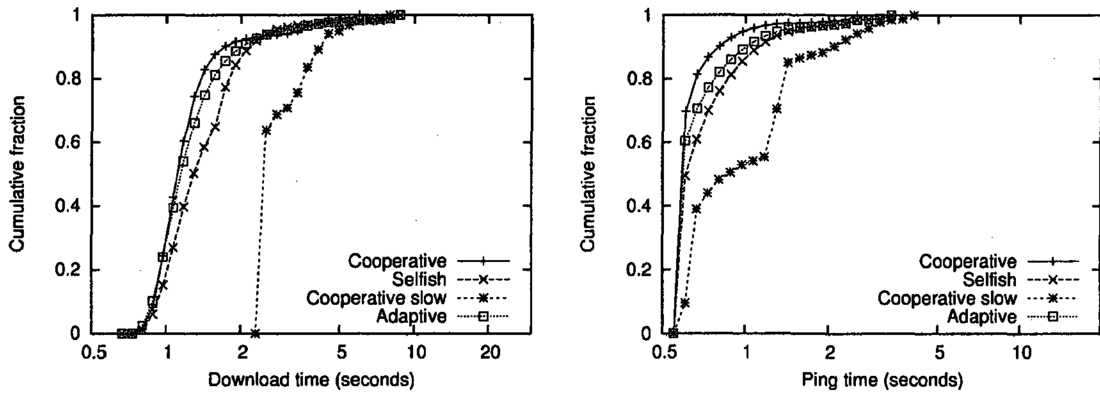


Figure 6.5: Cumulative download and ping time with the gold star scheme and no background traffic.

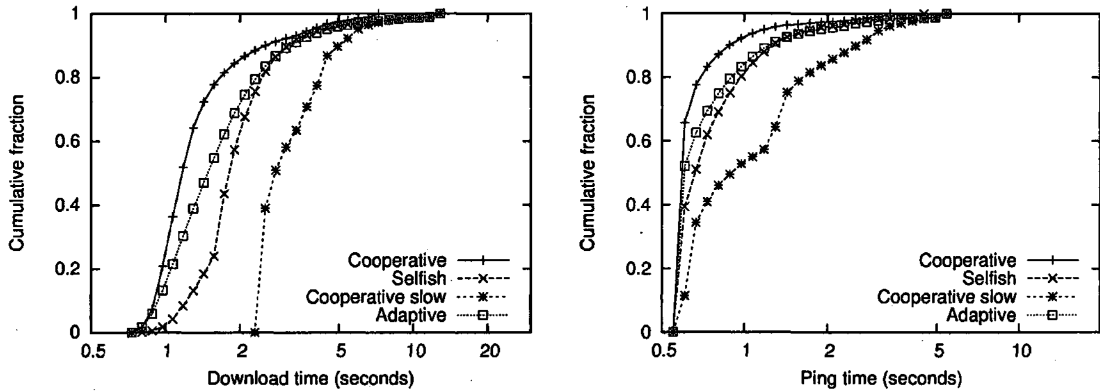


Figure 6.6: Cumulative download and ping time with the gold star scheme and light background traffic (10 BitTorrent clients and 1000 web clients). Selfish and adaptive relays now begin to suffer while cooperative relays maintain their performance.

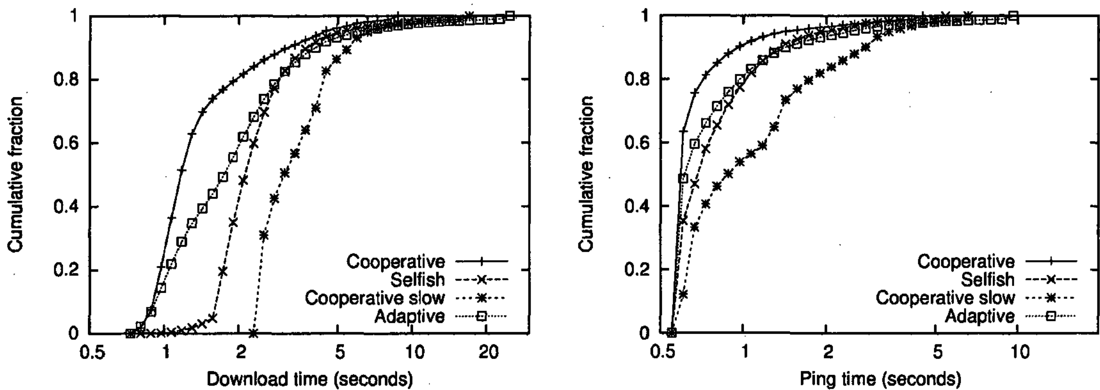


Figure 6.7: Cumulative download and ping time with the gold star scheme and heavy background traffic (20 BitTorrent clients and 2000 web clients). Cooperative nodes maintain their performance, while the penalty for selfish and adaptive nodes is more pronounced.

the level of background traffic in the overlay. When there is no background traffic, they slightly outperform the selfish and adaptive nodes, but once the traffic grows, the cooperative nodes see clear improvements in download time and in latency. For example, under heavy background traffic, 80% of the cooperative nodes see download times under two seconds, versus roughly 2.5 seconds for the selfish and adaptive nodes.

This experiment shows that the adaptive scheme is ineffective at defeating the gold star mechanism. Adaptive nodes will experience better performance while they have a gold star, but their benefit only splits the difference between the cooperative and selfish policies, roughly in proportion to the additional effort they are spending to maintain their gold star.

Cooperative slow nodes, like their fast counterparts, experience stable performance as the background load on the Tor network increases. This demonstrates that the gold star policy can effectively reward good behavior, regardless of a node's available bandwidth.

A further experiment replaces the cooperative slow nodes with cooperative reserve nodes, representing a possibly rational response to the gold star mechanism. As a node only needs to prove that it is relaying data in order to get the gold star, it might benefit by reserving most of its bandwidth for its own needs, here using only 10% of its bandwidth for its contributions to the good of other nodes. Figures 6.8–6.10 show the results of this experiment.

In each condition, both kinds of cooperative nodes observe identical distributions of bandwidth and latency. Again, selfish and adaptive nodes suffer as the background traffic increases. This experiment shows, unsurprisingly, that nodes need not be "fully" cooper-

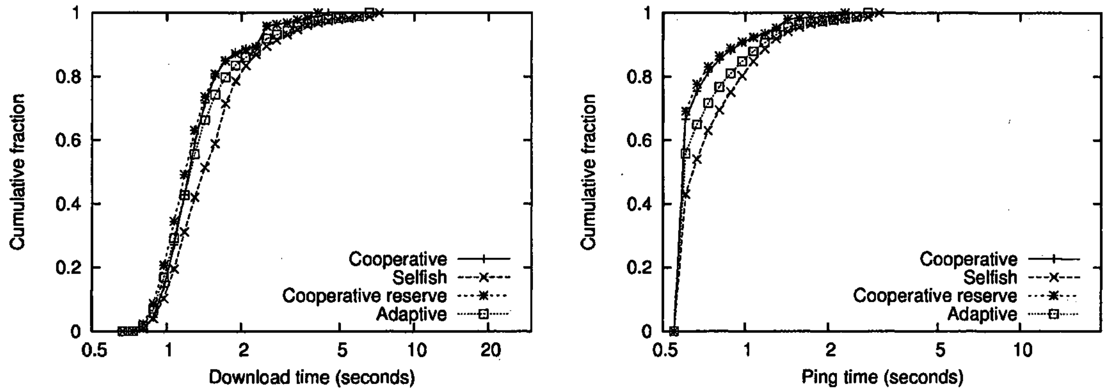


Figure 6.8: Cumulative download and ping time with the gold star scheme and no background traffic. Cooperative reserve relays, which replaced cooperative slow relays, have similar performance with cooperative relays.

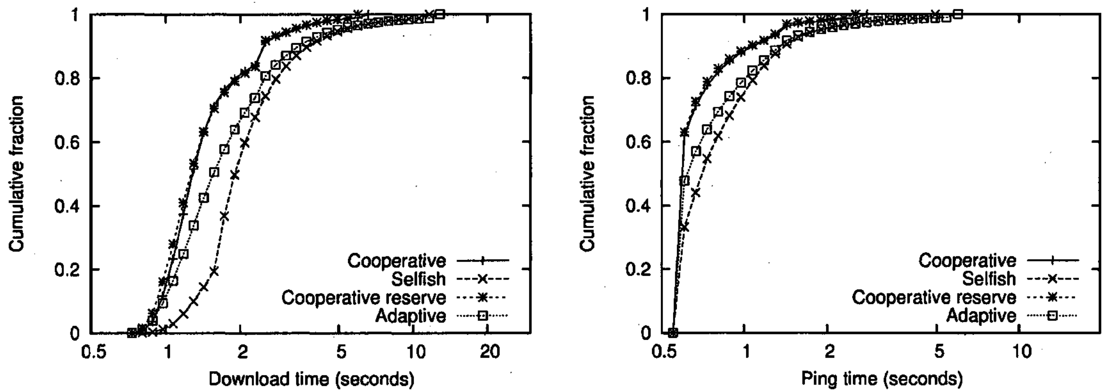


Figure 6.9: Cumulative download and ping time with the gold star scheme and light background traffic (10 BitTorrent clients and 1000 web clients). Only selfish and adaptive relays are affected with the increased traffic.

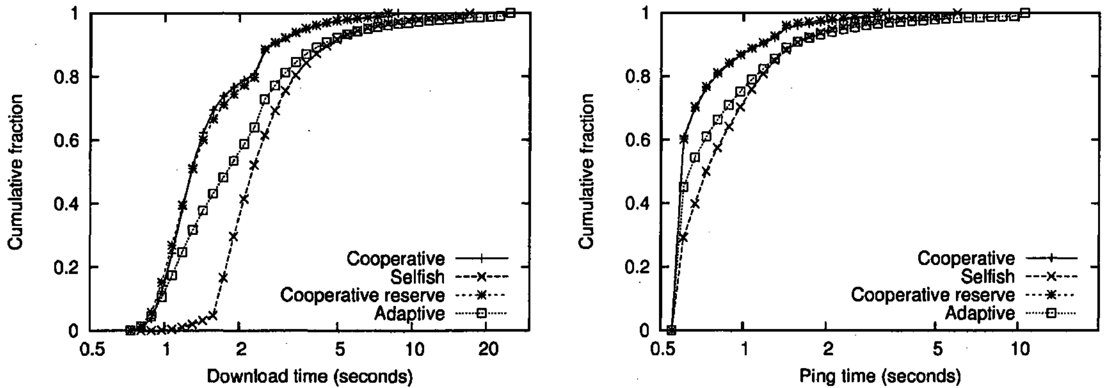


Figure 6.10: Cumulative download and ping time with the gold star scheme and heavy background traffic (20 BitTorrent clients and 2000 web clients). Again, cooperative and cooperative reserve relays are not affected, while the performance of selfish and adaptive relays become much worse.

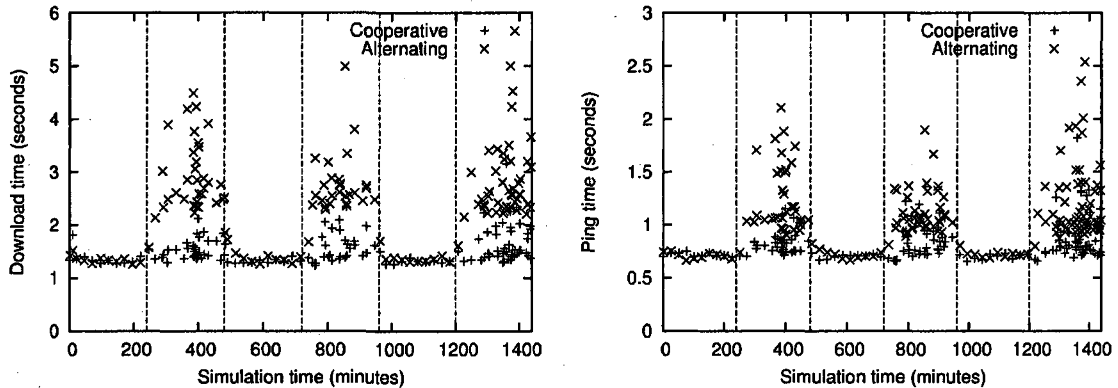


Figure 6.11: Average download and ping time with relays that alternate between being cooperative and selfish. This experiment is with gold star scheme in place and heavy background traffic (20 BitTorrent clients and 2000 web clients). Dotted lines show the times at which the alternating relays switch. The performance of alternating relays gets worse whenever they switched to being selfish, while that for cooperative relays only suffers a little.

ative to gain a gold star. In an actual Tor deployment, it would become a policy matter, perhaps an adaptive process based on measuring the Tor network, to determine a suitable cutoff for granting gold stars (see Section 6.5.1 for more discussion on handling strategic behaviors in Tor).

Alternating relays

This experiment considers a variation on the adaptive strategy, used previously. Alternating nodes will toggle between the cooperative and the selfish strategies on a longer timescale — four hours per switch. This experiment uses 50 such alternating relays with 50 cooperative relays and with heavy background traffic (20 BitTorrent clients and 2000 web clients).

Figure 6.11 shows the average download and ping time for both relay types over time. During the periods where the alternating relays are cooperative, they receive service of a

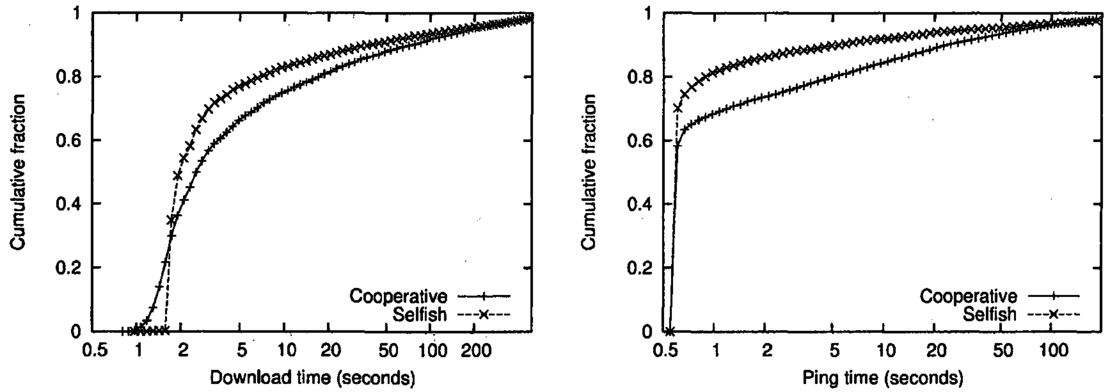


Figure 6.12: Cumulative download and ping time with the pair-wise reputation design and heavy traffic (20 BitTorrent clients and 2000 web clients). Four relay types (cooperative, selfish, cooperative reserve, and adaptive) are simulated, although only the performance of the former two are shown, as the latter two behave similarly to cooperative relays.

similar quality as the full-time cooperative nodes. However, once the alternating relays switch to become selfish, their download times quickly increase, representing the same quality of service that would be observed by a selfish node. Of interest, while the cooperative nodes do observe lower quality of service (after all, fully half of the Tor nodes stopped relaying any data), they still do much better than their selfish peers.

This experiment further demonstrates the system robustly responding to changes in node behavior.

6.4.5 Experiment 4: Pair-wise reputation

This final experiment investigates a variation on the gold star design, where individual circuits are not labeled as being low or high priority. In this variation, a low-priority node routing traffic through a gold-star node will experience priority delays getting the gold star node to accept the traffic, but the traffic will have the gold star priority in its subsequent hops. This alternative design has significant improvements from an anonymity perspective,

because traffic at a given hop does not give any hint about whether it originated from a low-priority or high-priority node. However, this design might fail from an incentives perspective, since there is less incentive for a node to earn its own gold star.

This experiment again simulates a network with 40 relays for each relay type: cooperative, selfish, cooperative reserve, and adaptive. For clarity, Figure 6.12 only shows the download and ping time for cooperative and selfish relays, as the performances for cooperative reserve and adaptive relays are very close to those for cooperative relays.

This experiment shows selfish nodes clearly outperforming their cooperative peers. This indicates that the gold star strategy requires a transitive property, i.e., each hop of a circuit must inherit the gold star status of the previous hop. Otherwise, selfish nodes will outperform their cooperative peers and there will be no incentive for cooperation.

6.5 Discussion

6.5.1 Strategic users

The proposed incentive scheme is not perfectly strategy-proof, in the sense that users can earn a gold star without providing *all* of their network capacity for the use of the Tor network. This creates a variety of possible strategic behavior.

Provide borderline or spotty service. A relay needs to provide only the minimal amount of bandwidth necessary to gain the gold star. Of course, if every user provided this amount, Tor would still have vastly greater resources than it does today. Next, because the band-

width policies are determined centrally, the minimum bandwidth necessary to obtain a gold star could be moved up or down manually. Strategic nodes will then adjust the capacity they give to the Tor network, making more bandwidth available whenever they are needed.

Only relay at strategic times. Such users might provide relay services only when the "local" user is away, and thus not making demands on the Tor network. Such behavior is not disincentivized by this research, as it still provides scalable resources to the Tor network. However, any users following such behavior may be partially compromising their anonymity, as their presence or absence will be externally observable.

Forward high-priority traffic as low-priority. A relay who correctly forwards traffic can still cheat by changing the priority on incoming traffic. The measuring authorities should build high priority test circuits back to a trusted relay, to see if the circuit arrives with the expected high priority status.

6.5.2 The audit arms race

Some attacks outlined above involve relays that provide some level of service but not quite as much as expected. The response in each case is a smarter or more intensive measurement algorithm so the directory authorities can more precisely distinguish uncooperative behavior.

To see why this would not be an arms race between increasingly subtle cheating and increasingly sophisticated audits, consider the incentives for ordinary users. The most chal-

lenging part of setting up a Tor relay is configuring the software, enabling port forwarding in the firewall, etc. Compared to this initial barrier, the incremental cost of providing a bit more bandwidth is low for most users. As long as the audit mechanism correctly judges whether the user relays any traffic at all, it is verifying that the user has performed the most costly step in setting up relaying. The diminishing returns a strategic relay gets in saving bandwidth as the arms race progresses will limit the complexity required for the auditing mechanism.

6.6 Summary

This chapter proposes an incentive scheme to reward Tor users who relay traffic. Simulations show that users who cooperate with the desired policies can be identified, and they can achieve sizable performance improvements, while at the same time also enjoy a better k-anonymity. This creates significant incentives for many users to join the Tor network as relays, further improving the system in both aspects.

There are some areas for further research, such as how to reward relays without separating anonymity sets, how to scale up the audits to work on a larger Tor network, what thresholds should merit a gold star, and whether simulations are needed to reflect a more realistic mix of users (e.g., more slow relays and/or relays with asymmetric bandwidth). Once these issues have been investigated, they should be integrated to the design for an upcoming Tor release and test how well it works in real network conditions.

6.7 Related Work

Real-world anonymizing networks have operated on three incentive approaches: *Community support*, *payment for service*, and *government support*. (Discussion of the funding approaches for research and development of anonymity designs, while related, is outside the scope of this thesis.) The Tor network right now is built on community support: A group of volunteers from around the Internet donate their resources because they want the network to exist.

Zero-Knowledge Systems' Freedom network [BSG00] on the other hand was a commercial anonymity service. They collected money from their users, and paid commercial ISPs to relay traffic. While that particular company failed to make its business model work, the more modest Anonymizer [Ano] successfully operates a commercial one-hop proxy based on a similar approach.

Lastly, the AN.ON project's cascade-based network is directly funded by the German government as part of a research project. Unfortunately, the funding ended in 2007, so they are exploring the community support approach (several of their nodes are now operated by other universities) and the pay-for-play approach (setting up commercial cascades that provide more reliable service).

Other incentive approaches have been discussed as well. Acquisti et al. [ADS03] argued that high-needs users (people who place a high value on their anonymity) will opt to relay traffic in order to attract low-needs users — and that some level of free riding is actually beneficial because it provides cover traffic to blend with. This is unclear how well that

argument transitions from high-latency systems analyzed to low-latency ones, especially since the different threat models change the incentive structure.

Chapter 7

Design Discussions

This thesis addresses the incentive problem in peer-to-peer systems with designs that vary based on application type. This is necessary because users' incentive is inherently different under different applications, and the best solution depends on the specific properties of the particular application type. To better understand the constraints and characteristics of different mechanisms, this chapter provides a summary of the mechanism types and discusses when and how they are useful.

7.1 Pairwise Exchanges

Tit-for-tat. Despite being the simplest and most straightforward mechanism, tit-for-tat rarely works except for a limited set of applications. The underlying reason is that tit-for-tat requires two parties to have available resources and be interested in each other's resources at the same time. Although it is typical for BitTorrent nodes to exchange content with each other, as well as nodes in backup applications to exchange storage space with each other, this prerequisite is generally hard to realize when users have diverse interest, say for general content distribution systems. It also becomes increasingly harder to find

exchange partners as the network size and resource diversification increase. Moreover, users cease to have incentives to provide resources when they do not have any immediate demand, and this limits the networks from achieving their full potential.

Electronic currencies. The use of electronic currencies is a natural extension over tit-for-tat, as it enables the division of "buying" and "selling" of service into separate transactions and allows peers to transact with different partners. Additionally, it encourages users to contribute resources even at times when they do not need anything in return, as currencies can be saved for later use.

However, without a working reputation system already in place, currencies issued by individuals do not have much perceived value. Otherwise, currencies need to be issued and controlled by some trusted authority, but such an authority may not be available or feasible in peer-to-peer systems. It is also relatively expensive to implement, since any transaction would either require cryptographic operations or has to be overseen by a trusted accounting party. Whenever there is a dispute over a transaction, it is usually difficult for the trusted party to decide which side is to blame. As a result, electronic currency is hard to define properly and difficult to deploy.

7.2 Reputation

Instead of relying individual nodes themselves to track the pairwise history with all the peers they have interacted, it is more efficient and effective to have a global notion of reputation of each node based on how it has interacted with all peers in the past. If

reputation is available, nodes can always make an informed decision on which of its peers to provide service to, even if they have not received service from those peers before. Nodes with poor reputation would find it difficult to receive any service from the system.

There are problems associated with reputation systems. For centralized reputation systems, there needs to be a trusted central authority to compute reputation, creating a problem similar to that with electronic currencies. If it is distributed, it is difficult to ensure its correctness and accuracy, particularly in the presence of collusion. Regardless, if reputation is computed based on peers' reported opinion, it would create an incentive for Sybil attacks.

The design for anonymous communication networks in Chapter 6 is somewhat similar to a reputation system. It is feasible only because it leverages the existing trusted authority in directory servers, and reputation is computed based on anonymous measurement performed by those trusted directory servers.

7.3 General Primitives

Ideally, if there were a general primitive that could deter freeloading, it would then be possible to apply across different applications and solve all incentive problems. An attempt towards this goal is the BAR model [AAC+05], which ensures the replicated state machine protocols it uses are incentive-compatible. However, this approach is quite expensive, because it relies on a multi-party agreement protocol for each decision. In addition, collusion posts a real challenge, as colluding peers could help each other to cover their misbehavior.

PeerReview [HKD07] employs a more practical approach. It ensures that Byzantine

faults observed by correct nodes are eventually detected and irrefutably linked to the faulty node. Since it is based on leaving evidence, there could be a time lag before misbehavior is discovered. If quick detection of misbehavior is important, PeerReview would require frequent audits, which would also become expensive. In these scenarios, a custom solution can usually be faster and more efficient.

Furthermore, some applications may have certain properties that cannot coexist with these primitives. For example, neither primitive is particularly suitable for anonymous communication networks discussed in Chapter 6 as both would generate traces to assist an adversary to deanonymize traffic.

It is also interesting to note that both schemes choose a punishment approach, where unknown peers are assumed cooperative and are punished only after they have deviated from expected behavior. As to be discussed in next section, this approach may not be very effective for untrusted peer-to-peer systems.

7.4 Other Design Choices

Punishment vs. rewarding. Traditional system prefers a "punishment" approach, where nodes are initially assumed to be cooperative, but misbehavior would be detected and misbehaving nodes punished. This design mindset is in general not suitable for peer-to-peer systems, since the majority of nodes may not be altruistically cooperative. Also, unless a perfect solution against Sybil attack exists, if the cost for a node to leave and rejoin the system is lower than that of the punishment, nodes will simply decide to leave, rendering

the threat of punishment toothless.

Thus, for the punishment approach to work, the cost of rejoining has to be so high that nodes would prefer not to jeopardize their status. Out of all applications considered, the only suitable case is archival storage systems, because when a node is expelled from the network, all the content it has stored on the network would be lost, and rejoining would mean re-uploading all their data.

Direct vs. indirect experience. Another design choice is whether nodes rely solely on its direct experience to evaluate its peers, or also rely on its peers' experience. The former case limits what and how fast a node can learn about its peers, and every node has to individually learn the same facts. However, first-hand observation is also more reliable. The design for streaming applications in Chapter 4 can rely on only direct experience since there is a large number of interactions between nodes, and the number is further increased through tree reconstructions. On the other hand, it does not work for anonymous communication networks due to the limited number of interactions with different nodes. In fact, to preserve anonymity, Tor explicitly discourages nodes from interacting with too many peers. This makes direct experience extremely limited.

7.5 Common Design Principles

After studying a number of different applications in this dissertation, this section summarizes several design principles that can be applied to a variety of peer-to-peer systems. These principles are aimed for very large scale systems in an untrusted and ungoverned

environment, for example an open peer-to-peer network on the Internet.

Principle 1: Understand users' interests. First and foremost, system designers must have a throughout understanding of users' incentives. In Economics terms, they must be able to accurately model the users' utility functions. This is necessary because these systems must be able to reward desirable behavior and punish undesirable ones, thus creating the foundation for cooperation.

Principle 2: Decentralized design. Peer-to-peer systems are meant to scale. Any centralized component can easily become a bottleneck or a single point of failure and disrupt the service. Thus, to maintain high availability, these systems must be designed in a mostly distributed fashion. In particular, peers should be able to make all decisions locally, without constantly consulting centralized parties like a bank or a reputation server.

Principle 3: Limited credits to strangers. In an open peer-to-peer system, the cost for one to leave and rejoin is typically very low. Yet users usually interact with a large number of peers, many of whom they have no prior knowledge with. On one hand, if all users refuse to service strangers at all, the whole system cannot bootstrap; on the other hand, if they blindly provide good services to any stranger, there will be little incentive for selfish users to contribute. As a result, the design philosophy should be reward-based instead of punishment-based, and users should give some limited credits, but not too much lenience, to strangers.

Principle 4: Use only trustworthy information. Users can obtain information about their peers through either their personal experience or third-parties. In a world where services are valuable, it is conceivable that users may lie or even collude if that could improve their services. This means most third-party information is unreliable. Thus, users should avoid using third-party information, unless it is from a known, reliable source, for example a trusted authority or friends.

Principle 5: Simple and intuitive policies. One of the aims for designing fair policies is so that self-interested users would follow them to maximize their own benefits. If users cannot understand what they need to do to receive better service, or if they do not agree with the policies, they may not behave in accordance with what is intended by the system designers, or may even leave the network. Thus, it is equally important to have the policies clearly communicated to users and for the policies to have a perceived fairness.

Chapter 8

Future Work

This thesis covers only four common peer-to-peer applications. An obvious future work is to apply the same design principles to other applications, for example Voice over IP (VoIP), non-streaming publish/subscribe networks, gaming framework, etc.

Under the proposed designs, simulation results show that freeloaders can only receive partial service, if any, and the extent depends on how much resources they are contributing to the system. However, it is not entirely clear in practice how selfish agents would react to these systems. Would the reduced service still be attractive to them? Or would they choose to leave the system for good instead? Questions like these can be answered by modeling the utility function of selfish peers with their level of resource contribution. As a simplified example, let $c(r)$ and $b(r)$ be the cost and benefit of a peer when it chooses to contribute resource r. Then the utility function $u(r)$ would be $b(r) - c(r)$. A rational agent would adjust r to maximize $u(r)$. Ideally, $u(r)$ should monotonically increase with r, so that agents would be inclined to provide as much resources as they could.

The designs in this thesis assume that users are unrelated and do not initially trust each other. As a result, the system as a whole pays the cost for suspecting strangers [DFM01,

FR01]. A recent research direction is to incorporate social networks into peer-to-peer systems [MGGM04, PGW+06]. For instance, in general content distribution networks discussed in Chapter 5, friends could give each other high initial confidence or even unbounded debt threshold. An interesting research question is how social networks can be integrated into incentive mechanisms, and how much additional improvement can be achieved.

Mobile ad hoc networks face a similar incentive problem, since nodes in those networks rely on each other to forward traffic. Numerous schemes have been proposed to provide incentive in routing [BL02, MGLB00, MRWZ05, SBHJ06]. In general, incentivizing mobile ad hoc networks may be more difficult than peer-to-peer networks due to the limited computational resources and peer connectivity in mobile nodes. It would be interesting to see how much of the incentive schemes for peer-to-peer networks can be applied to mobile ad hoc networks.

8.1 Conclusions

My research makes the following contributions.

- Identification of selfishness in peer-to-peer systems as a fundamentally separate and independent problem to the conventional adversarial attack model.
- A proposal of a general approach to designing incentive into peer-to-peer systems.
- Application of the proposed approach to four different peer-to-peer applications:

  - An auditing scheme for peer-to-peer archival storage systems.
  - A local observation and selective servicing scheme for peer-to-peer streaming systems.
  - A pairwise debt and transitive trading scheme for peer-to-peer content distribution systems.
  - A centrally measured and assigned but distributedly enforced priority scheme for anonymous communication networks.
- A summary of different mechanisms and a discussion of their suitability for different application types.

Incentive for resource contribution is a problem faced by all open, cooperative peer-to-peer applications. This problem is becoming more and more severe with the increasing number of freeloaders observed in these networks. The model and designs in this thesis provide peer-to-peer system designers, developers, and administrators suggestions and directions to handle freeloaders in their networks, as well as an arsenal of mechanisms for them to implement. With freeloaders taken care of, this removes the largest stumbling block for peer-to-peer applications to live up to their full potential.

Bibliography

[AAC+05] Amitanand S. Aiyer, Lorenzo Alvisi, Allen Clement, Mike Dahlin, Jean-Philippe Martin, and Carl Porth. BAR fault tolerance for cooperative services. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles (SOSP)*, Brighton, UK, October 2005.

[ABMW05] Martín Abadi, Mike Burrows, Mark Manasse, and Ted Wobber. Moderately hard, memory-bound functions. *ACM Transactions on Internet Technology (TOIT)*, 5(2):299–327, May 2005.

[AD01] Karl Aberer and Zoran Despotovic. Managing trust in a peer-2-peer information system. In *Proceedings of the Tenth International Conference of Information and Knowledge Management (CIKM)*, Atlanta, GA, November 2001.

[ADS03] Alessandro Acquisti, Roger Dingledine, and Paul Syverson. On the economics of anonymity. In *Proceedings of the Seventh Annual Conference on Financial Cryptography (FC)*, Gosier, Guadeloupe, January 2003.

[AFK+04] Aaron Archer, Joan Feigenbaum, Arvind Krishnamurthy, Rahul Sami, and Scott Shenker. Approximation and collusion in multicast cost sharing. *Games and Economic Behavior*, 47(1):36–71, April 2004.

[AG04] Kostas G. Anagnostakis and Michael B. Greenwald. Exchange-based incentive mechanisms for peer-to-peer file sharing. In *Proceedings of the Twenty-Fourth Interna-*

tional Conference on Distributed Computing Systems (ICDCS), Washington, DC, March 2004.

[AH00] Eytan Adar and Bernardo A. Huberman. Free riding on Gnutella. *First Monday*, 5(10), October 2000.

[Ano] The Anonymizer. `http://www.anonymizer.com/`.

[Apo01] John G. Apostolopoulos. Reliable video communication over lossy packet networks using multiple state encoding and path diversity. In *Proceedings of SPIE/Visual Communications and Image Processing (VCIP)*, San Jose, CA, January 2001.

[ARS+08] Elli Androulaki, Mariana Raykova, Shreyas Srivatsan, Angelos Stavrou, and Steven M. Bellovin. Par: Payment for anonymous routing. In *Processings of the Eighth International Symposium on Privacy Enhancing Technologies (PETS 2008)*, Leuven, Belgium, July 2008.

[AW01] John G. Apostolopoulos and Susie J. Wee. Unbalanced multiple description video communication using path decryption. In *Proceedings of the IEEE International Conference on Image Processing (ICIP)*, Thessaloniki, Greece, October 2001.

[Axe81] Robert Axelrod. The emergence of cooperation among egoists. *The American Political Science Review*, 75(2), June 1981.

[Bac02] Adam Back. Hashcash — a denial of service counter-measure. `http://www.hashcash.org/papers/hashcash.pdf`, August 2002.

[BBK02] Suman Banerjee, Bobby Bhattacharjee, and Christopher Kommareddy. Scalable application layer multicast. In *Proceedings of the ACM SIGCOMM*, Pittsburgh, PA, August 2002.

[BL02] Sonja Buchegger and Jean-Yves Le Boudec. Performance analysis of the CONFIDANT protocol. In *Proceedings of the Third ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc)*, Lausanne, Switzerland, June 2002.

[BLV05] Alberto Blanc, Yi-Kai Liu, and Amin Vahdat. Designing incentives for peer-to-peer routing. In *Proceedings of the Twenty-Fourth Annual IEEE Conference on Computer Communications (INFOCOM)*, Miami, FL, March 2005.

[BR03] Charles Blake and Rodrigo Rodrigues. High availability, scalable storage, dynamic peer networks: Pick two. In *Proceedings of the Ninth Workshop on Hot Topics in Operating Systems (HotOS IX)*, Kauaii, Hawaii, May 2003.

[BSG00] Philippe Boucher, Adam Shostack, and Ian Goldberg. Freedom systems 2.0 architecture. White paper, Zero Knowledge Systems, Inc., December 2000. `http://osiris.978.org/~brianr/crypto-research/anon/www.freedom.net/products/whitepapers/Freedom_System_2_Architecture.pdf`.

[CCZ04] Yang-hua Chu, John Chuang, and Hui Zhang. A case for taxation in peer-to-peer streaming broadcast. In *Proceedings of Workshop on Practice and Theory of Incentives and Game Theory in Networked Systems (PINS)*, Portland, OR, August 2004.

[CDG+02] Miguel Castro, Peter Druschel, Ayalvadi Ganesh, Antony Rowstron, and Dan S. Wal-

lach. Security for structured peer-to-peer overlay networks. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, December 2002.

[CDK+03] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. SplitStream: High-bandwidth multicast in cooperative environments. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, October 2003.

[CDKR02] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications (JSAC)*, 20(8):1489–1499, October 2002.

[CF05] Alice Cheng and Eric Friedman. Sybilproof reputation mechanisms. In *Proceedings of the Third Workshop on the Economics of Peer-to-Peer Systems*, Philadelphia, PA, August 2005.

[CGM02a] Brian F. Cooper and Hector Garcia-Molina. Bidding for storage space in a peer-to-peer data preservation system. In *Proceedings of the Twenty-Second International Conference on Distributed Computing Systems (ICDCS)*, Vienna, Austria, July 2002.

[CGM02b] Brian F. Cooper and Hector Garcia-Molina. Peer to peer data trading to preserve information. *ACM Transactions on Information Systems (TOIS)*, 20(2):133–170, April 2002.

[CGN+04] Yang-hua Chu, Aditya Ganjam, T. S. Eugene Ng, Sanjay G. Rao, Kunwadee Sri-

panidkulchai, Jibin Zhan, and Hui Zhang. Early experience with an Internet broadcast system based on overlay multicast. In *Proceedings of USENIX Annual Technical Conference*, Boston, MA, June 2004.

[CL99] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI)*, New Orleans, LA, February 1999.

[CMN02] Landon P. Cox, Christopher D. Murray, and Brian D. Noble. Pastiche: Making backup cheap and easy. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, December 2002.

[CN03] Landon P. Cox and Brian D. Noble. Samsara: Honor among thieves in peer-to-peer storage. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, October 2003.

[Coh03] Bram Cohen. Incentives build robustness in BitTorrent. In *Proceedings of Workshop on Economics of Peer-to-Peer Systems*, Berkeley, CA, June 2003.

[CRZ02] Yang-hua Chu, Sanjay G. Rao, and Hui Zhang. A case for end system multicast. *IEEE Journal on Selected Areas in Communications (JSAC)*, 20(8):1456–1471, October 2002.

[DDM03] George Danezis, Roger Dingledine, and Nick Mathewson. Mixminion: Design of a type III anonymous remailer protocol. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2003.

[DDPS03] Ernesto Damiani, Sabrina De Capitani di Vimercati, Stefano Paraboschi, and Pierangela Samarati. Managing and sharing servents' reputations in P2P systems. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 15(4):840–854, July 2003.

[DFM01] Roger Dingledine, Michael J. Freedman, and David Molnar. Accountability. In Andy Oram, editor, *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*, chapter 16. O'Reilly & Associates, first edition, February 2001.

[DGH+88] Alan J. Demers, Dan H. Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard E. Sturgis, Daniel C. Swinehart, and Douglas B. Terry. Epidemic algorithms for replicated database maintenance. *Operating Systems Review (OSR)*, 22(1):8–32, January 1988.

[DKK+01] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP)*, Chateau Lake Louise, Banff, Canada, October 2001.

[DLLKA05] George Danezis, Chris Lesniewski-Laas, M. Frans Kaashoek, and Ross Anderson. Sybil-resistant DHT routing. In *Proceedings of the Tenth European Symposium on Research in Computer Security*, Milan, Italy, September 2005.

[DM] Roger Dingledine and Nick Mathewson. Tor protocol specification. `https://www.torproject.org/svn/trunk/doc/spec/tor-spec.txt`.

[DMS04] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation

onion router. In *Proceedings of the Thirteenth USENIX Security Symposium*, San Diego, CA, August 2004. Project web site: `https://www.torproject.org/`.

[DN92] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In *Proceedings of the Twelfth Annual International Crpytology Conference on Advances in Cryptology (CRYPTO)*, Santa Barbara, CA, August 1992.

[Dou02] John R. Douceur. The Sybil attack. In *Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS)*, Cambridge, MA, March 2002.

[DR01] Peter Druschel and Antony Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems (HotOS VIII)*, Schoss Elmau, Germany, May 2001.

[DS02] Roger Dingledine and Paul Syverson. Reliable MIX cascade networks through reputation. In *Proceedings of the Sixth Annual Conference on Financial Cryptography (FC)*, Southampton, Bermuda, March 2002.

[DZD+03] Frank Dabek, Ben Zhao, Peter Druschel, John Kubiatowicz, and Ion Stoica. Towards a common API for structured peer-to-peer overlays. In *Proceedings of the Second International Workshop on Peer-to-Peer Systems (IPTPS)*, Berkeley, CA, February 2003.

[eba] eBay. `http://www.ebay.com/`.

[FCC+03] Yun Fu, Jeffrey S. Chase, Brent N. Chun, Stephen Schwab, and Amin Vahdat. SHARP: An architecture for secure resource peering. In *Proceedings of the Nine-*

teenth ACM Symposium on Operating Systems Principles (SOSP), Bolton Landing, NY, October 2003.

[FG02] Ernst Fehr and Simon Gächter. Altruistic punishment in humans. *Nature*, 415(6868):137–140, January 2002.

[FKSS03] Joan Feigenbaum, Arvind Krishnamurthy, Rahul Sami, and Scott Shenker. Hardness results for multicast cost sharing. *Theoretical Computer Science*, 304(1–3):215–236, July 2003.

[FM02] Michael J. Freedman and Robert Morris. Tarzan: A peer-to-peer anonymizing network layer. In *Proceedings of the Ninth ACM Conference on Computer and Communications Security (CCS)*, Washington, DC, November 2002.

[FNW03] Andrew C. Fuqua, Tsuen-Wan Johnny Ngan, and Dan S. Wallach. Economic behavior of peer-to-peer storage networks. In *Proceedings of Workshop on Economics of Peer-to-Peer Systems*, Berkeley, CA, June 2003.

[FPS01] Joan Feigenbaum, Christos Papadimitriou, and Scott Shenker. Sharing the cost of multicast transmissions. *Journal of Computer and System Sciences*, 63(1):21–41, August 2001.

[FR01] Eric Friedman and Paul Resnick. The social cost of cheap pseudonyms. *Journal of Economics and Management Strategy (JEMS)*, 10(2):173–199, June 2001.

[Fre] FreePastry. Open source implementation of Pastry. `http://freepastry.org/`.

[GDS+03] Krishna P. Gummadi, Richard J. Dunn, Stefan Saroiu, Steven D. Gribble, Henry M.

Levy, and John Zahorjan. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, October 2003.

[GJA03] Minaxi Gupta, Paul Judge, and Mostafa Ammar. A reputation system for peer-to-peer networks. In *Proceedings of the of the Thirdteenth International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, Monterey, CA, June 2003.

[Gnu] Gnutella. Gnutella.com. `http://www.gnutella.com/`.

[Gro03] Christian Grothoff. An excess-based economic model for resource allocation in peer-to-peer networks. *Wirtschaftsinformatik (Business Informatics)*, 3-2003, June 2003.

[Har68] Garrett Hardin. The tragedy of the commons. *Science*, 162(3859):1243–1248, 1968. Alternate location: `http://dieoff.com/page95.htm`.

[HC04] Ahsan Habib and John Chuang. Incentive mechanism for peer-to-peer media streaming. In *Proceedings of the Twelfth IEEE International Workshop on Quality of Service (IWQoS)*, Montreal, Canada, June 2004.

[HCJS03] Felíx Hernández-Campos, Kevin Jeffay, and F. Donelson Smith. Tracking the evolution of web traffic: 1995–2003. In *Proceedings of the Eleventh IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Orlando, FL, October 2003.

[HCW05] Daniel Hughes, Geoff Coulson, and James Walkerdine. Free riding on Gnutella revisited: The bell tolls? *IEEE Distributed Systems Online*, 6(6), June 2005.

[HKD07] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. PeerReview: Practical accountability for distributed systems. In *Proceedings of the Twenty-First ACM Symposium on Operating Systems Principles (SOSP)*, Stevenson, WA, October 2007.

[HPJ02] Yih-Chun Hu, Adrian Perrig, and David B. Johnson. Ariadne: A secure on-demand routing protocol for ad hoc networks. In *Proceedings of the Eighth Annual International Conference on Mobile Computing and Networking (MobiCom)*, Atlanta, GA, September 2002.

[JGJ+00] John Jannotti, David K. Gifford, Kirk L. Johnson, M. Frans Kaashoek, and James W. O'Toole. Overcast: Reliable multicasting with an overlay network. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, October 2000.

[Kaz] Kazaa. `http://www.kazaa.com/`.

[KRAV03] Dejan Kostić, Adolfo Rodriguez, Jeannie Albrecht, and Amin Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, October 2003.

[KSGM03] Sepandar D. Kamvar, Mario T. Schlosser, and Hector Garcia-Molina. The EigenTrust algorithm for reputation management in P2P networks. In *Proceedings of the Twelfth International World Wide Web Conference (WWW)*, Budapest, Hungary, May 2003.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM (CACM)*, 21(7):558–565, July 1978.

[LCW+06] Harry C. Li, Allen Clement, Edmund L. Wong, Jeff Napper, Indrajit Roy, Lorenzo Alvisi, and Michael Dahlin. BAR gossip. In *Proceedings of the Seventh Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, November 2006.

[LMSW06] Thomas Locher, Patrick Moor, Stefan Schmid, and Roger Wattenhofer. Free riding in BitTorrent is cheap. In *Proceedings of the Fifth Workshop on Hot Topics in Networks (HotNets-V)*, Irvine, CA, November 2006.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, July 1982.

[MCWG95] Andreu Mas-Colell, Michael D. Whinston, and Jerry R. Green. *Microeconomic Theory*. Oxford University Press, June 1995.

[MGGM04] Sergio Marti, Prasanna Ganesan, and Hector Garcia-Molina. SPROUT: P2P routing with social networks. In *Proceedings of the First International Workshop on Peer-to-Peer and Databases (P2PDB)*, Heraklion, Greece, March 2004.

[MGLB00] Sergio Marti, T.J. Giuli, Kevin Lai, and Mary Baker. Mitigating routing misbehavior in mobile ad hoc networks. In *Proceedings of the Sixth Annual International Conference on Mobile Computing and Networking (MobiCom)*, Boston, MA, August 2000.

[MM02] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS)*, Cambridge, MA, March 2002.

[MMGC02] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: A read/write peer-to-peer file system. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, December 2002.

[Moj] MojoNation. `http://sourceforge.net/projects/mojonation`.

[MPDG08] Alan Mislove, Ansley Post, Peter Druschel, and Krishna P. Gummadi. Ostra: Leveraging trust to thwart unwanted communication. In *Proceedings of the Fifth Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, April 2008.

[MR97] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing (STOC)*, El Paso, TX, May 1997.

[MRL00] Alexander E. Mohr, Eve A. Riskin, 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 (JSAC)*, 18(6):819–828, June 2000.

[MRWZ05] Ratul Mahajan, Maya Rodrig, David Wetherall, and John Zahorjan. Sustaining cooperation in multi-hop wireless networks. In *Proceedings of the Second Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, May 2005.

[NM04] Antonio Nicolosi and David Mazières. Secure acknowledgment of multicast messages in open peer-to-peer networks. In *Proceedings of the Third International Workshop on Peer-to-Peer Systems (IPTPS)*, San Diego, CA, February 2004.

[NNS+05] Animesh Nandi, Tsuen-Wan "Johnny" Ngan, Atul Singh, Peter Druschel, and Dan S. Wallach. Scrivener: Providing incentives in cooperative content distribution systems. In *Proceedings of the ACM/IFIP/USENIX Sixth International Middleware Conference (Middleware)*, Grenoble, France, November 2005.

[NR01] Noam Nisan and Amir Ronen. Algorithmic mechanism design. *Games and Economic Behavir*, 35(1–2):166–196, April 2001.

[NT04] Nikos Ntarmos and Peter Triantafillou. SeAl: Managing accesses and data in peer-to-peer sharing networks. In *Proceedings of the Fourth IEEE International Conference on Peer-to-Peer Computing*, Zürich, Switzerland, August 2004.

[NWD03] Tsuen-Wan "Johnny" Ngan, Dan S. Wallach, and Peter Druschel. Enforcing fair sharing of peer-to-peer resources. In *Proceedings of the Second International Workshop on Peer-to-Peer Systems (IPTPS)*, Berkeley, CA, February 2003.

[NWD04] Tsuen-Wan "Johnny" Ngan, Dan S. Wallach, and Peter Druschel. Incentives-compatible peer-to-peer multicast. In *Proceedings of the Second Workshop on the Economics of Peer-to-Peer Systems*, Cambridge, MA, June 2004.

[Ora01] Andy Oram, editor. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies.* O'Reilly & Associates, Sebastopol, CA, March 2001.

[PCST01] Adrian Perrig, Ran Canetti, Dawn Song, and J. D. Tygar. Efficient and secure source authentication for multicast. In *Proceedings of the Eighth Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2001.

[PGW+06] J.A. Pouwelse, P. Garbacki, J. Wang, A. Bakker, J. Yang, A. Iosup, D.H.J. Epema, M. Reinders, M. van Steen, and H.J. Sips. Tribler: A social-based peer-to-peer system. In *Proceedings of the Fifth International Workshop on Peer-to-Peer Systems (IPTPS)*, Santa Barbara, CA, February 2006.

[PIA+07] Michael Piatek, Tomas Isdal, Thomas Anderson, Arvind Krishnamurthy, and Arun Venkataramani. Do incentives build robustness in BitTorrent? In *Proceedings of the Fourth USENIX Symposium on Networked System Design and Implementation (NSDI)*, Cambridge, MA, April 2007.

[PWCS02] Venkata N. Padmanabhan, Helen J. Wang, Philip A. Chou, and Kunwadee Sripanidkulchai. Distributing streaming media content using cooperative networking. In *Proceedings of the Twelfth International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, Miami Beach, FL, May 2002.

[RD01a] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object address and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, Heidelberg, Germany, November 2001.

[RD01b] Antony Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP)*, Chateau Lake Louise, Banff, Canada, October 2001.

[RFH+01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A

scalable content-addressable network. In *Proceedings of the ACM SIGCOMM*, San Diego, CA, August 2001.

[RH03] Timothy Roscoe and Steven Hand. Palimpsest: Soft-capacity storage for planetary-scale services. In *Proceedings of the Ninth Workshop on Hot Topics in Operating Systems (HotOS IX)*, Kauaii, Hawaii, May 2003.

[RR98] Michael K. Reiter and Aviel D. Rubin. Crowds: Anonymity for web transactions. *ACM Transactions on Information and System Security (TISSEC)*, 1(1):66–92, November 1998.

[RWW05] Michael K. Reiter, XiaoFeng Wang, and Matthew Wright. Building reliable MIX networks with fair exchange. In *Proceedings of the Third Applied Cryptography and Network Security Conference (ACNS)*, New York, NY, June 2005.

[SB08] Robin Snader and Nikita Borisov. A tune-up for Tor: Improving security and performance in the Tor network. In *Proceedings of the Fifteenth Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2008.

[SBHJ06] Naouel Ben Salem, Levente Buttyán, Jean-Pierre Hubaux, and Markus Jakobsson. Node cooperation in hybrid ad hoc networks. *IEEE Transactions on Mobile Computing (TMC)*, 5(4):365–376, April 2006.

[SGG02] Stefan Saroiu, P. Krishna Gummadi, and Steven D. Gribble. A measurement study of peer-to-peer file sharing system. In *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking (MMCN)*, San Jose, CA, January 2002.

[SGM04] Qixiang Sun and Hector Garcia-Molina. SLIC: A selfish link-based incentive mechanism for unstructured peer-to-peer networks. In *Proceedings of the Twenty-Fourth International Conference on Distributed Computing Systems (ICDCS)*, Washington, DC, March 2004.

[SMK+01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of the ACM SIGCOMM*, San Diego, CA, August 2001.

[SNDW06] Atul Singh, Tsuen-Wan "Johnny" Ngan, Peter Druschel, and Dan S. Wallach. Eclipse attacks on overlay networks: Threats and defenses. In *Proceedings of the Twenty-Fifth Annual IEEE Conference on Computer Communications (INFOCOM)*, Barcelona, Spain, April 2006.

[SPCY07] Michael Sirivianos, Jong Han Park, Rex Chen, and Xiaowei Yang. Free-riding in BitTorrent networks with the large view exploit. In *Proceedings of the Sixth International Workshop on Peer-to-Peer Systems (IPTPS)*, Bellevue, WA, February 2007.

[Swe02] Latanya Sweeney. Achieving *k*-anonymity privacy protection using generalization and suppression. *International Journal on Uncertainty, Fuzziness and Knowledge-based Systems*, 10(5), October 2002.

[The] The Distributed and Real-Time Systems Research Group, UNC. Data for the UNC HTTP traffic model. `http://www.cs.unc.edu/Research/dirt/proj/http-model/`.

[Var92] Hal R. Varian. *Microeconomic Analysis.* W. W. Norton & Company, third edition, March 1992.

[Var95] Hal R. Varian. Economic mechanism design for computerized agents. In *Proceedings of the First USENIX Workshop on Electronic Commerce*, New York, NY, July 1995.

[VCS03] Vivek Vishnumurthy, Sangeeth Chandrakumar, and Emin Gün Sirer. KARMA: A secure economic framework for peer-to-peer resource sharing. In *Proceedings of Workshop on Economics of Peer-to-Peer Systems*, Berkeley, CA, June 2003.

[Way97] Peter Wayner. *Digital Cash: Commerce on the Net.* Academic Press, second edition, March 1997.

[WM01] Marc Waldman and David Mazières. Tangler: A censorship-resistant publishing system based on document entanglements. In *Proceedings of the Eighth ACM Conference on Computer and Communications Security (CCS)*, Philadelphia, PA, November 2001.

[YGKX08] Haifeng Yu, Phillip B. Gibbons, Michael Kaminsky, and Feng Xiao. SybilLimit: A near-optimal social network defense against Sybil attacks. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2008.

[YKGF06] Haifeng Yu, Michael Kaminsky, Phillip B. Gibbons, and Abraham Flaxman. SybilGuard: Defending against Sybil attacks via social networks. In *Proceedings of the ACM SIGCOMM*, Pisa, Italy, September 2006.

[You98] H. Peyton Young. *Individual Strategy and Social Structure: An Evolutionary Theory of Institutions.* Princeton University Press, July 1998.

[ZCB96] Ellen W. Zegura, Kenneth L. Calvert, and Samrat Bhattacharjee. How to model an internetwork. In *Proceedings of the Fifteenth Annual IEEE Conference on Computer Communications (INFOCOM)*, San Francisco, CA, March 1996.

[ZHS+04] 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 (JSAC)*, 22(1):41–53, January 2004.

[ZZJ+01] Shelly Q. Zhuang, Ben Y. Zhao, Anthony D. Joseph, Randy H. Katz, and John Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *Proceedings of the of the Eleventh International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, Port Jefferson, NY, June 2001.