ABSTRACT

Title of dissertation: SYSTEMS-COMPATIBLE INCENTIVES

David Levin, Doctor of Philosophy, 2010

Dissertation directed by: Professor Samrat Bhattacharjee,
Department of Computer Science

Originally, the Internet was a technological playground, a collaborative endeavor among researchers who shared the common goal of achieving communication. Self-interest used not to be a concern, but the motivations of the Internet's participants have broadened. Today, the Internet consists of millions of commercial entities and nearly 2 billion users, who often have conflicting goals. For example, while Facebook gives users the illusion of access control, users do not have the ability to control how the personal data they upload is shared or sold by Facebook. Even in BitTorrent, where all users seemingly have the same motivation of downloading a file as quickly as possible, users can subvert the protocol to download more quickly without giving their fair share. These examples demonstrate that protocols that are merely technologically proficient are not enough. Successful networked systems must account for potentially competing interests.

In this dissertation, I demonstrate how to build systems that give users incentives to follow the systems' protocols. To achieve incentive-compatible systems, I apply mechanisms from game theory and auction theory to protocol design. This approach has been considered in prior literature, but unfortunately has resulted in few real, deployed systems with incentives to cooperate. I identify the primary challenge in applying mechanism design and game theory to large-scale systems: the goals and assumptions of economic mechanisms often do not match those of networked systems. For example, while auction theory may assume a centralized clearing house, there is no analog in a decentralized system seeking to avoid single points of failure or centralized policies. Similarly, game theory often assumes that each player is able to observe everyone else's actions, or at the very least know how many other players there are, but maintaining perfect system-wide information is impossible in most systems. In other words, not all incentive mechanisms are systems-compatible.

The main contribution of this dissertation is the design, implementation, and evaluation of various systems-compatible incentive mechanisms and their application to a wide range of deployable systems. These systems include BitTorrent, which is used to distribute a large file to a large number of downloaders, PeerWise, which leverages user cooperation to achieve lower latencies in Internet routing, and Hoodnets, a new system I present that allows users to share their cellular data access to obtain greater bandwidth on their mobile devices. Each of these systems represents a different point in the design space of systems-compatible incentives. Taken together, along with their implementations and evaluations, these systems demonstrate that systems-compatibility is crucial in achieving practical incentives in real systems. I present design principles outlining how to achieve systems-compatible incentives, which may serve an even broader range of systems than considered herein. I conclude this dissertation with what I consider to be the most important open problems in aligning the competing interests of the Internet's participants.

SYSTEMS-COMPATIBLE INCENTIVES

by

David Levin

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2010

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The Internet is no longer the cooperative, technological playground it once was. Successful networked systems must account for potentially competing interests among Internet Service Providers (ISPs), companies, and even users [27]. Internet protocols in a sense define the rules of how these self-interested participants interact and compete. However, these rules often lack enforcement.

Despite the fact that cooperation cannot be taken for granted in the Internet, there are many remarkably popular systems that rely on cooperating users [106]. For example, TCP is predicated on end-hosts decreasing the rate at which they send traffic when they detect congestion. Similarly, BGP relies on ASes to truthfully report routes, offering no inherent guarantee against hijacking.

Decentralized systems: Freedom or anarchy?

Decentralized systems, such as TCP, BGP, and BitTorrent, have no single authority, and instead allow users to apply their own local policies.

The greatest feature of decentralized systems is that there is no single administrative entity imposing policies with which not all users might agree. Users need not trust a centralized authority with their private information, nor is there a single point of failure which must be carefully provisioned. Additionally, decentralization of policy gives users the freedom to choose how they use the system and the ability to experiment with protocol modifications. This freedom to modify and explore ultimately leads to better, more widely usable systems. For instance, BitTorrent's decentralized policies have allowed researchers to improve its performance [73] and broaden the set of users who can legally use it [79].

The lack of a single administrative entity is also the greatest *weakness* of decentralized systems. There is no "police force," no entity with a clear mandate to keep the system operational or to kick out the spammers and attackers. Users do not even have the option of placing their trust in a centralized authority because there is none. While the freedom to modify the protocol can lead to improvements, it can also lead to exploits. For example, TCP is susceptible to end-hosts ignoring congestion signals [5] or sending incorrect signals [109] to increase their own performance at the potential degradation of others'.

Many decentralized systems *assume* user cooperation but, despite considerable efforts, few *guarantee* it. When this fundamental assumption is violated, the global system properties may suffer; selfish participants will attempt to gain at the expense of others [1, 79], and when a system becomes sufficiently large or popular, misbehaving principals may want to break it for notoriety or profit [77].

The underlying position of this dissertation is that a centralized authority is not necessary to ensure cooperation among a large number of self-governed entities on the Internet. Instead, we can design and build a system's protocol to include guarantees for cooperation. Furthermore, these guarantees can, themselves, be decentralized, thus maintaining the original goals and intent of the system.

Guaranteeing cooperation

There are two fundamental ways to guarantee cooperation. The first is to simply lock down the system's protocols, effectively centralizing policy and restricting what users can do. A prime example of this is Skype [115], which uses a proprietary and secret protocol. Requiring a signed code base or a small trusted component [72] are other means of making uncooperative behavior impossible. While reasonable in some settings [12], we posit that unilaterally taking such an approach is not compatible with many systems. A signed code base requires a trusted authority to verify and attest to the quality of the implementation, and trusted components require additional assumptions about the hardware users have access to. Furthermore, restricting the actions users can take limits their ability to experiment with and improve existing protocols. The lack of applications that are compatible with or improve upon Skype is evidence that a closed protocol stifles innovation.

The second fundamental way to guarantee cooperation is to provide users with *incentives* to cooperate with one another. By aligning the goals of the individual with the goals of the system, users can be provided with a vested interest in the system as a whole. As such, users can be expected to follow not only the letter of the protocol, but the spirit of the protocol, as well. Providing incentives is preferable to an outright restriction of user behavior in that it maintains the openness and decentralization of the Internet while distributing the work of policing the system. Indeed, many systems, notably BitTorrent [30], have acknowledged the importance of incentives being built into the protocol, and have strived to do so. Throughout this dissertation, we will discuss systems whose goal it is to provide users with incentives, and various mechanisms to provide these incentives.

The role of economics in systems

Economic theory, in particular game theory and mechanism design, has provided powerful tools toward making decentralized systems robust to manipulation by selfish participants. Game theory provides a formal framework with which to understand how selfish, competing parties will interact, while mechanism design yields a rigorous methodology to create games with a particular set of desired outcomes in mind [94]. Such tools are steeped in decades, or in some cases millennia [8], of refinement and application.

However, decentralized systems represent a vastly different set of goals and requirements than prior applications of game theory and mechanism design. There have been many proposed incentive-compatible decentralized systems, some with strong provable properties of resilience to strategic manipulation. Unfortunately, although the mechanisms may be incentive-compatible, many are not necessarily *systems-compatible*. That is, many mechanisms are unrealistic in today's Internet because of mismatched assumptions; they require infeasible infrastructure, system-wide knowledge, or a trusted authority, for example.

Hypothesis

In this dissertation, I demonstrate that *the goals and techniques of economic mechanism design and decentralized systems can be aligned.* My evaluation of this hypothesis is constructive. I show how to build systems that provide incentives by introducing what I

	Repeated interactions	One-shot interactions
Symmetric interest	**BitTorrent** (Ch. 3)	**BitTorrent** (Ch. 3)
	PropShare	Bootstrapping
Asymmetric interest	**PeerWise** (Ch. 4)	**Hoodnets** (Ch. 5)
	SLAs	Dual auctions

Table 1.1: The space of possible peer interactions and the incentives proposed in this dissertation to support cooperation.

call *systems-compatible incentives*: mechanisms that not only have strong provable properties in the economic sense, but that are also consistent with a given system's design and assumptions. The true test of systems-compatibility lies in building and running the system in real network conditions. I demonstrate that systems-compatible incentives can be implemented and applied in real, existing systems. This is a relatively broad claim, and evaluating it requires us to consider—and *build*—a wide range of systems. I cover several systems in this dissertation, each of which covers what I find to be a unique area in the space of decentralized systems. My collaborators and I have built and evaluated each of the systems presented in this dissertation.

Dissertation outline

In Chapter 2, I define the class of incentive mechanisms that are applicable to networked systems. Whereas mechanism design seeks incentive-compatibility, we also seek compatibility with the goals and assumptions of the underlying system. Viewing mechanism design through the lens of a system designer, we see that many of the assumptions that are often taken for granted in a standard economic setting—such as a trusted auctioneer, an escrow service, or money itself—are incompatible in many systems settings. While this restricts the set of applicable incentives, it motivates a new space of mechanisms that are tailored to the realistic settings of today's Internet. The remainder of the dissertation seeks to populate and better understand this space.

Chapters 3, 4, and 5 each focus on a representative system, and introduce new mechanisms that provide incentives to users to truthfully contribute their resources to others. These systems cover a broad range of goals and assumptions. BitTorrent (Chapter 3) is a peer-to-peer system used to distribute a large file to many downloaders by having downloaders assist one another by trading pieces of the file with one another. PeerWise (Chapter 4) is an Internet routing overlay that reduces end-to-end latencies by allowing peers to forward through a relay instead of connecting directly to their destinations. Peers in PeerWise effectively "exchange bandwidth" by forwarding traffic for each other. Finally, we introduce a new system called Hoodnets (Chapter 5), a multi-homed, ad hoc wireless network that aggregates users' bandwidth and routes around poor cellular coverage. Hoodnets users achieve this by sharing their cellular access to the Internet with proximal users.

As we will see, different types of systems lend themselves to different types of incentive mechanisms. The choice of incentive depends on how a system's participants interact. There are two axes that classify all such interactions, which we show in Table 1.1. Participants exchange resources with one another. Thus the first axis captures whether or not two participants exhibit symmetric interest in one another's resources. BitTorrent peers exhibit symmetric interest in one another; peers in a given swarm have pieces of a file that they all wish to download, and actively trade with those who have pieces that they do not. PeerWise peers, on the other hand, exhibit asymmetric interest; peers request others to forward their traffic to achieve lower latency paths on the Internet, but peers may not always have traffic to send.

This one axis is not sufficient to classify all classes of interaction. The incentives in BitTorrent's trading algorithm and in PeerWise leverage, to some extent, the fact that peers will interact for an extended period of time. However, there are instances where peer interactions may be ephemeral, particularly in mobile applications, where peers may move in and out of range of one another. The second axis along which we classify peer interactions captures whether peers interact repeatedly or only once. Hoodnet peers interact over remarkably short intervals—as little as the time to forward a single packet. Unlike BitTorrent and PeerWise, wherein a peer can eventually reward others for their contributions, Hoodnet peers require immediate payment for their work, for fear that they may never interact with a given peer again.

As shown in Table 1.1, the three systems analyzed in this dissertation cover the full breadth of the incentives space. This will allow us to draw conclusions about the space of systems-compatible incentives as a whole. Additionally, the incentive mechanisms proposed in this dissertation can, we believe, be applied to a much broader range of systems than specifically discussed herein.

Chapters 6 and 7 turn to the potential shortcomings and limitations of incentives in practical systems. Chapter 6 addresses the limitations of using punishment as a disincentive of bad behavior in lieu of rewards as an incentive for good behavior. Punishment is compelling for system designers as it typically leads to simpler mechanisms and rigorous proofs of cooperation. However, punishments can have detrimental effects on a system's performance and, by design, introduces attacks to a system. We present the design and implementation of a file swarming system that uses punishment to ensure cooperation among peers. This system, Fair Optimal eXchange (FOX) [74], disseminates a large file to a large number of users in a way that is provably fair: each user downloads almost exactly as much as they upload, and punishes peers who fall below this one-for-one threshold. FOX is a prototypical application of punishments, and demonstrates a trade-off between strict fairness and performance. This trade-off is one of the unfortunate limitations of punishments as an incentive mechanism.

Chapter 7 asks: can peers be compelled to tell the truth using systems-compatible incentives? The answer to this is largely negative. Although truthfulness is a common property of many economic mechanisms, we demonstrate it is remarkably difficult to achieve in systems that do not have either extensive infrastructure for holding peers accountable for their actions nor monetary rewards. As our focal system, we again look at BitTorrent and show that, counter-intuitively, BitTorrent peers have incentive to intelligently *under*-report what pieces of the file they have to their neighbors.

We conclude this dissertation in Chapter 8 by reviewing the lessons learned—positive, negative, expected, and surprising—from the broad range of systems and incentives covered throughout the dissertation.

Intended audience

My intended audience of this dissertation covers a wide range of researchers and practitioners in both systems and mechanism design. System designers and implementors can apply the specific mechanisms I introduce to ensure proper operation of their systems' participants. This dissertation also informs those interested in designing new systems-compatible incentive mechanisms by describing why certain classes of incentives are or are not applicable to various types of systems. Throughout this dissertation, I outline the fundamental differences between what is acceptable in real, large-scale systems and what is assumed in certain mechanism design contexts. Some of these differences serve as restrictions—such as the lack of a centralized auction clearing house—and some as relaxations—it is reasonable to assume individual rationality when a user's actions are dictated by an application they have installed. Algorithmic game theorists can apply these lessons to their own work, ideally resulting in mechanisms that can be more directly applied to real systems.

The end result of this dissertation is a set of techniques to understand and improve upon the incentives in deployable systems. While this work in and of itself ought not be expected to put an end to self-interested parties' manipulation of others, it does play a crucial role in understanding what degrees of cooperation we *can* achieve on the Internet.

Chapter 2

Systems-Compatible Incentives

To align decentralized systems and mechanism design, we must first understand the goals and requirements of each. Also in this chapter, we discuss some of the potential limitations and relaxations that arise when combining the two.

2.1 Systems goals and requirements

The main distinguishing factor between distributed systems and decentralized systems is that decentralized systems allow for multiple policy domains. While a single principal may deploy a distributed system, a decentralized system consists of multiple principals, each of whom may have their own policies regarding trust, privacy, willingness to contribute, and so on. Designing systems that appeal to a wide range of policies and allow principals to expressively solve contentions in-band [27] is made all the more difficult without knowing these policies a priori.

In addition to addressing various user policies, a successful decentralized system must be able to scale with growing demand and participation. Otherwise, a system's popularity would be its own undoing. For example, requiring perfect, system-wide information such as the identity and status of each participant would render a system incapable of scaling to millions of users. Few could have predicted BitTorrent's popularity, but even its early design accommodated a large, ever-growing user base; each peer, for instance, maintains state on only a few dozen peers. The requirement of being able to scale limits the applicability of some forms of game theory; for example, players in a standard-form game reason based on common knowledge of all other players' utility functions and strategy sets.

2.2 Incentives goals and relaxations

Mechanism design seeks to provide incentives to selfish but rational principals to follow a given protocol. Such protocols involve truthfully reporting private information, and fairly providing resources to other, competing principals. Without incentives, selfish users will attempt to gain at the potential cost of others by lying about their private valuations or attempting to distribute their resources unfairly.

In attempting to apply mechanism design to Internet protocols, it is important to ask whether or not end-hosts can be considered both selfish and rational. One is said to be *selfish* if they seek to gain as much as possible, regardless of how that might affect others. A *rational* agent is one that, given a choice of multiple courses of action to take, will chose the one that benefits him the most. It should come as no surprise to the reader that human beings often do not act rationally. Indeed, game theoretical analyses typically do not hold when human players are involved. However, it is the software implementations of Internet protocols, not the end-users, making the protocol-level decisions. We therefore posit that it is generally reasonable to assume perfect rationality of system participants. A strategic BitTorrent client, for instance, may be designed to precisely follow a utility-maximizing strategy that end-users would not or could not perform on their own accord.

One acts selfishly to obtain greater utility despite the potential loss of utility of others. Protocol implementations that selfishly manipulate a system are one example of this; they seek greater performance for themselves at the potential degradation of performance of others. Users seek greater performance, and would be considered selfish only if they did so with the knowledge that this would hurt others' performance. However, implementations that selfishly manipulate a protocol generally do not inform their users that there is any loss of utility for others. BitThief [79] and BitTyrant [98] are two BitTorrent clients that offer users benefit over the standard BitTorrent implementation. BitThief downloads files without uploading, making the use of BitTorrent legal in some countries, while BitTyrant achieves faster download times by manipulating BitTorrent's trading protocol. Both of these systems hurt other users: BitThief peers take from the system without giving back, and we show in Chapter 3 that BitTyrant increases system-wide download times. Without knowledge of the potential negative impact these systems would have on the system as a whole, even a selfless user may choose selfish protocols based only the improvements they offer. Hence, although not all users may have selfish inclinations, we posit that it is reasonable to assume that users may be modeled as purely selfish because they are not aware of (or do not care about) the negative impact their application's protocol decisions may have on others.

2.3 Systems-compatible incentives

An incentives mechanism is compatible with a decentralized system if it does not violate the assumptions and requirements of the system. In particular, to be systems-compatible, an incentive mechanism should not impose undue communication or computation burden, and is itself decentralized. We will demonstrate in the remainder of this dissertation that many incentive mechanisms require a centralized, often trusted principal. Centralized solutions do not allow for the complete expression of policy for all peers; for instance, they may not be able to choose with which set of peers to share a file, how much (or little) personal information to expose, whether or not to encrypt their content, or issue their own currency. Thus, such mechanisms are not systems-compatible.

This definition of systems-compatibility is intentionally flexible, so as to accommodate a broad range of systems. For example, we consider in Chapter 5 several different auctions to be run every ten seconds on mobile devices. A combinatorial auction [34] would be the ideal incentive mechanism in that setting, but because they are more computationally intensive than today's mobile devices permit, they are not systems-compatible. However, combinatorial auctions may be compatible with a system running on desktop machines, or if the auctions were run far less frequently.

A thorough understanding of what limitations a system places on mechanisms, as well as what windfalls it brings, can help guide designers toward more robust systems. For example, Afergan demonstrated that a detail as seemingly mundane as the number of bits used to represent prices can have profound effects on participants' strategies [4]. Conversely, we demonstrate with Hoodnets (Chapter 5) that 802.11's omnidirectional broadcast can be leveraged to efficiently implement a sealed-bid auction. We believe that systems-compatibility is an increasingly important consideration in the study and design of incentive-compatible systems.

2.4 The systems-compatibility of money

To make the notion of systems-compatibility more concrete, let us consider money as an example.

Money is the cornerstone of economy. It is well understood in economics literature and practice how to leverage money to obtain a wide array of cooperative user behavior. We briefly review the vast benefits of applying money to settings of selfish participants. We also discuss some of the reasons why, unfortunately, money is not systems-compatible. We close this section with alternatives to true monetary systems, and directions for future work.

Throughout this dissertation, when we speak of “money” we are referring to “real” currency, such as dollars or euros, that have value outside of a given system.

2.4.1 What does money buy systems?

Money gives rise to immensely powerful mechanisms and results that are at best difficult without it. A common use of money is to elicit truthful statements from participants who have incentive to lie about their private information. One of the fundamental goals of auctions, for instance, is for participants to bid their true valuation of a good, so that the auctioneer can award the good to the participant with the highest valuation [119, 28, 49]. Peers in a decentralized system have private information, such as their willingness to perform work [67], job priorities [65], or valuation of a good [4, 41, 3]. Money could yield tangible benefits to these various systems problems, and some systems have been designed with money as a system primitive [127, 4, 41, 3, 75].

One of the most challenging problems of designing incentive-compatible systems is in handling cases of asymmetric interest. Two peers p and q exhibit *asymmetric interest* when p desires a service from q, but q desires something of differing value (perhaps nothing) from p. One of the defining benefits of money is that it adds liquidity; regardless of what interest peers have in one another in the context of the system, they are always interested in money. In effect, systems that allow for monetary payments do not suffer from asymmetric interest. Lottery trees [37] are a prime example of overcoming asymmetric interest. The goal of lottery trees is to encourage users to join, and solicit others to join, a system they may have otherwise no intention of joining. The reward to these users is the chance to enter a lottery for a tangible reward outside the system, which from a systems-compatibility perspective is equivalent to offering money.

2.4.2 Is money compatible with decentralized systems?

Given the many potential benefits of including money in a decentralized system, we seek to understand the extent to which money can be applied in real, deployable systems. Prior work has demonstrated what is possible when applying economic operating points to networks [3, 41, 75, 127, 21]. However, none of these prior systems has experienced widespread deployment. This is in part due to the fact that they require money exchange at certain time intervals, typically on a per-packet basis. Supporting such payments requires extensive infrastructure support, typically through a centralized money-clearing system, which is difficult to deploy and scale with increasing demand. Certainly, ISPs make extensive use of money, but typically on much larger, more easily supported time scales.

Another reason money is difficult to incorporate into decentralized systems is the legal concerns it raises. In the context of Lottery Trees, Douceur and Moscibroda briefly review some of the extensive legal issues involved in running a lottery, including the disparities of laws from region to region [37].

Last, we argue that monetary systems are inherently centralized. Users must access a centralized money clearing mechanism, such as PayPal, or a centralized bank. This is a clear violation of many decentralized systems' goal of having no centralized trust domain.

2.4.3 Alternatives to money

Digital currency

In an effort to maintain the basic semantics of money without requiring users to pay for the service, various digital currency schemes have been proposed. A digital currency replaces money with a *proof of work*, such as the solution to a computationally intensive mathematical problem [22], so as to limit the amount of money any one principal can generate.[1] Digital currency does not have value outside of a system, making it fundamentally different from money. A major ramification of this is that peers do not have as much incentive to preserve digital currency, because they cannot remove it from the system for external goods. This may in fact be beneficial for a system, in that it creates a closed economy from which users cannot arbitrarily remove liquidity.

One of the fundamental technical challenges of a digital currency system is ensuring that users cannot double-spend. Paper currency is difficult to double-spend because of the

[1]The term "digital currency" has also been used to refer to any electronic money transaction, such as credit cards. Here, we refer to it strictly as the proposed alternatives to money.

considerable effort in making bills difficult to copy or forge. However, in a digital setting, copying is trivial. Solutions to double-spending typically involve a third-party mediator to track the history of individual units of currency [120, 46]. While not necessarily introducing a high barrier of entry—peers would not, for instance, have to register a credit card—providing such mediators does requires considerable systems infrastructure.

These systems allow users to generate their own currency, which raises another fundamental technical challenge: ensuring that users do not flood the system with currency, thereby devaluing it. A standard approach is to rate-limit users' currency creation by requiring a proof of work [38], such as the solution to a cryptographic puzzle [96].

Mechanism design without money

Mechanism design without monetary payments is a rich area [91, Chapter 10] that, for the reasons discussed in this section, is receiving renewed interest. One way to broadly view this line of work is as an investigation into the role that money plays in the positive results obtained from mechanisms. One such result, *money burning* [53], acknowledges the infeasibility of incorporating money in a networked system, and proposes replacing money payments with decreased quality of service. For example, rather than place monetary bids in an auction, peers could bid the level of service degradation they are willing to accept. Clearly, such an approach comes at the cost of decreased social good; Hartline and Roughgarden demonstrate that optimal mechanisms typically involve money transfer [53].

Exploiting users' impatience

Although a general replacement of money may ultimately come at a cost of performance, there remain scenarios in which a money-less mechanism can ensure truthful reporting with very little negative impact to the system. We discuss here one such example: leader election among impatient participants.

Leader election can be viewed as a game of public good; at least one of the players must pay the cost of being the leader so that all may benefit. In a selfish environment, the challenge is to elicit truthful reports of peers' costs to act as a leader; with no mechanism in place, all participants would have incentive to inflate the costs they would incur so as to avoid having to serve other peers.

Lee et al. observe that in multi-hop wireless networks, truthful reporting can be obtained without monetary payments; in fact, it can be obtained without explicitly reporting any values whatsoever [67]. The result leverages the specific application domain: wireless nodes wish for routing paths to be found quickly, else they may experience prolonged disconnected operation. This observation allows for the application of the *volunteer's timing dilemma* (VTD) [121]. In VTD, one out of a set of players must volunteer to complete a job; none of the players wish to be the volunteer, but all of them benefit from there being a volunteer, and benefit more the more quickly someone volunteers. The VTD game translates a player's cost-to-volunteer into an amount of time to *wait* until volunteering; the game itself consists of silently waiting until one player—the one with the lowest cost—volunteers. Lee et al. demonstrate how to apply this to a practical system. The game is periodically run, so as to cycle the leaders and maintain high system-wide battery levels.

Interestingly, even without money, participants' dominant strategy is to act truthfully. This is similar to money-burning mechanisms in that participants degrade the network's service by silently waiting for their neighbors to volunteer. The system by Lee et al. demonstrates that, in practice, this sub-optimality can be amortized over time, and serves as a considerable improvement to extensive infrastructure changes [67].

To summarize, money is a remarkably powerful tool in ensuring cooperation among peers who may otherwise have no reason to interact. It is difficult to replace money in general, but recent work has demonstrated the power of tailored mechanisms.

2.4.4 When is money systems-compatible?

In the example systems we have discussed thus far, money has not been systems-compatible because it was not an *a priori* component of the system. Some systems already assume or accommodate money, such as pay services like Internet or cellular connections, eBay, and Lottery Trees [37]. An incentives mechanism is systems-compatible with respect to a given system so long as it does not violate that system's assumptions. Hence, when money—or any similar architectural component—is incorporated into the fundamental design of a given system, an incentives mechanism may leverage it while remaining systems-compatible.

For example, in Chapter 5, we consider how to provide incentives for users with cellular data access to forward traffic for one another. Currently, users must pay for such services. Forwarding traffic for another incurs a monetary cost, and it is therefore natural to charge money to cover this cost.

In summary, systems-compatibility has a rigid definition only within the context of a specific system. Understanding the space of systems-compatible incentives mechanisms therefore requires us to study a broad range of systems.

Chapter 3

Incentives Under Symmetric Interest

The first system we discuss, BitTorrent, involves a rather straightforward game of bilateral exchange. Users have pieces of a file and trade these pieces with one another in order to construct the whole file. The defining characteristic of trading in BitTorrent is *symmetric interest*; two peers only trade with one another if they are interested in one another, that is, if each one has pieces of the file that the other does not. Even given symmetric interest, peers may attempt to selfishly attempt to gain more from others while giving as little back as possible. Incentives play a crucial role in BitTorrent, motivating users to upload to others to achieve fast download times for all peers. Though long believed to be robust to strategic manipulation, recent work [98] has empirically shown that BitTorrent does not provide its users incentive to follow the protocol. We propose an auction-based model to study and improve upon BitTorrent's incentives. The insight behind our model is that BitTorrent does not use tit-for-tat as widely believed, but an auction to decide which peers to serve. Our model not only captures known, performance-improving strategies, but shapes our thinking toward new, effective strategies. We implement and evaluate a modification to BitTorrent in which peers reward one another with proportional shares of bandwidth. With experiments on PlanetLab, a local cluster, and live downloads, we show that a proportional-share unchoker yields faster downloads against BitTorrent and BitTyrant clients, and that under-reporting pieces yields prolonged neighbor interest. [1]

3.1 Selfish bulk data transfer with BitTorrent

BitTorrent [13] is a remarkably successful decentralized system that allows many users to download a file from an otherwise under-provisioned server. To ensure *quick* download times and *scalability*, BitTorrent relies upon those downloading a file to cooperatively trade portions, or *pieces*, of the file with one another [30]. Incentives play an inherently crucial role in such a system; users generally wish to download their files as quickly as possible, and since BitTorrent is decentralized, there is no global "BitTorrent police" to govern their actions. Users are therefore free to attempt to strategically manipulate others into helping them download faster. The role of incentives in BitTorrent is to motivate users to contribute their resources to others so as to achieve desirable *global* system properties—fast download times for all peers, resilience to failures, and so on—without having to resort to a centralized coordinator.

Though long believed to be robust to strategic manipulation, BitTorrent's incentives have recently come under scrutiny. Clients such as BitThief [79] and BitTyrant [98] empirically demonstrate that users do *not* currently have incentive to follow the BitTorrent protocol. These results are a surprising contrast to the general understanding that BitTorrent uses a tit-for-tat-like mechanism. We address this apparent contradiction by

[1] Portions of this chapter originally appeared in *BitTorrent is an Auction: Analyzing and Improving BitTorrent's Incentives*, SIGCOMM 2008, August 17–22, 2008, Seattle, Washington, USA.

considering the following natural questions: Can we rigorously show *why* BitTorrent is vulnerable to strategic manipulation? Can we develop *new incentive mechanisms* to make BitTorrent robust to a wide range of selfish gaming while retaining, or even *improving*, its performance?

Analyzing BitTorrent's incentives

We find that these questions have, to date, gone unanswered in large part due to a basic misinterpretation of BitTorrent's incentives. BitTorrent is widely understood to use tit-for-tat [30, 69, 83, 87, 101, 99].[2] The first broad contribution of our work is in developing a game-theoretic model of BitTorrent's incentives that shows that *BitTorrent does not use tit-for-tat*. We find that *an auction-based model is more accurate*. The difference is subtle, as BitTorrent does share some of the same properties of tit-for-tat. But we show that re-framing the existing mechanism as an auction is powerful in that (1) it captures the known, performance-improving strategies of BitThief and BitTyrant, (2) it reveals *new* means of strategically manipulating BitTorrent, and (3) it provides insight into the design of a more robust incentive mechanism.

The studies of BitTorrent's incentives have focused predominately on how to game the *unchoking algorithm* which peers use to determine how much to upload to others. Our auction-based model captures this algorithm, but also reveals another area for strategic manipulation: the *piece revelation strategy*, which dictates what pieces a peer tells others that it has. The standard practice is for a peer to truthfully report all of the pieces it has to other peers. To the best of our knowledge, we are the first to consider non-trivial piece revelation strategies. We draw inspiration from the economic phenomenon of *decoupling*,

[2]These are but several examples; that BitTorrent uses tit-for-tat is virtually a consensus in the literature.

in which a group of market countries shed their dependence on another by trading among themselves. We show that a BitTorrent peer has incentive to intelligently *under-report* the pieces it has and to intentionally reduce the multiplicity of others' pieces so as to keep peers uploading to it instead of to each other. With evaluation on PlanetLab, we demonstrate that strategic piece revelation can yield prolonged neighbor interest, which can result in faster download times. We also find that even when *all* peers under-report their pieces to one another, system-wide download times increase by a surprisingly small percentage.

In a system as complex as BitTorrent, there are potentially many components peers could game. We believe that the two we consider—the unchoking algorithm and the piece revelation strategy—constitute the essence of a "BitTorrent-like" file swarming system. To obtain a more complete understanding of BitTorrent's incentives, we also consider what role *piece rarity* plays in strategic behavior. We show experimentally that achieving a block monopoly is infeasible—even with a large fraction of colluding peers—and conclude that BitTorrent's rarest-first piece selection is a natural deterrent to such strategies.

Improving BitTorrent's incentives

The second main contribution of our work is in applying our theoretical understanding of BitTorrent to develop new, more robust incentives. Our auction-based model reveals a rather straightforward modification in the way a peer *clears* its auction, that is, in how much a peer rewards others for uploading to it. We consider a *proportional share* auction clearing, in which each player is rewarded with an amount of good proportional to how much it bid. This approach has recently received attention in resource allocation [65, 42], and has been shown to converge quickly to a market equilibrium [124]. Applying this straightforward model in practice introduces several technical challenges which we address, including how to find other peers with whom to trade, and how to bootstrap new participants.

We emphasize that our proportional share mechanism is intended to *replace* the current unchoking algorithm, not to game it like BitThief and BitTyrant. However, we show that proportional share performs better against BitTorrent peers than the standard protocol. Further, with the intent of incremental deployment, we present a solution that *does not require any modification to BitTorrent's wire protocol*; the only modifications we make are to the *local* decisions a peer makes. Our client is therefore usable today, and we show with extensive experiments on PlanetLab and live swarms that as more users adopt our client, system-wide performance improves.

Roadmap

The rest of this chapter is structured as follows. We present related work in Section 3.2. Section 3.3 contains the necessary background to our analysis: our goals, assumptions, and the pertinent aspects of the BitTorrent protocol. In Section 3.4, we present our auction-based model of BitTorrent and use it to prove several properties of BitTorrent, including: it is indeed not tit-for-tat and it is susceptible to Sybil attacks. In Section 3.5, we apply the proportional share mechanism as a BitTorrent unchoker, and prove several properties in the context of BitTorrent: that it is fair, not susceptible to Sybil attacks, and more resistant to collusion. We present in Section 3.6 several extensions to our proportional share mechanism we found useful in implementation. We have implemented our proportional share client, and evaluated experimentally on PlanetLab, a local cluster, simulations, and live swarms; we present these results in Section 3.7. In Section 3.8, we propose a bootstrapping algorithm that improves upon BitTorrent's optimistic unchoking by ensuring that new peers contribute to the swarm as soon as they join. Finally, we conclude in Section 3.9.

3.2 Related Work

File swarming has received significant attention from researchers and users alike, BitTorrent [13] in particular. Our work involves modeling, gaming, and improving BitTorrent's incentives.

3.2.1 Studies of BitTorrent's incentives

Cohen began the study of his BitTorrent protocol's incentives by demonstrating that tit-for-tat-based incentives make BitTorrent robust to strategic gaming [30]. However, Cohen later found strict tit-for-tat to come at too high a cost [29], and weakened the protocol's incentives to achieve better performance. Strategies to game BitTorrent to obtain faster downloads were not long to follow.

Gaming BitTorrent

BitTyrant [98] is a modification of the Azureus BitTorrent client that exploits the "last place is good enough" observation discussed in Section 3.4. Schneidman et al. [111] were the first to observe this strategy; BitTyrant is an empirical study of its effectiveness. BitTyrant attempts to find the smallest winning bid by beginning at some initial bid and then increasing or decreasing the bid by some small percentage. BitThief [79] studies the feasibility of downloading in BitTorrent without uploading. A BitThief client attempts to enter as many peers' optimistic unchoke slots as possible: the so-called large view exploit [112]. This strategy results in a tragedy of the commons; entering as many optimistic unchoke slots as possible is a rational strategy for all peers to make (it always improves the expected completion time), but when a significant percentage[3] of the peers run the BitThief client, overall performance will logically decrease.

Some have also considered implementation-specific attacks, such as uploading garbage data [111, 78]. We focus only on the components of BitTorrent's protocol that we find to be characteristic of its class of file swarming, not any particular implementation. Strategically gaming BitTorrent helps further the understanding of its incentives; our auction-based model (Section 3.4) is validated in part by the fact that it captures both BitTyrant and BitThief, and reveals new means of gaming BitTorrent.

Modeling BitTorrent's incentives

Others have considered game-theoretic models of BitTorrent. Coupon replication [83] has been used as a way to model trading in BitTorrent and to show that altruism does not play a critical role in file swarming systems' performance, nor is rarest-first block scheduling of critical importance. This model may prove too simplistic in modeling BitTorrent's incentives in a heterogeneous environment; by assuming that blocks are traded in discrete segments (coupons), it effectively assumes homogeneous bandwidth. Proportional share [42, 65, 124] has been studied in many contexts. Zhang and Wu show that proportional share in a BitTorrent-like system quickly achieves a market equilibrium. However, they do not consider any of the common externalities that can destabilize the equilibrium—such as block rarity and availability (Chapter 7) or churn—nor do they address bootstrapping

[3]The precise percentage depends on the number of unchoke slots per peer.

the system (Section 3.8).

3.2.2 Improving BitTorrent's incentives

Much work has gone into encouraging cooperation among selfish BitTorrent participants. We present related work based on the incentive mechanisms used to achieve this.

Tit-for-tat

Tit-for-tat is a common incentive mechanism in which peers provide blocks to those who have provided them blocks in the past. BitTorrent was originally described as using tit-for-tat [30]; we discuss BitTorrent's mechanism in more detail in Section 3.4. Jun and Ahamad [57] propose removing optimistic unchoking from BitTorrent in favor of a k-TFT scheme, in which peers continue uploading to others until the *deficit* (blocks given minus blocks received) exceeds some *niceness* number k. Their results show, as Cohen found with BitTorrent [70], that removing optimistic unchoking increases the average completion time, but does indeed punish free-riders more heavily. Jun and Ahamad observe that if free-riders *can* benefit in a system, then eventually the system will devolve into all free-riders, so a strong disincentive to free-ride is necessary to halt this "evolution." Garbacki et al. [45] consider an *amortized tit-for-tat* scheme that effectively allows contributions made while downloading one file to apply in a tit-for-tat-like manner to other files in the future. Other solutions to this problem generally involve monetary mechanisms.

Monetary mechanisms

Dandelion [113] is a file distribution protocol that uses currency and key exchanges through a centralized server to provide incentive for sharing across different downloads. By requiring centralized servers, Dandelion trades off scalability for incentive-compatibility.

BitStore [102] is an exploratory work into applying monetary, second-price auctions to bulk data transfer.

Both Dandelion and BitStore use currency to address what we call the *seeder promotion problem*: providing incentive to peers who have completed downloading a file to seed the file. This is a very important problem, as well; solving it would increase file availability. We believe the mechanisms presented in this chapter can complement future solutions to seeder promotion.

Topology-based reciprocation

FOX's structured, cyclic topology provides a means for a peer to punish nodes both upstream and downstream from it [74]. Ngan et al. [89] take a similar approach by having peers periodically re-form a SplitStream [20] structure, with the hopes that peers who were downstream from a cheater will later be upstream from it, thereby giving it the ability to punish. In addition to the overhead of re-forming the structure, peers must wait for a new restructuring to even have the possibility of punishing free-riders. In both of these systems, fairness is defined very strictly; peers receive as much as they give. We find this to be unnecessarily strict, as there may be highly provisioned nodes who are willing to give much more to the system as long as they can download more; FOX, for one, does not provision for this.

Bootstrapping file exchange

We present a mechanism in Section 3.8 to provide newly joined peers an initial set of pieces of the file to trade. The standard mechanism in BitTorrent is *optimistic unchoking*; each peer uploads to at least one other peer at random, with a weighted preference to new nodes. Optimistic unchoking is the basis of BitThief [79] and the large view exploit [112], while our mechanism encourages peers to trade immediately. Our mechanism is similar in nature to Dandelion [113], but is much lighter weight; it need only be applied when a node has no currency—either when it first joins the system or when it has no blocks of interest to its neighbors—and not for every subsequent block. Also, the technique does not require any trade-off of scalability for incentives, and can be applied to any scenario employing simultaneous exchange of blocks with node churn. *Super-seeding* is a new feature in some BitTorrent clients in which a seeder will upload block b to some new peer p, but will not upload any more blocks to p until the seeder observes that other peers have block b, and hence p must be uploading to others. Our mechanism differs from super-seeding; since ours applies to selfish leechers, not altruistic seeders, we cannot afford to give a block and hope for one to be returned later. Instead, our mechanism ensures that new peers upload blocks *at the same time* as downloading. Further, super-seeding is clearly susceptible to a Sybil attack, in which p requests b, then with Sybil p' states that it has block b. We show that our mechanism is resilient to Sybil attacks.

3.3 Background and Goals

In this section, we provide a basic overview of the portions of the BitTorrent protocol pertinent to our study of its incentives. Our description of BitTorrent is intentionally broad so as to capture a broader class of "BitTorrent-like" protocols. We also define the goals of the individual users—their *utility*—and the goals of the system designer—the *social good*. The goal in designing an incentive mechanism is to achieve some desirable, provable properties of the social good, while appealing to users' selfishness.

3.3.1 BitTorrent basics

BitTorrent peers are classified on a per-file basis as *leechers* if they are downloading (and uploading) the file and *seeders* if they have the entire file and are uploading (only) to leechers. A *tracker* stores a small amount of state to assist leechers in discovering other peers. Files in BitTorrent consist of *pieces* which in turn consist of *blocks*. A leecher ℓ is *interested* in another peer p if p has pieces that ℓ does not; similarly, ℓ finds p *interesting*. All leechers are interested in all seeders.

BitTorrent peers may maintain open connections to multiple *neighbors*, but generally only upload to, or *unchoke*, a small number of them. Users may adjust this number of *unchoke slots*, but it is generally either some small constant (4) or some function of their upload bandwidth. The *unchoking algorithm* dictates to whom and how much to unchoke. To bootstrap, the unchoking algorithm consists of at least one random peer to *optimistically unchoke* regardless of that peer's contribution.

Peers inform one another of the pieces they have; when they first connect to a neighbor, they send a bit array of their pieces, called a *bitfield*, which they later update with per-piece *have messages*. All peers maintain an estimate of each piece's *availability*: a count of how many neighbors have that piece. When a peer p begins unchoking leecher ℓ, ℓ informs p which of p's blocks it wishes to receive next. The common strategy is *rarest-first*, in which ℓ prioritizes the pieces it views as least available.

3.3.2 Assumptions: Selfishness and rationality

The value that leecher ℓ gains from a file swarming system can be naturally defined to be how quickly ℓ downloads the file. We can thus define ℓ's utility u_ℓ to be the average download speed: if it takes ℓ time T_ℓ to download a file of size F then $u_\ell = F/T_\ell$. Note that we do not define utility at some specific time t, as that would not capture the fact that faster download speeds at the beginning of a download can be offset by poor download speeds at the end. We assume that *all leechers are selfish*—they each attempt to maximize their own utility—and *rational*—for any two strategies, ℓ will choose the one that yields greater expected utility.

Our definition of ℓ's utility does not capture the leecher's cost of uploading; this is intentional. We do not believe it is a leecher's goal to minimize the amount it has to upload in order to download as quickly as possible. Instead, we rely, as do existing BitTorrent clients, on the user to specify the amount of upload capacity they wish to allow BitTorrent to allot. Users are thus expected to find their own best *return-on-investment*, that is, the ideal ratio of utility (download speed) to cost. A peer's upload costs can come in many forms, for instance: (1) actual costs to send, as in a price-per-byte cell phone data plan, or (2) the contention with other networking applications' performance. Upload costs can therefore be extremely complex and change over time. We believe it is best to not include them in a formal definition of u_ℓ, and to instead allow the user to set her own upload capacity. In game theory parlance, the user's chosen upload capacity would represent her *type*, and the optimal advertised type would depend on externalities of which the BitTorrent application is unaware.

3.3.3 Goals: Desirable system properties

As system designers, we prefer outcomes that maximize some notion of the *public good*, which may in fact be at odds with selfish peers' individual strategies. A goal of our work is to understand the *price of anarchy*: the difference between the social good obtained from fully cooperative peers and that obtained from selfish peers. There are many ways to define public good: max-min utility, max-average utility, max-average return-on-investment, and so on. Strict “get exactly as much as you give” fairness properties seem to require undesirable overhead, such as extensive bookkeeping [88], topology constraints [89, 74], or a monetary system [120]. Further, we find that strict fairness requirements can degrade overall performance. In FOX [74], for instance, peers are motivated to give precisely as much as they receive. Under heterogeneous network conditions, higher-provisioned FOX peers may not have incentive to give more to the system, even though they would have been willing to, if it led to faster downloads.

We favor a simpler, more practical definition of fairness: *the more a peer gives, the more it gets*. Consider the ramifications this fairness property would have on the social good; a highly provisioned peer will upload as much as possible to download as quickly as possible. While serving the selfish peer, the rest of the peers benefit because the highly provisioned peer gives as much as it can. Legout et al. [69] suggest a similar, relaxed form of fairness, and conclude that BitTorrent’s current rarest-first and unchoking strategies suffice. We arrive at a different conclusion in section 3.4.3: that BitTorrent’s incentives are not fair, even under this relaxed definition.

3.4 BitTorrent as an Auction

Auctions offer a natural model of interactions among selfish BitTorrent peers. In such auctions, each peer places bids in the form of bandwidth to its neighbors, who in return give bandwidth as the good. With an auction-based model, we expect to gain insight into

BitTorrent's incentive structure. Surprisingly, though our specific model is quite simple (a two-line algorithm), it is validated by the fact that it captures recent attacks on the BitTorrent protocol, and reveals new attacks as well. Thus, while we do not expect our model to be the most accurate representation of BitTorrent, it does provide the necessary motivation and insight into a new solution (Section 3.5). In this section, we use our model to analyze what incentives BitTorrent gives selfish clients to cooperate, and BitTorrent's susceptibility to gaming.

3.4.1 An auction-based model

We propose the following auction-based model of BitTorrent. Each peer i separates time into *rounds*[4], which are not synchronized among peers. At round t, i runs the auction in Algorithm 1.

Algorithm 1 BitTorrent's current auction.

```
1. Run an auction for $i$'s bandwidth; accept bandwidth $b_j(t-1)$ from *interested* peer
   $j$ as $j$'s bid.
2. Send $1/s$ fraction of $i$'s total outgoing bandwidth to each of the highest $(s-1)$
   *interesting* bidders from the previous round, as well as one other *interested* peer at
   random.
```

3.4.2 BitTorrent is not tit-for-tat

BitTorrent has historically been described as employing tit-for-tat. The tit-for-tat strategy in a repeated game is defined as cooperating in the first round, and, in every subsequent round, replicating the opponent's strategy from the previous round [9]. Though similar to tit-for-tat—in that i sends predominately to peers that send to it—Algorithm 1 differs fundamentally in terms of *to whom* i sends.

[4]Azureus uses 10 second rounds.

To see the difference, consider a rational peer p's response to node i playing Algorithm 1. The ideal strategy against tit-for-tat is to cooperate in every round [9]. BitTyrant [98] is built off of the following observation of p's behavior: *It is in a rational player p's best strategy to bid as little as possible to still be one of the $(s-1)$ (deterministic) winners of i's auction.* BitTyrant finds this minimal amount by adjusting their estimate by small multiplicative factors; Piatek et al. observe that binary search is not suitable as node churn can rapidly alter peer reciprocation. Further, Piatek et al. find that, with a finite upload bandwidth "budget," one should favor uploading to peers who offer a high return on investment, potentially uploading none to peers that offer poor returns.

That BitTorrent does not employ tit-for-tat is not in and of itself a bad thing. In fact, this choice was intentional, to improve performance [29]. However, it brings to light the importance of understanding precisely what properties BitTorrent's incentive structure offers.

3.4.3 BitTorrent is not fair

There are many definitions of fairness. Our proposed, natural fairness property from Section 3.3.3 can be formalized as follows: if peer p_1 uploads more to peer q than does peer p_2, then q should reward p_1 more than p_2. This is clearly not the case in BitTorrent. Only the top $(s-1)$ uploading peers are guaranteed to receive any data from q, and each of these receive the same allocation: $1/s$ of q's bandwidth. Further, of the top $(s-1)$ uploaders to q, as they upload more, they are not guaranteed to obtain more. Once a highly-provisioned peer wins all of the s auctions in which it bids, it has no incentive to bid more in these auctions.

3.4.4 BitTorrent is susceptible to Sybil attacks

A Sybil attack [36] consists of a single host representing itself as many peers in an attempt to gain more from the system than it could as a single peer. BitTorrent is susceptible to two classes of Sybil attacks. The first is in regards to optimistic unchoking. By creating S Sybils $\{\sigma_1 \ldots, \sigma_S\}$ and using each of them to request to be in others' optimistic unchoking slots, the host will obtain in expectation S times more optimistic unchoking slots. The only Nash equilibrium of such an attack is for all peers to create as many Sybils as possible and, as in BitThief [79], to request to be optimistically unchoked by as many other peers as possible. Indeed, for every peer, this is a *dominant strategy*. Unfortunately, this results in a tragedy of the commons, like in BitThief (see Section 3.2). We address this problem with a proposed bootstrapping mechanism that would remove the need for optimistic unchoking (Section 3.8).

The auction in Algorithm 1 reveals another class of Sybil attacks to which BitTorrent is susceptible, but which, to the best of our knowledge, has yet to be exploited by any selfish clients. Recall that a rational peer has incentive to come in last, $(s-1)$'th, place in the auction so that it receives the *same* good for the cheapest price. To obtain *more* good, the peer would have to win more than one slot. Let c_i be the current bid on (cost of) slot i, $c_{s-1} \leq c_{s-2} \leq \cdots \leq c_1$, and let ϵ be the smallest increase such that if a peer were to bid $c_i + \epsilon$, it would win slot i. The optimal strategy of rational peer p with upload capacity U_p is:

Algorithm 2 A Sybil attack on the BitTorrent .

```
1. Find the maximum $k$ such that $k \cdot (c_{s-k} + \epsilon) \leq U_p$.
2. Create $k$ Sybils, $\{\sigma_1, \ldots, \sigma_k\}$, and with $\sigma_i$ bid $c_{s-k} + \epsilon$.
```

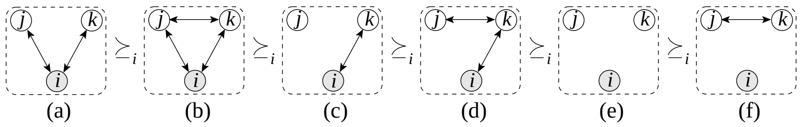


Figure 3.1: Node i prefers to be as interesting as possible, and more interesting to its neighbors than its neighbors' neighbors.

To summarize Algorithm 2, p would attempt to come in last (s'th) place, $(s-1)$'th place, …, and $(s-k+1)$'th place. To do so, p would have to outbid the current $(s-k+1)$'th place peer with each of p's bids, or else the $(s-k+1)$'th place peer would obtain one of the last k places.

The feasibility of Algorithm 2 in practice depends on the number of the victim's upload slots (s), the other peers' bids, and the attacker's upload capacity. The best case scenario for an attacker p would be to find a peer who is currently receiving many small bids, in which case p could potentially win all of the bids at that peer. The worst case scenario for an attacker p would be where p could only afford the last-place slot at any other peer; in this case, Algorithm 2 is identical to BitTyrant. Hence, we can view this Sybil attack as a *generalization of BitTyrant*.

In a sense, BitTorrent could be made more fair if each peer employed this Sybil-based strategy; if a peer so desired, it could upload more (with more Sybils) to a fast neighbor to download more. The mechanism we present in Section 3.5 has the similar property of "the more you give the more you get."

General-purpose solutions to Sybil attacks could apply, such as money [120] or proofs of work [38]. We show in Sections 3.5 and 3.8, however, that there exist natural incentive schemes that are Sybil-proof and would not require additional infrastructure or significant computational overhead.

3.4.5 BitTorrent's susceptibility to collusion

BitTorrent's auction-clearing mechanism is open to collusion. To date, selfish BitTorrent clients such as BitTyrant and BitThief have considered only non-collusive strategies.[5] We briefly sketch here how one could collude against BitTorrent. In Section 3.5, we present a mechanism that is less susceptible to this class of gaming.

A set of colluding peers can form a *coalition* $\mathcal{C}$ against a set of victims $\{v_1, \ldots, v_k\}$ by simply agreeing with one another to upload only nominal amounts to each v_i. The coalition is built on the premise of "turning victims into seeds" by uploading only a nominal amount of data to the victims and getting a full unchoke slot in return.

The feasibility of such an attack depends on how many "cheap slots" victim v_i has. In the best case scenario for a coalition $\mathcal{C}$, each v_i would currently not be receiving any bids, in which case the coalition could effectively act as the Sybil attack in Algorithm 2 and achieve all of each v_i's unchoke slots. In the worst case scenario, each v_i with s_i slots would be receiving at least s_i large bids. Put another way, for a victim v_i to be resilient to collusion, it must have s_i neighbors who are not participating in the coalition. We present a mechanism in Section 3.5 in which collusion fails as long as a *single* neighbor is not in the coalition.

3.5 Clearing Auctions with Proportional Share

The auction-based model of Section 3.4 motivates a new means of clearing BitTorrent peers' auctions. We extend recent results on the market equilibria achieved by *proportional share* [124]. Let $b_x^y(t)$ denote the amount of bandwidth x gives to y during round t.

[5]Though BitTyrant peers ramp allocations to one another, they do not directly collude against BitTorrent peers.

Algorithm 3 captures a proportional share auction, run at round t.

```
Algorithm 3 Proportional share auction clearing.
  1. Run an auction for $i$'s bandwidth; accept bandwidth $b_j^i(t)$ from peer $j$ as $j$'s bid.
  2. Let $B_i$ represent $i$'s total available upload bandwidth. Send to peer $j$ his proportional share:
        $b_i^j(t) = B_i \cdot \frac{b_j^i(t-1)}{\sum_k b_k^i(t-1)}$.   (3.1)
```

Clearly, the prop-share auction requires bootstrapping; peers must give some initial amount of goods to one another to obtain any in return. Though similar to BitTorrent's optimist unchoking in that a peer must initially give without getting anything in return, the amount of "altruism" of a prop-share client is much less. A peer could give an arbitrarily small amount, e.g., a single block. In the subsequent round, a prop-share client will ramp up its allocations according to Eq. (3.1). This bootstrapping mechanism also serves as a means of discovering new peers; we return to this point in the context of "finding a good deal."

Our extensions to Zhang and Wu's work [124] focus predominately on how this simple mechanism can be applied to dynamic network settings. A successful mechanism must maintain BitTorrent's proven robustness to such conditions.

We stress that prop-share is intended to be a *replacement* of BitTorrent's current auction clearing mechanism, not a strategy to game existing BitTorrent clients. As we showed in Section 3.4.4, BitTyrant with a Sybil attack is the ideal strategy against BitTorrent, and we do not expect prop-share to perform as well as this strategy. In the rest of this section, we show that prop-share is (nearly) a Nash equilibrium, and is thus not susceptible to a single peer's deviation.

3.5.1 Best response to prop-share

We can capture peer i's *best response* to all other nodes playing proportional share as follows:

$$\begin{aligned} \text{maximize} \quad & \sum_j B_j \cdot \frac{b_i^j(t)}{\sum_k b_k^j(t)} \\ \text{s.t.} \quad & \sum_j b_i^j \le B_i \ \text{ and } \ \forall k : b_i^k \ge 0 \end{aligned}$$

An immediate observation of this nonlinear program is that each peer has incentive to allocate all of its upload bandwidth, that is, $\sum_k b_i^k = B_i$. The solution to this nonlinear program [42] involves finding the bid to peer j at which the marginal benefit of increasing that bid is less than the marginal benefit of increasing the bid to some other peer k.

The best response to all peers playing prop-share is distinct from prop-share; that is, *prop-share is not always a Nash equilibrium.* This is because there may be some peers whose marginal benefit never exceeds others', and the best response is to never send to these peers. Conversely, with prop-share, peer i will provide bandwidth to *all* peers that uploaded to i in the previous round.

Computing the best response to other nodes playing prop-share requires peer i to know each of its neighbors' upload capacity, B_j, and the sum bids its neighbors have received from other peers. This information is reasonable to assume in the setting Feldman et al. [42] studied, wherein the auctioneers are not profit-maximizing, but this is clearly not the case in file swarming. One could envision trying to estimate these values, but the accuracy of such an estimation would be at best difficult to ensure, and would likely require extensive book-keeping.

3.5.2 Prop-share is enough

We show that simply employing prop-share with incomplete information achieves nearly the same return on investment as having perfect information, even against perfectly-informed peers. To demonstrate this, we simulated the scenario in which all peers except for peer p play prop-share, and compare p's allocations and resulting download speed when it plays prop-share versus the best-response algorithm provided by Feldman et al. [42]. In the case of best-response, we allowed p to have *complete information* of other peers' bandwidth, B_j, and bids, b^j_{-i}. For prop-share, such complete information is unnecessary, as p needs only to know the local information of how much each peer gave to p in the previous round. We present values averaged over 30 runs, varying the number of peers as well as p's bandwidth. Each game lasted for 30 iterations, but we observed that they converged after just a few rounds, confirming previous results [42, 124].

First we consider the question: does the best-response strategy result in similar allocations to those of prop-share, that is, are they effectively the same strategy? Figure 3.2 shows the average relative difference between best-response and prop-share allocations; on a per-peer basis, the best-response allocation differs by an average of roughly 30% from prop-share, and decreases as p has increasing bandwidth. Put simply: *best-response and prop-share strategies result in vastly different bandwidth allocations*. Even as a peer's "bargaining power" increases, best-response remains consistently "different" from prop-share, which we demonstrate by allowing p's bandwidth to vary from .1 to 40 times the average system-wide bandwidth.

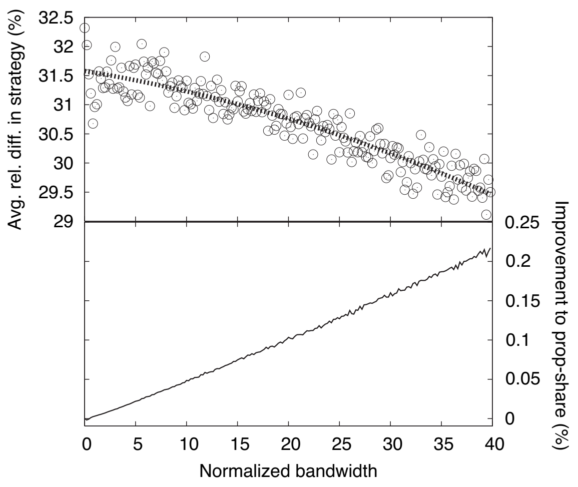


Figure 3.2: Prop-share is distinct from the best response. Though significantly different, completion times were consistently no worse than 0.25% longer.

Surprisingly, the clear difference between best-response and prop-share results in little improvement to utility. Best-response improved download speed by significantly less than 1% in all of our simulated scenarios. These results lead us to conclude that *prop-share is the preferred response to all other peers playing prop-share* because it achieves download speeds nearly equal to best response and, importantly, does not require complete information.

3.5.3 Prop-share is Sybil-proof

BitTorrent's auction is susceptible to Sybil attacks because its auction returns discretized goods (Section 3.4.4). We show that prop-share, on the other hand, is resilient to Sybil attacks. Note that a Sybil attack applies only to a given neighbor v. Let y_v denote the sum of other peers' bids at v. Suppose peer p were to create S Sybils, $p_1, \ldots, p_S$, where p_i contributes c_i to some victim neighbor v, for a total contribution $C = \sum_{i=1}^{S} c_i$. To show that prop-share is Sybil-proof, it suffices to show that, for a fixed amount of contribution C, p will receive the same amount of bandwidth for any set of Sybils, including the set of size 1. Indeed, for an arbitrary set of Sybils, p will in turn receive from prop-share-playing node v:

$$\sum_{i=1}^{S} \frac{B_v \cdot c_i}{\sum_{j=1}^{S} c_j + y_v} = \frac{B_v \cdot \sum_{i=1}^{S} c_i}{\sum_{j=1}^{S} c_j + y_v} = \frac{B_v \cdot C}{C + y_v} \tag{3.2}$$

In practice, Sybils perform even worse against a prop-share client. Each Sybil would require additional communication overhead for transmitting various protocol messages: declaring interest, listing the pieces they have, etc. Eq. (3.2) shows that Sybils would not improve performance even without such overhead, and would in practice result in poorer performance, as the peer would be forced to spend more of its budget (bandwidth) on protocol messages in lieu of data.

3.5.4 Prop-share is (more) collusion-resistant

Coalitional strategies would succeed against BitTorrent because no colluding peer has incentive to upload more to its respective victim; doing so would not result in greater download speeds. This property does not hold in prop-share, and is the basis for prop-share's collusion resistance. Consider a coalition $\mathcal{C}$ against a victim v who is playing prop-share. Suppose that, if the coalition were to play prop-share, then peer $i \in \mathcal{C}$ would have uploaded b_i^p to p and received fraction $f_i = \frac{b_i^p}{\sum_k b_k^p}$ of p's bandwidth. Let $f_{min} = \min_{i \in C}\{f_i\}$. Ensuring that each member of $\mathcal{C}$ obtains as much bandwidth from p as if each member were to strictly play prop-share is one way to keep the coalition from dissolving. To achieve this, each coalition member i can agree to upload bandwidth $\epsilon f_i / f_{min}$ to p, where ϵ is some nominal amount of bandwidth. p will thus return to i a fraction of p's bandwidth equal to

$$\frac{\epsilon f_i / f_{min}}{\sum_{j \in \mathcal{C}} \epsilon f_j / f_{min}} \;=\; \frac{f_i}{\sum_{j \in \mathcal{C}} f_j} \;=\; \frac{f_i}{1} \;=\; f_i$$

as long as the only peers uploading to p are the members of the coalition. If even a single peer $k \notin \mathcal{C}$ were to bid at p during $\mathcal{C}$'s collusion, k stands to gain a large return on investment. Indeed, as the coalition becomes more beneficial to its users ($\epsilon \to 0$), its susceptibility to being dissolved increases; k's share of p's bandwidth as the coalition's nominal bandwidth decreases is

$$\lim_{\epsilon \to 0} \frac{b_k}{b_k + \sum_{j \in \mathcal{C}} \frac{\epsilon f_j}{f_{min}}} \;=\; \frac{b_k}{b_k + 0} \;=\; 1$$

Hence, as long as there is a single peer not in $\mathcal{C}$, the coalition as a whole will have to dissolve, in which case all the members of $\mathcal{C}$ play prop-share. Compare this to BitTorrent, in which a node with s slots needs s non-colluding peers to dissolve a coalition.

While this demonstrates prop-share's natural defense against collusion, we discuss an extension in Section 3.6 that provides prop-share greater resilience.

3.6 Implementation

A nice property of the proportional share mechanism in Section 3.5 is its simplicity; each peer needs only to recall what its neighbors have most recently sent, and reply proportionally. An evaluation of proportional share's incentive properties is thus rather straightforward, and proven to be strong in theory. It is not unreasonable to assume that providing strong incentive properties would result in a decrease in performance. To study this, we have implemented a proportional share client, and evaluate it on PlanetLab and live swarms in Section 3.7. The core of the mechanism is the same in implementation as in Algorithm 3. We present in this section additional features we have found important in implementation.

3.6.1 Finding a good deal

Some peers are better deals than others, in the sense that some have higher values of $B/\sum_{j\neq i} b_j$. Zhang and Wu [124] assume that a node's neighbors are given, and show that proportional share converges quickly to an equilibrium. In reality, peers may learn of others' and *strategically* change their neighbor set. An optimal strategy would require a peer to have global knowledge of all other peers. The "large view exploit" [112], which BitThief [79] employs, attempts precisely this by continually requesting peers from the tracker, but with the goal of being optimistically unchoked by as many peers as possible. In fact, it is in any BitTorrent peer's best interest to learn of as many peers as possible so as to try to download from those with which it has a better connection.

However, such global knowledge does not scale, and instead each peer operates on a smaller subset of neighbors. Instead, in our implementation, we do not modify the number of neighbors a peer gets from the tracker, but a combination of prop-share and the large view exploit would, we believe, result in a more efficient market equilibrium.

Bootstrapping prop-share allows peers to "research" new neighbors and potentially find those with better return on investment, that is, larger values of $B_j/\sum_k b_k^j$. Zhang and Wu's analysis [124] assumes a fixed topology and initialize everyone's sharing to some random value. However, in practice, node churn variable traffic conditions require peers to discover new neighbors. We propose that peers allot a fraction (80% in our implementation) of their bandwidth to returning proportional shares to their neighbors, and use their remaining "budget" to research new neighbors. This requires no changes to the BitTorrent wire protocol, and is what we use in our implementation.

3.6.2 What have you done for me lately?

Recall that a prop-share peer runs its auction (Algorithm 3) once per round, using only the information of its neighbors' contributions from the previous round. This simple mechanism can lead to oscillations. Suppose for instance that at round t, peer p decides to "research" his neighbor q as above. In the following round, q will reply with p's proportional share, but since q did not upload to p in the previous round, p uploads none to q. This will clearly continue oscillating unless there is some external force to bring them to convergence. In our implementation, we employ a weighted average of a node's neighbors' four most recent contributions, and reward peers proportionally based on this.

3.6.3 Fighting collusion with ratio caps

Although prop-share is *more* collusion-resistant than BitTorrent (Section 3.5.4), there still exists the possibility for a large coalition to extort a victim's bandwidth at little cost. Protecting peers from collusion is difficult, as it is often unclear whether nodes are, as a coalition, appearing slow to some victim, or whether they are actually slow. There is a very interesting trade-off here: One could err on the side of caution, and better protect peers from periods of attack. Alternatively, one could err on the side of the social good, allocating large amounts to new neighbors early at the risk of them not reciprocating the favor.

In our implementation, we simply cap the amount of bandwidth a peer gives to any of its neighbors to a factor, f. This is similar in nature to k-TFT [57], but differs largely in the sense that, in our implementation, it is intended to only be applied when there is a high disparity. When a peer attempts to undersell another peer, this mechanism will limit that peer to achieving at most f times the amount of bandwidth it sends. When two peers wish to upload to one another, the multiplicative factor ramps up the allocation exponentially fast.

3.7 PropShare Evaluation

In this section we present our experimental evaluation of our PropShare client.

We emphasize the goal of our PropShare client: *to maintain robust incentive properties without sacrificing speed.* The previous sections of this chapter show PropShare's resilience to many forms of strategic gaming. Demonstrating these points experimentally is difficult if not impossible, as one failed attempt at gaming a system is hardly proof that it is impervious to strategic manipulation. The auction-based model we presented in Section 3.4 is intended to serve as a general tool for a rigorous study of incentives in BitTorrent-like settings where experiments do not apply. Here, we focus on PropShare's performance both in live swarms and on PlanetLab.

3.7.1 How do we expect PropShare to perform?

In our evaluation, we compare our PropShare client to BitTorrent and BitTyrant. While BitTyrant was built specifically to game BitTorrent, the goal of our PropShare client is to provide robust incentives even against future clients. Our implementation therefore takes *no BitTorrent- or BitTyrant-specific actions.* As such, we expect that BitTyrant will perform better in a swarm consisting predominately of BitTorrent peers.

In the publicly available BitTyrant implementation, BitTyrant peers ramp bandwidth allocations to one another using a k-TFT-like scheme. This strategy is not shown to be strategyproof [98]. We considered modifying the BitTyrant client to remove this potentially game-able strategy, or to announce our PropShare client as a Tyrant to gain from others' sharing. However, in practice, one would not be able to do this, and there may be many peers employing many different strategies. We thus opted to let our PropShare client "fly blind" against BitTyrant peers.

3.7.2 Experiments on live swarms

We begin our evaluation by comparing the performance of BitTorrent, BitTyrant, and PropShare on live swarms. In these experiments, we chose torrents with a large leecher-to-seeder ratio to test the various clients' ability to trade with others. We started the three clients simultaneously from three separate machines, each on the University of Maryland network. We limited each client's upload bandwidth to 100 kilobytes per second.

Figure 3.3 validates our hypotheses regarding PropShare's performance relative to other clients. We plot results from 21 live swarms, sorted by PropShare completion time.

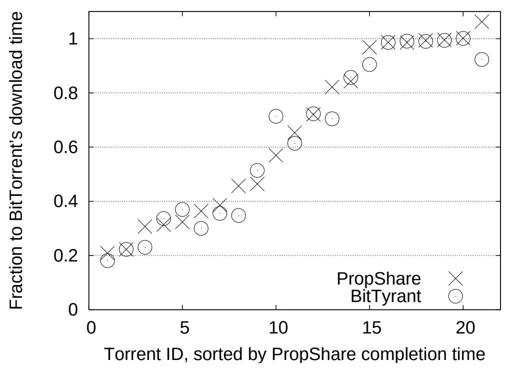


Figure 3.3: Runs on live swarms

BitTyrant, being tailored specifically to exploiting BitTorrent peers, frequently performs the best of the three clients. Although our PropShare client is a straightforward realization of the proportional share mechanism from Section 3.5 that employs no BitTorrent-specific mechanisms, it performs comparably well to BitTyrant, and in all but one download, much better than BitTorrent. We believe PropShare experiences good performance in live swarms because it shares some similarities with BitTyrant; PropShare allocates more bandwidth to peers with greater return on investment. The main difference between PropShare and BitTyrant is that PropShare will reward all peers, even those with poor return on investment, with bandwidth. The results in Figure 3.3 indicate that this additional expenditure is not detrimental, and in some cases improves performance.

We conclude that PropShare is incrementally deployable. Users could begin using and benefiting from a PropShare client today, and we intend to make our client available as open source. PropShare does not require a full deployment to benefit, or a change in the protocol to improve performance. This is a result of PropShare's resilience to strategic manipulation; PropShare does not require mechanisms such as voting or long-term agreements in order to benefit, and is even resilient to colluding peers. It is thus natural that a single PropShare peer would perform well in a swarm of non-PropShare (but rational) peers.

3.7.3 Competitive experiments

It is reasonable to assume that the vast majority of the peers contacted in our live swarms experiments ran unmodified BitTorrent clients. We now study how PropShare performs when either it or BitTyrant hold the majority.

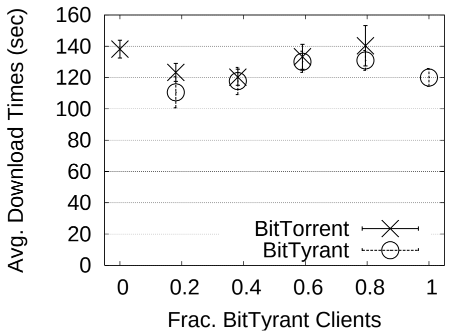


Figure 3.4: BitTyrant vs. BitTorrent

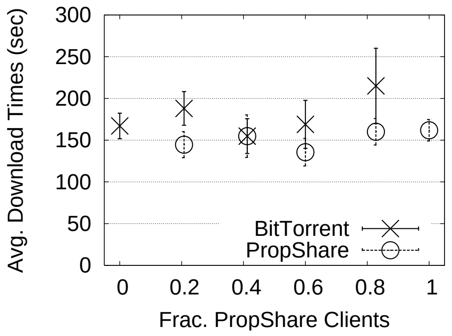


Figure 3.5: PropShare vs. BitTorrent

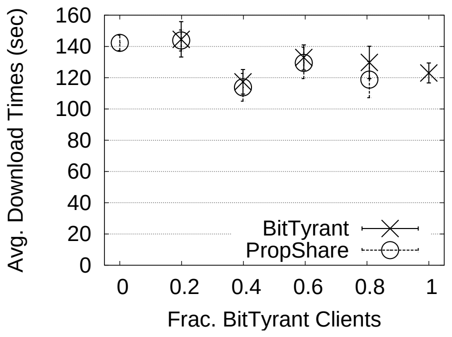


Figure 3.6: PropShare vs. BitTyrant

Experimental setup

We ran competitive experiments on roughly 110 PlanetLab nodes. In these experiments, we pitted two clients against one another by keeping the number of peers fixed across all experiments, but varying the relative fraction of client types. We adopted Piatek et al.'s BitTyrant experimental setup: three seeders per file, with a combined upload bandwidth of 128KBps, a locally run tracker, and peers downloading 5MB files. We used the bandwidth distribution presented by Piatek et al. [98], and varied each peer's bandwidth cap in each run. Each peer left the swarm as soon as it was done downloading the file. We reduced dependencies across runs by not using the same file between two separate runs; many trackers impose a limit on how often peers can request new peers, which could cause some of the slower peers from an earlier experiment to be forced into long waits at the beginning of the next. Each pair of points in the figures that follow represents the average over at least 3 runs, and error bars denote 95% confidence intervals.

BitTyrant vs. BitTorrent

The original BitTyrant study [98] measured average download times on swarms consisting of all BitTyrant or all BitTorrent peers, or when one BitTyrant peer attempted to game the rest of the system. We augment that study by considering intermediate ratios of clients. Figure 3.4 shows that there are interesting dynamics between the two extreme points. There is a clear trend toward an increase in BitTyrant performance as there are fewer of them.

The trend of Figure 3.4 is clear within the context of the auction model of Section 3.4. Consider a strategic bidder b; if there are few other strategic bidders in the system, then b

can more easily obtain the last (but still paying) place in the auction. As bidders begin to learn that they do not need to place such large bids at their respective auctions, they are free to bid at more peers. This in turn raises the contention at each peer for the last-place slot.

Figure 3.4 also reveals a tragedy of the commons. The users with the best download times in this figure are the BitTyrant peers who were in the *minority*; the early BitTyrant adopters, so to speak, were rewarded well, but as more users switch to BitTyrant, overall performance degrades for all users. With a majority of BitTyrant peers, the predominate interactions system-wide are between BitTyrant peers; their strategy of ramping up bandwidth allocations to one another compensates for this tragedy of the commons. The end result is a system with overall better performance than BitTorrent—BitTyrant incurs lower average download times as a whole—but one that is not known to offer strategyproofness guarantees [98].

PropShare vs. BitTorrent

When run against BitTorrent, PropShare clients exhibit different behavior than do BitTyrant clients. Rather than experience the best performance when a minority, PropShare clients maintain low download times even as the number of PropShare clients increases. This appears to come at the BitTorrent peers' expense, as PropShare induces more significant increase in download times for non-PropShare peers. This implies, along with our live swarm results, that PropShare not only can be used today to achieve better download times, but that its performance will not degrade as more users switch to it.

PropShare vs. BitTyrant

Next, we run PropShare clients against BitTyrant clients. In each *mix* of clients in Figure 3.6, PropShare clients out-perform BitTyrant peers. This by no means proves that PropShare is not game-able, but lends further credence to PropShare's ability to be used today, among clients of varying strategies. A swarm consisting of all PropShare clients performs worse on average than a swarm of all BitTyrant clients. That PropShare would even be competitive while BitTyrant peers ramp allocations to one another is encouraging; it shows that *PropShare's cost of robust incentives is low*.

All-versus-one

From Figure 3.6, we see a general decrease in download times as the fraction of BitTyrant peers increases. We now study the cause of this. BitTyrant effectively runs two parallel strategies: one against other BitTyrants, and one against all other strategies. The inter-BitTyrant strategy has not been shown to be strategyproof [98]. We do not consider here how one might game such a strategy. Instead, we focus on testing the following hypothesis: that BitTyrant's improved download times are strictly a result of the (potentially game-able) actions they take against one another, and not indicative of BitTyrant gaming PropShare. To test this hypothesis, we run all-versus-one experiments, where $N-1$ peers run one strategy (PropShare or BitTyrant) the remaining peer runs alternates between the two strategies in subsequent runs. We present our results in Table 3.1. These results are

	All other peers	
One peer	BitTyrant	PropShare
BitTyrant	86.9 (6.40)	109 (15.6)
PropShare	70.5 (4.61)	107 (14.3)

Table 3.1: Average download times (seconds, with standard deviation) for a single client choosing between two strategies.

consistent with Figure 3.6, in that a majority of BitTyrant peers yields faster system-wide download times on average. Table 3.1 shows that a BitTyrant peer does not on average successfully game PropShare clients, and that in fact a PropShare client is the preferred strategy against a swarm of all (other) BitTyrants. We conclude that the decrease in download times observed with more BitTyrant peers is not a result of gaming PropShare, but the speedup from the potentially game-able inter-BitTyrant strategy.

3.8 Bootstrapping File Sharing

Viewing BitTorrent as an auction further motivates the need for seeding: how can a node bid in an auction without any form of "currency" (blocks)? A node can exploit the optimistic unchoking by going to each peer; this is the so-called large view exploit [112] on which BitThief [79] is based. Optimistic unchoking attempts to trade off robust incentives for a greater social good by decreasing the average download time. However, the lack of incentives opens it to attack, which decreases in social good [79]. One might think that the seed nodes are enough to bootstrap the system, but in fact this is not strictly true, and is what motivates the need for optimistic unchoking at all peers. This is a somewhat orthogonal problem to the auction mechanism, so we propose a mechanism that solves this problem in the more general context of *bootstrapping* piece exchange.

Suppose two nodes A and B have been trading blocks for multiple rounds, and a new node n has just requested that A "bootstrap" him by giving him blocks. A may be suspicious for two reasons: (1) n may simply take the block and leave, never giving A anything in return, or (2) n may be a Sybil [36] of B (or some other node), and B is simply trying to extort more bandwidth out of A.

A may settle his suspicions by agreeing to send block b encrypted with a symmetric key K_A, $[b]_{K_A}$, to n. A must ensure that the bandwidth it gives to n is not "wasted," i.e., that (1) A receives bandwidth in return, and (2) A's established trading with B is not adversely affected. Thus, A does not reveal K_A unless n forwards the block to B.

Algorithm 4 Bootstrapping piece exchange

```
1. $n$ requests a block (of $A$'s choosing) from $A$.
2. $A$ informs $B$ that it will be using $n$ as a "proxy." $A$ encrypts $b$ with $K_A$ and sends a
   hash of $[b]_{K_A}$ to $B$.
3. $A$ informs $n$ to forward packets to $B$.
4. $A$ sends $[b]_{K_A}$ to $n$, which $n$ forwards to $B$. Concurrently, $B$ sends an encrypted
   block $[b']_{K_B}$ to $A$.
5. $B$ verifies that the hash of the block $n$ sent matches what $A$ sent. If correct, $B$
   informs $A$ that it received the correct block from $n$, i.e., that $n$ performed the task
   as instructed.
6. If $B$ sends $A$ the proper hash, then $A$ reveals $K_A$ to $n$ and $B$, and $B$ reveals $K_B$ to
   $A$.
```

The newcomer, n, pays his dues in two ways: (1) by forwarding (to B) as much as it receives (from A), and (2) by placing its trust in A to truthfully reveal K_A. Note that n need not place any trust in B; if B reports that the data it received is incorrect, then A will not reveal K_A to either B or n, so it is in B's best interest to truthfully report in step 6. Friedman and Resnick [44] observe that there is a trade-off between a system's barrier of entry (i.e., how easy it is to bootstrap) and the long-term participants' protection against free-riders with cheap pseudonyms. Algorithm 4 addresses this trade-off reasonably: n's barrier of entry is low (placing trust in A), and n cannot free-ride.

One could envision using weak keys, so that even if A does not reveal K_A, n can find K_A in a reasonable amount of time (though more time than it would take for A to have simply sent K_A).

3.9 Conclusion

We have formalized some of the debate on incentives in BitTorrent by focusing on two of its main components: the unchoking algorithm and the piece revelation strategy. Within a game-theoretic model, we have shown that BitTorrent does not use tit-for-tat, and we have proposed an auction-based model that we find to be more accurate. When viewed as an auction, it becomes clear that BitTorrent's current unchoking algorithm does not yield the fairness and robustness guarantees desired from such a system. With the goal of "the more you give the more you get," we have investigated the use of a proportional share mechanism as a replacement for BitTorrent's unchoker, and shown that it achieves fairness and robustness without any wire-line modifications to the BitTorrent protocol.

Additionally, we have proposed a new bootstrapping mechanism with the goal of replacing BitTorrent's optimistic unchoking in favor of an approach that encourages peers to contribute to the system as soon as they join.

There remain many interesting areas of future work. The bootstrapping mechanism we propose is intended to be used by new peers, but is there incentive for an existing peer to use it, for instance, to obtain rare blocks from peers who are otherwise uninterested? Our results in this chapter focus on incentives *within* a swarm, and not *between* swarms. The problem of *seeder promotion*—providing incentives to peers who have completed downloading the file to seed the file—is very important.

We believe that the mechanisms we have presented can complement future solutions to seeder promotion. For example, suppose peers who have finished downloading may stay in the swarm, not to obtain pieces of the file, but to obtain a virtual currency that they may spend for faster download times for files they download in the future. This is a natural approach to solving the seeder promotion problem, and has been investigated in various forms [113, 97]. Our mechanisms can complement such a scheme; PropShare could accommodate a market consisting of both file pieces and currency so long as there is a way to value the virtual currency in terms of file pieces. Further, our bootstrapping mechanism is useful in such a system in that it allows a peer to enter a swarm consisting of peers who are unwilling to accept any virtual currency.

Chapter 4

Motivating Repeated Interactions

The system we analyze in this chapter, *PeerWise*, also exhibits a bilateral exchange game, but differs fundamentally from that of the previous chapter. PeerWise is an Internet routing overlay that reduces end-to-end latencies by allowing peers to forward through a relay instead of connecting directly to their destinations. Peers in PeerWise exchange bandwidth by forwarding traffic for each other. Whereas BitTorrent peers are always interested in trading—the sooner they get more pieces of the file, the sooner they finish downloading—PeerWise peers may have transient periods wherein they have no traffic to forward. PeerWise would offer virtually no utility if a user only forwarded for others precisely when he had his own traffic to forward, as well. This motivates new mechanisms to maintain a *peering agreement* between two peers that provides incentives to each peer to contribute resources even when they do not immediately wish for something in return. We propose two mechanisms to define and maintain peering agreements: First, we use Service Level Agreements (SLAs) to expressively negotiate peers' demands and the recourses they will take when SLAs are violated. Second, we propose a mechanism to address SLA violations that differs from the standard notion of punishment via service degradation. Our simulation results demonstrate that our mechanism causes peers to avoid those who violate SLAs, and to favor long-lived peerings. Last, we discuss potential, emergent behaviors in a selfish routing overlay. [1]

4.1 Overlay routing

Routing overlays [6, 85, 50, 80] allow users to forward traffic through other users to influence the path their packets take through the Internet. With such influence, users can achieve greater resilience to network outages and decrease end-to-end latencies.

As more users participate in a routing overlay, more paths become available, and hence the utility of the overlay increases. We therefore propose as a goal of routing overlays to motivate participation: *encourage users to join the routing overlay and actively forward for others for as long as possible*. Until recently, routing overlays have been relegated to small deployments controlled by a single administrator; RON [6] for instance creates a fully connected mesh in which all nodes measure latencies to all other nodes, limiting a feasible deployment to only a few dozen nodes. PeerWise [80] makes scalability possible in these systems by using network coordinates [35] to reduce the number of peers users must contact to find shorter-latency paths to their destinations.

The feasibility of a large-scale, decentralized routing overlay raises problems of ensuring cooperative participation in the presence of selfish users. We expect users to selfishly attempt to maximize the *benefit*—latency reduction and bandwidth—they receive from the system while minimizing their *cost*—the amount of traffic they forward for others.

[1]Portions of this chapter originally appeared in *Motivating Participation in Internet Routing Overlays*, NetEcon 2008, August 22, 2008, Seattle, Washington, USA.

Selfish users complicate participation because they compete with one another for latency-reducing peers, and will not forward for one another if they do not receive something in return. As a system scales to a large number of users, maintaining global state becomes infeasible. Corbo et al. use local decisions to form economically-driven peerings to model AS topology [32]; we apply a similar philosophy in routing overlays to model user interactions and provide economic incentives for users to join.

The goals of this work are to motivate users to *establish* and *maintain* mutually beneficial peerings, and to do so in a completely decentralized, scalable manner. Our first contribution in this work is in demonstrating that *routing overlays present unique challenges in providing incentives for selfish users to participate* (§4.2). For instance, as compared to a system like BitTorrent [30], time plays a significant role in user demands; BitTorrent peers want as much as possible *now*, while users of a routing overlay may generate traffic at varying times throughout the day, and with varying demand.

In attempting to solve the *selfish routing overlay problem*, we focus on building incentives in PeerWise. Our solution to motivating participation is based on PeerWise's notion of *peering agreements*. A peering agreement in PeerWise is made between two participants, p and q, wherein p agrees to act as a latency-reducing relay for q, and vice versa.

A selfish PeerWise user is faced with two interdependent decisions: (1) choosing with whom to form peering agreements, and (2) choosing how much to allocate to the peers it chooses. We propose two mechanisms to address these. The first mechanism (§4.4) borrows from the notion of Service Level Agreements (SLAs) that autonomous systems in the Internet use to establish business agreements with one another. SLAs in PeerWise allow peers to expressively negotiate their demands and the recourses they take when the SLAs are violated.

SLA violations require recourse. The second mechanism we present (§4.5) addresses the problem of avoiding negotiations with peers who continually violate their SLAs. An intuitive solution to this is for peers to punish one another by offering reduced quality of service, but this runs the risk of incurring unnecessary performance degradation or, if engineered poorly, system-wide collapse. Our mechanism takes a subtly different approach than punishment. We propose that peers instead use SLA violations as a negative reflection on the perceived benefit that violators offer. We present simulation results that show that our mechanism limits how often victims return to an SLA violator, and achieves long-lived peerings with peers that satisfy SLAs. We conclude with a discussion of potential emergent behaviors in a selfish routing overlay, and open problems (§4.6).

4.2 The Selfish Routing Overlay Problem

The problem of providing incentives in routing overlays represents a new point in the design space of incentives for selfish participants. Broadly, we aim to promote participation and resource sharing among selfish peers. In this section, we formulate the problem of providing incentives in routing overlays in the context of how it differs from well-studied problems.

4.2.1 Participants may opt out

Latency-reducing routing overlays are an optimization on standard Internet routing. Participants in routing overlays may always leave the system and "fall back" on the higher-latency paths that BGP returns. Thus, when a participant in a routing overlay perceives their costs from remaining in the system to exceed their benefits, they may *opt out* of the system. In this sense, fairness comes “for free”: routing overlay participants can always achieve correctness without their costs exceeding their benefit. This is unlike most P2P systems such as BitTorrent [30], where for many files the only way to download the file is via BitTorrent.

We view the fundamental goal of providing incentives in routing overlays to be to *motivate peers to stay in the system as long as possible*. This equates to allowing peers to easily find benefit from the system (if it exists for them) and to maintain that benefit over long periods of time.

4.2.2 Not all peers benefit from one another

It is not the case that a given peer can benefit from every other peer. Though the triangle inequality does not always hold in the Internet, it often does [35, 123, 126]. Conversely, in many peer-to-peer systems, all peers stand to benefit from all others. In BitTorrent, for example, a peer can potentially benefit from every other peer, as long as they do not have all of the same pieces of the file. Peers in a routing overlay, however, may have a small (or perhaps even empty) set of other peers who can offer them lower latency paths to a given destination. A goal is thus to *match up the peers in a way that benefits as many as possible.*

4.2.3 Users have varying demand

Users’ traffic demands—in terms of both volume and latency—vary across the applications they use, the destinations they contact, the time of the day, and so on. For some applications, like ssh, a small amount of bandwidth with a large reduction in latency is sufficient, while in others, like VoIP, a reduction in latency is allowed only as long as there is a moderate amount of bandwidth. This is distinct from systems like BitTorrent, wherein everyone's *demand* is the same: download the file as quickly as possible. Another goal of a routing overlay is thus to allow peers to communicate their demands with one another. If two peers are willing to relay for one another, then they should ideally be able to negotiate a mutually advantageous agreement. Put simply: *allow willing peers to peer*. We propose to apply the notion of Service Level Agreements (SLAs) to routing overlays to achieve this expressiveness (§4.4).

4.2.4 Long-lived agreements are preferred

Users vary not only the *amount* of resources they demand, but also the *time* at which they demand it. Often in routing overlays, peers may be expected to generate traffic at different times. Conversely, BitTorrent peers benefit from one another *simultaneously*. An important goal in selfish routing overlays is therefore to *motivate peers to maintain long-lived agreements with one another to forward, and to not require simultaneous interest.* Long-lived agreements are natural in routing overlays, because if p benefits from having r as a relay at one point in time, p is likely to continue benefiting from r in the future [80].

This differs from applications where there is an end-game. In BitTorrent, for example, there is a finite-sized file, so peers will not benefit from one another indefinitely. Ad hoc wireless networks might also be considered to have an end-game in that the comprising nodes are generally energy-constrained, or are mobile and therefore unlikely to interact with others for long periods of time.

We propose a mechanism to ensure long-lived peerings. Interestingly, our mechanism does not require any degraded service, and is resilient to varying network conditions (§4.5).

4.2.5 No modifications to destinations

Routing overlays are intended to be used to contact any destinations in the Internet. Requiring modifications to all destinations in the Internet would render deployment infeasible. We do not require any modifications to the users' destinations.

This affects how selfish peers in a routing overlay interact with one another. Suppose peer p is relaying his traffic through peer r to reach destination p_d. Tunneling packets through r entails r rewriting the source address of the packet with his own IP address. From p_d's perspective, r's host is connecting to p_d, not p's. If r were to stop relaying for p while p had an open connection to p_d, then p would have no means of recovering this connection. Modifications to the destination could alleviate this, but again, at the cost of ease of deployment. This further motivates the need for long-lived peerings; by providing incentive for r to maintain its peering agreement with p, p can gains (more) assurance that its connections through r will remain.

4.2.6 Peerings are not roommates

Establishing peering agreements in PeerWise is similar to the classic stable roommates problem [55]. The stable roommates problem takes as input a set of agents, and outputs a *matching*: a set of pairs of agents ("roommates"). A matching is *stable* if there are no two agents that would both prefer to be matched with one another over the agents with whom they are respectively matched. A matching M is *maximal* if there is no other matching M' such that $|M'| > |M|$.

Stable and maximal matching problems appear to be a natural fit for PeerWise. Stability relates to long-lived peering agreements, and maximal matching equates to maximizing participation in PeerWise. However, the problem of choosing relays in PeerWise is vastly more complex than the stable roommates problem. Instead of being matched with one other peer, participants in PeerWise make many peering agreements; preferences in PeerWise may therefore be exponential in size. Varying network conditions and user demands add further complexity. It is therefore unlikely that a generalization of the stable roommates problem to PeerWise-like settings will allow a polynomial time algorithm. A more rigorous understanding of the connection between the two problems is an area of future work, and may help to reveal the theoretical limits of the mechanisms we present in this chapter.

4.3 Selfishness in PeerWise

We focus on providing incentives in the PeerWise [80] routing overlay. In this section, we give a brief overview of PeerWise, and define the utility of a PeerWise participant.

4.3.1 PeerWise Overview

PeerWise [80] is a latency-reducing routing overlay that provides scalability and fairness. The key idea of PeerWise is that two nodes can cooperate to obtain faster end-to-end paths without either being compelled to offer more service than they receive. PeerWise comprises two mechanisms: (1) it uses network coordinate systems, such as Vivaldi [35], to aid in discovering triangle inequality violations in the Internet; and (2) nodes negotiate and establish *peering agreements* to each other based strictly on mutual advantage.

Figure 4.1 shows a two-node PeerWise overlay. Node A discovers a faster path to D

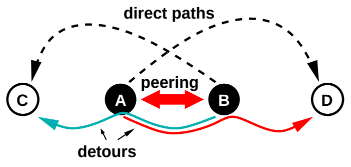


Figure 4.1: Obtaining faster paths with PeerWise: A discovers a detour to C through B; B also finds that it can reach D faster if it traverses A; A and B create a mutually advantageous peering which they both use to get more quickly to their destinations.

via B. However, B will not help A unless A provides a detour in exchange. Since there is a shorter path from B to C going through A, A and B can help each other communicate faster with their intended destinations. Therefore they establish a pairwise peering.

4.3.2 Self-interest in PeerWise

Selfish peers will strategically attempt to form peering agreements that maximize their individual utility, regardless of how it might impact the rest of the system. Here, we formalize the goals and utilities of selfish peers. We assume that peers only seek out mutually advantageous latency reduction, as opposed to one party paying the other to relay. We leave the use of transferable utility (money) as an alternative form of payment to future work.

A PeerWise peer's utility

A peer's utility is defined by the benefits and costs that it perceives. The *benefit* that a PeerWise peer p obtains from using r as a relay to reach destination d is an increasing function of its latency reduction:

$$L_p(r) \stackrel{\text{def}}{=} 1 - \frac{\ell(p \to r \to d)}{\ell(p \to d)}$$

where $\ell(x)$ denotes the latency for path x. A peer's benefit is also a function of the amount of traffic it is allowed to forward. For example, a 99% latency reduction for one packet is likely not preferable to a 20% reduction for all of a peer's traffic. For ease of exposition, we assume that p has a minimum amount of required bandwidth, and if r does not provide this, then p receives no benefit from r.

The *cost* that peer p incurs from relaying for r is an increasing function in $b_p(r)$, the amount of bandwidth p forwards for r, and is normalized by the amount of bandwidth p is willing to provide to PeerWise, B_p:

$$C_p(r) \overset{\text{def}}{=} \frac{b_p(r)}{B_p}$$

Such cost can come in many forms: the impact to their own applications' performance, actual money-per-byte costs, the need to leave their computer turned on over night, and so on.

We propose the following general form utility function for peer p:

$$u_p(r) \overset{\text{def}}{=} \alpha_p L_p(r) - \beta_p C_p(r) \tag{4.1}$$

Constants $\alpha_p \geq 0$ and β_p can be chosen to suit a wide range of users:

- $\alpha_p = 0, \beta_p < 0$: p is altruistic.
- $\alpha_p > \beta_p > 0$: p is willing to pay more for his latency savings than he receives.
- $\beta_p > \alpha_p > 0$: p expects to get more than he gives back to the system.
- $\alpha_p > 0, \beta_p = 0$: p does not incur costs on his uplink.

As discussed in §4.2.1, a peer will not remain in the system if its total costs outweigh its total benefits. We now see that p will leave the system if its sum utility is less than zero. Note that if p is altruistic then $\alpha_p = 0$ and $\beta_p < 0$, hence u_p is never negative.

Selfishly selecting peering agreements

Each PeerWise peer p must decide when to commit to and when to dissolve a peering agreement. *This is the fundamental problem a selfish peer must solve in PeerWise.*

More formally, a selfish peer p's goal is to choose a set $\mathcal{P}$ of peers with whom to make peering agreements. Suppose that p that is willing to contribute a total amount of bandwidth $B_p \cdot T$ to other PeerWise peers over any period of time T. To limit the *burstiness* of p's contribution, p may also wish to ensure that it never has to provide more than S_p bandwidth at any point in time. This protects p from having to provide all of its $B_p \cdot T$ bandwidth in a very small period of time, which could adversely affect p's performance. We formalize this with the following natural optimization problem:

$$\begin{aligned}
\text{maximize} \quad & \sum_{q \in \mathcal{P}} u_p(q) && && (4.2)\\
\text{s.t.} \quad & \sum_{q \in \mathcal{P}} C_p(q,t) \le S_p, && \forall t && (4.3)\\
& \sum_{q \in \mathcal{P}} \sum_{t=t_0}^{T} C_p(q,t) \le B_p \cdot T, && \forall t_0, T && (4.4)
\end{aligned}$$

This formulation makes clear an important difference between selfish behavior in PeerWise and selfish behavior in a system like BitTorrent: *time*. Peers' demands in BitTorrent are constant: send as much as possible *now*. However, a peer's demands in PeerWise depend on when that user wishes to access its destinations, and could be nil when the user is asleep, for instance. We investigate this further in the context of over-committing one's resources (§4.6.1). In §4.4 and §4.5, we present mechanisms to assist PeerWise peers in obtaining a beneficial a set of peering agreements, while motivating them to uphold their responsibilities in their agreements.

4.4 Service Level Agreements

In PeerWise, nodes negotiate and establish peering agreements to each other based strictly on mutual advantage. Lume, et al. [80] we showed how potentially good peers are found—by exploiting the inability of network coordinates to work with triangle inequality violations—and how the peering agreements are negotiated—nodes that provide each other latency reduction to at least one destination form a peering. Next we focus on mechanisms that ensure the validity and stability of such agreements over long periods of time.

A service level agreement (SLA) is a formal contract between two peers that establishes all aspects of the service that each provides to the other. Nodes in routing overlays are inherently selfish and SLAs are an efficient method to curtail the effects of the selfishness. SLAs ensure that each node receives the expected level of quality-of-service even when the traffic demand in the network varies and the mutual advantage offered by a peering is time dependent. We describe the design of the PeerWise SLAs next.

4.4.1 SLA design space

We identify three performance metrics to be used as the basis for an SLA between two PeerWise users:

- **latency reduction**: the minimum latency reduction that a peer offers to the other to a specific destination

- **bandwidth**: the average bandwidth at which one peer forwards packets for the other to a specific destination
- **burstiness**: the maximum bandwidth at which one peer forwards packets for the other to a specific destination

Each SLA is associated with a re-evaluation timeout which triggers a verification of the SLA against the traffic exchange since the previous timeout. We assume that each PeerWise user has the means to monitor the traffic over all its peerings. Because IP is best effort and cannot provide measurable service, we do not require the bounds on the performance metrics defined in the SLA to be strictly enforced; it is enough if most of the packets sent over the peering satisfy the bounds specified by the SLA. Furthermore, we enforce frequent re-evaluations (on the order of minutes or hours rather than days or weeks). In doing so, we seek to protect the users against long periods when although the SLA is not violated, the peering is unusable.

Next we present a simple example of SLA between two PeerWise users. In Figure 4.1, nodes A and B discover mutual advantage and decide to form a peering. The SLA of the peering may state that A and B must offer each other latency reductions of at least 10ms for 95% of the packets that each send to C and D. The re-evaluation timeout is set 60 seconds. Furthermore, the bandwidth that each peer uses to forward each other's packets must not exceed 20kB/s averaged over the 60 seconds, with its maximum value always below 100kB/s.

Our SLA design draws from the agreements between autonomous systems in the Internet. Similarly to AS SLAs, PeerWise SLAs are intended to be maintained over long periods of time. In this way, long term reputation can motivate cooperation. However, AS SLAs are defined over much larger time scales. We require SLAs to be verified more often because traffic fluctuations may have bigger effects on a single link that connects two users than on a collection of links that interconnects two autonomous systems.

4.4.2 Dynamic SLAs

Because nodes generally do not know *a priori* their traffic demands and because the service performance metrics are chosen based on at most a few measurements, we do not require fixed SLAs between two PeerWise nodes. Instead, two nodes start with a temporary SLA, which specifies latency reductions as advertised during the negotiation and bandwidth metrics according to each node's expected demand. This is different from AS level SLAs, which are based on previous study of the network utilization and on performance metrics averaged over long periods of time [82].

The bandwidth and burstiness parameters of the SLA can be automatically changed after every re-evaluation to align with the real traffic exchanged between the peers. We allow both positive and negative parameter scaling. If two peers respect the terms of the SLA and if they have enough spare bandwidth, they may offer to relay each other's traffic at faster and faster rates. On the other hand, if a peer does not offer as much latency reduction as promised, then the available bandwidth through its peer will be reduced. In the next section we present mechanisms to encourage long-lived peerings even when users are selfish.

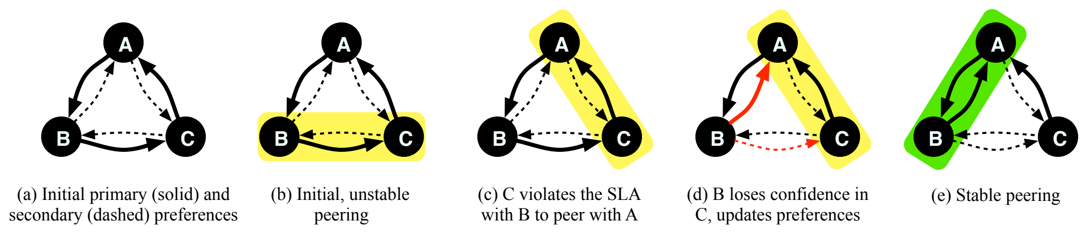

A
B
C
(a) Initial primary (solid) and secondary (dashed) preferences
(b) Initial, unstable peering
(c) C violates the SLA with B to peer with A
(d) B loses confidence in C, updates preferences
(e) Stable peering

Figure 4.2: Cyclic preferences lead to unstable peerings, but participants prefer long-lived peerings. Algorithm 5 addresses this by incorporating into a peer B's preference the confidence B has in its neighbors to maintain a long-lived peering.

4.5 Mechanisms for Long-Lived Peerings

PeerWise participants' choice of relays is driven by their selfish interest in obtaining their most preferred relays. Each peer p has a partially ordered preference, $\succ_p$, of relays. If, for two peers a and b, $a \succ_p b$, then p will prefer to use a as a relay regardless of the impact on b.

Given the anarchistic nature of this process, it is not surprising that it can yield poor outcomes. For example, consider three peers, a, b, and c, where $b \succ_a c$, $c \succ_b a$, and $a \succ_c b$. These cyclic preferences lead to oscillating peering agreements: if a obtains its preferred relay b, then that leaves c available. Since b prefers c, b will selfishly break its peering agreement with a in favor for c. This in turn leaves a available, leading c to break its peering agreement with b, and the process repeats indefinitely. These oscillating peering agreements resemble persistent oscillations which could occur in BGP [118]. Next we present a simple mechanism that limits the effect of selfishness on the stability of peerings.

4.5.1 Long-term cooperation with confidence

A fundamental difference between route changes in BGP and PeerWise is that a route change in PeerWise will generally result in a lost connection (§4.2.5). As such, PeerWise participants desire long-lived peerings, and should prefer relays who respect their SLAs for long periods of time. Though a peer i can measure the reduction in latency that a neighbor j provides, i cannot know if or when j will violate their SLA. Instead, i maintains some measure of *confidence* that j will relay i's packets. i's confidence in j can change over time, based at least in part on the history of interactions between i and j.

We propose that, in determining neighbor preferences, a peer *explicitly* incorporates its confidence that a given neighbor will maintain an SLA. Specifically, we propose that each peer i maintain a *confidence value*, c_{ij} for each peer j with whom it has interacted. A peer's confidence in another increases or decreases based on the interactions those two peers have had. For instance, when peer j violates or prematurely dissolves a peering agreement with i, i lowers his confidence of j. Though c_{ij} could also incorporate *others'* interactions with j, we focus in this chapter only on confidence based on one's own interactions.

Confidence values more accurately capture a peer's expected utility from a neighbor because they incorporate not just the instantaneous utility, but the history of interactions between the peers. Peers' preferences over one another can then combine the potential benefit of a peering—based on latency reduction, bandwidth, and burstiness described in §4.4—and the expectation that the peer will actually provide it. The power of incorporating history is demonstrated in Figure 4.2, wherein B's loss of confidence in C eliminates the initial, oscillating peering agreements.

To experimentally study the use of confidence values, we now propose a specific algorithm, Algorithm 5, to capture peers' confidence in one another. Peer i runs the algorithm at every predetermined period of time T. c_{step} represents the default value for increasing

Algorithm 5 Handling SLA violations.

```
Periodically run at peer $i$:

  • when $i$ and $j$ have a peering agreement
      – if $j$ respects the SLA, $c_{ij} \leftarrow c_{ij} + c_{step}$
      – if $j$ violates the SLA or drops the peering, $c_{ij} \leftarrow c_{ij}/2$
  • when $i$ and $j$ do not have an agreement, but they previously had one
      – $c_{ij} \leftarrow c_{ij} + c_{step}/(s_{ij} + 1)$
```

confidence. s_{ij} represents the number of times j has dropped a peering with i.

In Algorithm 5, the confidence value for a node increases slowly (additively) as the SLA is respected, and decreases quickly (multiplicatively) whenever the node violates the SLA. In this way, it will be more difficult for j to re-establish the agreement with i. This mechanism acts as a disincentive for nodes to constantly drop and add peerings on a small time scale.

4.5.2 Experimental results

To study the stability of PeerWise peerings we evaluate our confidence mechanism with a simulator. The simulations use a fixed topology of 16 PlanetLab nodes—the PeerWise nodes—and 16 popular web servers—the destinations. The latencies between PlanetLab nodes and web servers were measured in January 2008.

Each PeerWise node attempts to connect to each destination. SLAs are established between nodes that offer each other at least 10ms or 10% latency reduction. In our experiments, bandwidth is not a constraint. A peer violates an SLA associated with an existing peering whenever a new peering that offers higher benefit to the same destination is discovered. We assume that the re-evaluation period and the performance metrics of the SLAs never change. To assign confidence values for each node, we use the algorithm de-

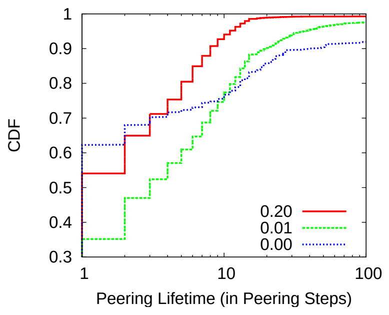


Figure 4.3: Peering lifetimes for different values of c_{step}.

scribed in §4.5.1. We restrict the confidence values to the interval [0,1] and we use three different values for c_{step}: 0, 0.01 and 0.2. After each re-evaluation period (peering step), we compute the total number of existing peerings, as well as the number of peerings that have been destroyed since the last re-evaluation. We present the results in Figures 4.3 and 4.4.

When c_{step} is 0, the confidence value of a peer that destroys a peering can never increase. Destroyed peerings are not recreated and the number of existing peerings after each re-evaluation eventually converges to a small value (around 50 in Figure 4.4). However, the peerings that *do* exist are long-lived, as shown in Figure 4.3. As c_{step} increases, so does the chance that a destroyed peering will be re-established. For $c_{step} = 0.2$, almost 150 exist at a given moment. Though there are many more peerings with greater values of c_{step}, the longevity of these peerings is diminished.

Thus, confidence values quantify the trade-off between how long-lived a peering is—longer with a smaller c_{step}—and how many peerings there are—more with a larger c_{step}.

4.6 Emergent Behaviors

In this section, we discuss two behaviors that may emerge in a system of selfish PeerWise participants.

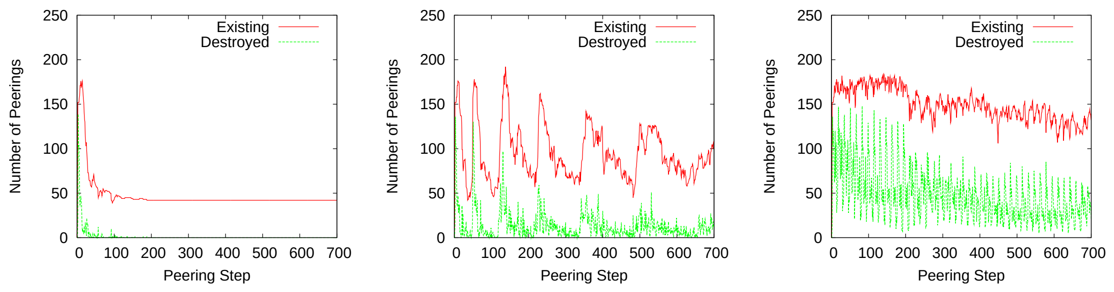


Figure 4.4: Example runs for different values of c_{step}: 0 (left), 0.01 (middle) and 0.2 (right).

4.6.1 Overselling one's resources

Peers may be inclined to promise more bandwidth across all of the peering agreements than they choose to actually provide over any period of time. This is common with today's ISPs, which advertise more bandwidth to home users than the ISP has provisioned. ISPs do this under the expectation that few enough users will wish to consume the advertised bandwidth at the same time. In other words, ISPs provision for average demands, and offer no assurance to customers when demand far exceeds the average.

In PeerWise, when a peer p makes a peering agreement with peer q, p commits to providing some amount $b_p(q)$ of bandwidth to q. Recall from Eq. (4.4) that p provisions B_p bandwidth for participation; p wishes to never provide more than $T \cdot B_p$ bandwidth over any period of time, T. This does not necessarily mean that, across all of p's peering agreements, $\sum_{q \in \mathcal{P}} b_p(q) \leq B_p$. Whenever this inequality does not hold, we say that p has *oversold* his resources.

The benefit of overselling is that it allows a peer to create more peering agreements, and therefore reach more destinations with lower latency. The risk of overselling is that, when more peers attempt to claim their promised bandwidth than p provisioned for, then p will be forced to violate the constraints in Eqs. (4.3) and (4.4). Violating these constraints may result in p violating some of its peering agreements, and as losing some of its relays (§4.5).

4.6.2 Reselling peering agreements

Selfish participants in PeerWise may attempt to create multi-hop paths to increase their profits. One such opportunity arises because PeerWise peers are unlikely to have global information about all relays. In a cooperative setting, if peer p knew that peer r could act as a latency-reducing relay for q—that is, $\ell(q \to r \to d_q) < \ell(q \to d_q)$—then p would inform q about r. In a selfish environment, p may trivially attempt to charge q for this information. An interesting scenario occurs when $\ell(q \to p \to r \to d_q) < \ell(q \to d_q) < \ell(q \to p \to d_q)$. In this case, p may attempt to *pretend* that he can offer q a one-relay, latency-reducing path, and secretly "re-relay" through r. Although q may not obtain as much benefit as if he were to know of and relay directly to r, q does obtain improved performance. This provides peer p an in-band incentive to assist peers in finding lower-latency paths.

Reselling need not be an act of subterfuge. Although most latency reduction is possible with one-relay paths, there are some source-destination pairs which obtain their optimal latency reduction with multi-hop paths [80]. Building multi-hop paths is a natural extension of peering agreements in PeerWise, as above.

An interesting area of future work is in studying the incentives properties of a path of these bilateral agreements. Would, for instance, a path built of bilateral peering agreements have comparable incentive properties to mechanisms that require more communication, such as VCG routing [119, 28, 49]?

4.7 Conclusion

We have proposed the selfish routing overlay problem, and demonstrated its difference from other, well-studied problems such as incentives in BitTorrent and the stable roommates problem. This problem is complex, in part due to the fact that participants' utilities can vary greatly. The crux of the selfish routing overlay problem is a local decision made by each peer: which peering agreements to maintain, and when, if ever, to dissolve the agreements it has.

Within the context of the PeerWise Internet routing overlay, we proposed two mechanisms to assist selfish routing overlay participants in choosing and negotiating their peering agreements. We believe these two mechanisms to be fundamental components toward solving the selfish routing overlay problem. An in-band mechanism as general as an SLA addresses the disparate goals of these peers. Reacting to the history of interaction between peers addresses persistent free-riders and unstable preferences.

Chapter 5

Systems-Compatible Bandwidth Auctions

Taken together, Chapters 3 and 4 demonstrate that systems that rely on users exchanging resources with one another can be built and deployed with incentives that ensure fair, truthful contributions. The crucial characteristic of BitTorrent and PeerWise that permits such incentives is that users in such systems interact for a relatively long period of time. In this chapter, we investigate peer interactions that may be extremely short-lived.

To drive our discussion, we focus on one of the nascent problems of incentive-compatible systems: provisioning bandwidth among a set of self-interested senders and receivers. Ideally, there would be sufficient bandwidth to meet all users' needs, but barring that, the bandwidth would be allocated to those who desire (or need) it the most. The crux of the problem is, put simply, to get users to *truthfully* state how much they want bandwidth. The wealth of literature investigating this problem has leveraged some of the powerful, truth-eliciting mechanisms from auction theory. The basic idea is for users to include a small bid in each packet, and for routers to hold auctions to decide the order in which to forward packets.

While such prior approaches have led to strong theoretical proofs of incentive-compatibility, it is difficult at best to infer their systems-compatibility. This is a problem of scope: prior work focuses on provisioning bandwidth across the entire Internet. Building and testing these systems would require sweeping modifications to routers, routing protocols, and ISP SLAs, to name a few. Unfortunately, this leaves us with the question: are truth-telling auction mechanisms compatible with systems, not just on a per-packet basis, but for realistic traffic profiles?

To answer this question, we introduce a new space within which to design and evaluate bandwidth-provisioning mechanisms: sharing access to cellular devices. It is becoming increasingly common for users to have mobile devices that have both a short-range 802.11 interface as well as a long-range cellular interface through which they access the Internet. Furthermore, many such users find themselves within short distance of one another, such as on a train or in a coffee shop. We introduce a new system, *Hoodnets*, that allows proximal users to share their Internet access with one another. Hoodnets aggregates users' bandwidth, effectively providing a single pooled resource of bandwidth roughly equal to the sum of the individual user's bandwidths. Unlike the systems in the two prior chapters, mobile users may enter and leave an instance of Hoodnets very quickly—even after sending as little as one packet. The system benefits from enabling as much cooperation in a short period of time as possible, while allowing peers to benefit from long-lived interactions, as well.

Studying incentive-compatible forwarding at a smaller scale allows us to implement and evaluate mechanisms proposed in prior literature. We find that, while the theoretical incentives properties are upheld, prior approaches do not efficiently support long-lived flows of traffic. We propose a new mechanism, *bandwidth leases*, that, conversely, supports long-lived flows but not short bursts of traffic. Finally, we demonstrate that per-packet auctions and bandwidth leases naturally compose, and together support a wide range of traffic patterns.

In addition to the incentives underlying Hoodnets, this chapter discusses the architecture necessary to support bandwidth sharing in mobile devices. We demonstrate that incentives, much like security, often cannot be "stuck on top" of a system. Rather, considerations must be taken throughout a system's design in order to provide incentives to its users.

We have implemented the hoodnet architecture, and we demonstrate with a thorough evaluation on real cellular links that it achieves nearly additive bandwidth aggregation while maintaining strong security and incentives properties.

5.1 Introduction

Auctions have emerged as a popular solution to scarce resource distribution between selfish participants. They have been applied in practice outside at a large scale for spectrum allocation [40], distribution of electricity [33, 31], or physical goods (eBay), and in principle at the packet or transaction level [125, 73, 81, 39]. Yet the practicality of per-packet or per-flow auction-based pricing remains dubious. In this chapter, we introduce *Hoodnets,* a system for channel bonding limited bandwidth wireless links, and show how to make auctions a practical component of the Hoodnets design.

Hoodnets permit machines with spare, typically 3G cellular wireless, uplink capacity to share that resource with nearby nodes over a faster, typically 802.11, backplane. Devices with such dual connectivity are increasingly plentiful. Prior work (FatVAP [59], MoB [21], MAR [103]) has shown the feasibility of combining cellular wireless links on a per-flow basis among cooperative parties: that is, nearby connected nodes can volunteer to forward a complete flow at a time. In contrast, hoodnets expects that nodes will require incentive to forward traffic to offset the battery power drain and the cost of preempting a node's own traffic. Hoodnets also expects that pinning a flow to one relay compromises performance (maximum throughput is limited by the node), reliability (should that 3G connection drop out or the node fail, progress is stalled), and resilience to price gouging (later packets in a flow become more valuable if that transfer cannot be resumed elsewhere).

To relax the assumptions of cooperation and per-flow channel bonding requires a significant redesign of the forwarding architecture. Foremost, the Internet-visible endpoint of the connection must be under the client's control, able to reassemble upstream flows and adaptively split downstream flows, provide feedback about the performance of relays, and take responsibility for what might otherwise be laundered attack traffic. To solve the inherent security, privacy, and accountability in Hoodnets, our insight is to forward all of a node's traffic through an Internet-side proxy, where relayed traffic passes through both the relay and the relay's Internet-side proxy. We motivate and describe this two-proxy routing approach in Section 5.3. Secondly, the performance deliverable by any peer varies, both in design because of contention and varying bids by peers, and in practice because cellular wireless networks experience varying bandwidth due to contention and signal quality. We describe our approach to multi-link scheduling, which relies on estimating delay through each path and using a small reorder buffer to hide ordering mistakes from TCP endpoints in Section 5.5.

These technical solutions represent the foundation of the hoodnet architecture. However, to mediate access to the pooled uplink resource requires the design of auction mechanisms that will provide both performance and fairness.

We apply two classes of auction: the per packet auction, capable of responding to bursty traffic, and the bandwidth lease auction, capable of providing sustained performance. Individually, each is insufficient to provide good performance to short and long flows. To establish leases fairly, we use a time-based spot market, which is commonly applied in the day-ahead electricity market [33]. We can use a spot market without centralization by exploiting the broadcast nature of the wireless medium to hold bidders to a single bid per epoch. We describe the design of the auctions in Section 5.4.

This chapter makes the following contributions:

- We integrate auctions into packet forwarding in a real system.
- We implement the hoodnets architecture using window-based flow control and two-proxy routing.
- We show the performance of per-packet channel bonding to increase single-flow throughput across cellular wireless links.
- We show the importance of merging short-term and long-term resource allocation with distinct auction mechanisms.

5.2 Background and Related Work

In this section, we describe prior work in channel bonding systems, in using auctions to price network resources, and in auction design in general where there are many sellers and many bidders.

5.2.1 Bonding selfish channels

At its core, Hoodnets is a *channel-bonding* system, which increases performance by using multiple poor-performance links at once. Channel bonding, or inverse multiplexing, naturally applies whenever there is a fast, local-area backplane and multiple, slow, wide-area connections. It is therefore unsurprising that it has been studied extensively in many domains. Load balancing across wired telephone lines [114] and Ethernet and ATM [26, 2] are some of the earliest examples, while somewhat more recent work has bound 802.11 [59, 10, 116, 117] and cellular [103, 21, 100] links.

Hoodnets is distinguished from the majority of prior approaches in that it is designed to allow mobile users to share their cellular links with other nearby users. Selfish behavior is reasonable to assume when sharing mobile links because forwarding for another can incur additional monetary cost, and can more quickly drain a mobile device's battery. MoB [21] suggests overcoming such participation concerns by allowing peers to compensate those who forward for them [21, 125]. However, MoB does not specify a mechanism by which peers can negotiate link access. Although standard auction theory would seem to fill this gap, we discuss later in this section the challenge in applying existing auction techniques to channel bonding.

In addition, we find that obtaining *efficient* channel bonding in a selfish environment requires a ground-up design of the system's forwarding infrastructure. Opportunistic forwarders make for unpredictable links that can harm performance. Because a participant is compensated when forwarding for others, he may have incentive to cease forwarding for one in favor of another who offers greater compensation. Thus, systems that pin a TCP flow to a particular neighbor, including FatVAP [59], MAR [103], MoB [21], and CUBS [117], are susceptible to dropped connections in selfish environments. Hoodnets instead exports a single, high-bandwidth link and forwards traffic on a per-packet, not per-flow, basis. While resilience to selfishness is an impetus, performance is a major added benefit; in Hoodnets, no TCP flow is bound by any individual link's capacity.

5.2.2 Auctioning off link access

Hoodnets uses auctions as a means of negotiating link access among peers. Auctions are a natural mechanism for allocating a network resource because they can be designed to encourage users to truthfully state how much of the resource they need [119, 28, 49]. MacKie-Mason and Varian [81] proposed using *per-packet auctions* as a means of pricing congestion in the Internet. At a high level, an end-user in a per-packet auction places a bid in each packet, and routers prioritize packets with greater bids. As a reaction to congestion pricing, Shenker et al. [108] note that per-hop auctions would permit bids to be rejected far from the source, would require bids to take each hop into account, would be mismatched when users' interest is in complete flows not packets, and would respond poorly to immediate demands.

Yang uses auctions for service differentiation; packets with bids may contend for a fast path, those without or that have exhausted their bids are sent along a slower path [125].

This avoids the sensitivity to bids that Shenker points out in congestion pricing.

These approaches that use auctions at each hop differ significantly from the local deployment of Hoodnets, in which the bidder and seller are within one 802.11 hop of each other, a bid need only win at one location, and there are potentially many neighbors (sellers), any of which could be chosen to forward, without significant complexity in disseminating information for source routing. Although Hoodnets does expect a default path through a node's own uplink, there is no provision for a "slow" path as in Yang's approach for ISPs: forwarding a packet through another node may fail if a steady stream of others is willing to bid more.

5.2.3 Bonding over auctions

Hoodnets combines the two areas of channel bonding and bandwidth auctions. As a result, users attempt to win auctions at *multiple* neighbors simultaneously. In a simultaneous auction [47], a user must decide not only how much to bid, but *where* to bid. This setting is similar to combinatorial auctions [34, 91] in that a bidder may seek a bundle of goods rather than a single good. Such auctions are notorious for being computationally expensive when truthful; in general, determining the winning allocations and payments of a combinatorial auction is NP-complete [104]. Because Hoodnets peers hold auctions either every ten seconds or on a per-packet basis, such complexity is untenable. To vastly reduce the computational and communication overhead, Hoodnets does not strive to make truth-telling a dominant strategy. Instead, Hoodnets operates in the *core* [94], upholding several reasonable fairness constraints, which we outline in Section 5.3.

In summary, though channel bonding systems and packet auctions are far from new, we believe that the increasing prevalence of dual-network devices motivates combining the two. In Hoodnets, auctions represent a practical means of allocating network resources between selfish network participants.

5.3 Hoodnets Design

Hoodnets' fundamental benefit to users is improved bandwidth and reliability through channel bonding without requiring extensive deployment; as long as other hoodnet participants are nearby, these improvements are possible. Hoodnets achieve this with its routing protocols, which we present in this section. The two architectural components of a hoodnet are shown in Figure 5.1: (1) a set of *hoodnet peers* that exchange service, and (2) a set of *hoodnet proxies*: machines on the Internet that multiplex the connections of the peers. Hoodnet peers send their packets across the cellular links of neighbors; these packets are recombined into one connection at a hoodnet proxy, which then splits return packets across multiple peers on the way back to the hoodnet. In the following sections, we present the incentives and security mechanisms that encourage adoption and ensure correctness even among selfish users.

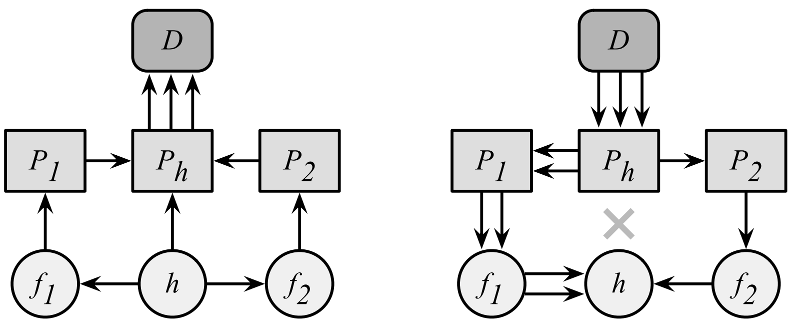
5.3.1 Architectural components

Hoodnets require no special infrastructure beyond a small amount of code on a peer machine, in contrast to systems like MAR [103] require a multi-interfaced router being deployed in hotspots like buses. The hoodnet software implements an overlay network to tunnel traffic in an application- and transport-independent manner.

Hoodnet peers

Each user is associated with a *hoodnet peer*: their mobile device through which they use networking applications or can simply forward traffic for others. Hoodnet peers serve one major role: to forward their and other users' traffic. Hoodnet peers communicate with one another via 802.11 and connect to the Internet via their cellular link. We assume all hoodnet peers have 802.11 interfaces, and some of them also have cellular connections. This setting exists *today*; many mobile users connect to the Internet via a bluetooth connection to their mobile phone, and the iPhone represents a trend toward devices with both 802.11 and cellular interfaces. Our design is not bound to these particular technologies; the necessary condition for bandwidth improvement is simply that the devices can communicate with one another with greater bandwidth than their individual links to the Internet. Reliability comes from a diversity in the users' cell providers; a Verizon user near a Sprint user can achieve connectivity whenever *either* of them is in their provider's coverage area.

We present the components of a hoodnet peer in Figure 5.2. The Hoodnets module obtains local application packets via a `tun` interface, and enqueues them on a queue we call the *antechamber*. Packets enter the antechamber before departing for another interface, either the peer's cellular link (`ppp0` in the figure), or to a neighbor (over the 802.11 link). Hoodnet peers have components for buying access to others' links and selling access to their own, which we describe in detail in Section 5.4. If a peer is selling access to its link, as is the peer on the right in Figure 5.2, it may maintain multiple queues concurrently. Peers may purchase bandwidth from their neighbors; a hoodnet seller uses weighted fair queuing to schedule outgoing packets, weighted by the amount the various neighbors have



(a) *h* uploads (b) *h* downloads while disconnected

Figure 5.1: Bidirectional traffic flow in a hoodnet. Traffic is multiplexed in both directions, offering greater upload- and download-bandwidth and reliability.

purchased.

Hoodnet proxies

A strawman protocol would be to split packets across hoodnet peers, have them forward to the destination, and require the destination to piece the packets together into a single coherent connection. This approach would require extensive deployment at destinations and compounds the problem of holding peers accountable for their actions.

To alleviate these issues, we combine packets from disparate forwarders at a per-peer *hoodnet proxy*. Each hoodnet peer h has a logical proxy P_h; multiple hoodnet peers may share the same proxy, but for ease of presentation we assume each peer has a dedicated proxy. The main role of a hoodnet proxy is to act as a single point of presence for h and the neighbors of h forwarding on h's behalf. Suppose h is connecting to destination D. P_h bonds the channels of h's hoodnet peers into one high-bandwidth connection to D, and splits packets from D to h's hoodnet peers on the downlink. Proxies also serve an important role in providing incentives and safety by assisting their clients in monitoring peers who (are supposed to) forward for them.

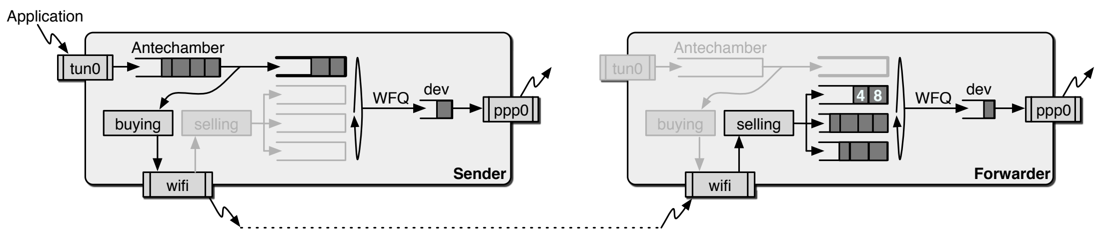


Figure 5.2: Components of a hoodnet peer. Because the leftmost peer's antechamber is full, he has purchased additional capacity at the rightmost peer's.

5.3.2 Two-proxy routing

A hoodnet provides each user multiple paths to their proxy. We present a *two-proxy routing* protocol that bonds these multiple paths to achieve greater bandwidth and connectivity. The protocol is driven by two high-level goals: transparency to the overlying applications, and flexibility for a wide range of user policies. We provide an overview of two-proxy routing in Figure 5.1, and detail it here.

Consider a peer h with proxy P_h and neighbors $f_1, \ldots, f_k$ who have proxies $P_1, \ldots, P_k$. As described in §5.3.1, all packets sent from h must arrive at P_h. h therefore has available to him one path for each hoodnet peer with an uplink; in Figure 5.1(a), h has 3 available paths: from h, through f_1, and through f_2. Upstream paths through a neighbor take the form $h \rightarrow f_i \rightarrow P_i \rightarrow P_h$, that is, each path goes through *two proxies*: the neighbor's and h's. Downstream paths are simply the reverse.

Forwarding through two proxies has several benefits. First, it protects f_i from DoS attacks; f_i need not reveal his cellular link's address to any of his hoodnet peers, because the only host with whom f_i communicates via his cellular link is f_i's (trusted) proxy. Second, as we will see, hoodnet peers have incentive to maintain accounting information on those for whom they have forwarded traffic. This accounting information can be costly—

in terms of computation, storage, and energy—on a mobile device. Routing through one's proxy allows a peer to offload such accounting from the mobile device to a more powerful machine. Last, the proxy can serve as a continual, reliable endhost for monitoring and measuring one's cellular link's bandwidth and latency, which can assist peers when negotiating how much link access to buy from others.

Two-proxy routing comes at the cost of introducing additional latency. However, the extremely high latencies on current cellular links renders two-proxy routing's additional latency marginal. If future mobile access achieves lower latency, one could use Hoodnets for high-bandwidth, delay-tolerant applications, and the direct cellular connection for low-latency applications.

5.4 Buying and Selling Link Access

Hoodnet peers buy and sell link access among one another. Even small hoodnets may contain multiple concurrent sellers and buyers, each with different demands and capabilities. In this section, we present the design of hoodnet markets that allocate resources and payments so as to achieve fair equilibria.

5.4.1 Problem definition

What do peers sell?

Informally, a member of a hoodnet sells to others relatively short-term access to her link. Though tempting to view bandwidth as a sellable good, it is difficult in practice to share bandwidth over a short period of time. Instead, hosts sell a best-effort service of forwarding a specific amount of traffic over a period of time.

How do peers pay?

The markets we define in the rest of this section dictate how much a peer pays another, but here we address what form this payment is in. We assume the existence of a central service that users can access periodically—perhaps even monthly—to purchase and redeem "hoodnet credits." These hoodnet credits are backed by real currency; users can submit credits to the "hoodnet bank" in exchange for money.

An alternative approach is for peers to simply trade goods—uplinks for uplinks. Such an approach relies on peers' expectations that they will interact with some peers repeatedly in the future; otherwise, they would be unwilling to enter a deficit by forwarding for another. While none of the mechanisms we discuss in this section obviate this form of "currency," we expect that in a mobile environment the set of peers with long-lived, interested neighbors is exceedingly small.

Why do peers charge?

A natural question to ask is: why have a market at all? Certainly, users with pay-per-byte cellular data plans would incur a loss for every packet they forward for another hoodnet member. However, as is typically the case in communication networks [93], cellular data prices are increasingly moving toward flat-rate, unlimited plans. Should users not treat their data plan as a sunk cost and give out their links for free?

We contend that there are multiple user-perceived costs for forwarding on behalf of another user. These costs include but are not limited to the cost of the user's data plan (be it flat- or usage-priced). Users who wish to conserve their mobile's battery can charge higher prices, if there is some revenue for which they would be willing to accept a drained battery.

More generally, setting ask prices and bid values allow hoodnet members to express policy as it pertains to data priority and willingness to participate.

5.4.2 Per-packet auction

The first auction mechanism available to Hoodnets peers allows them to attempt to purchase link access on a packet-by-packet basis. Our *per-packet auction* is specified in Algorithm 6. The auction clearing mechanism (step 4) we use is similar to a standard second-price auction, wherein the highest bidder wins and is charged the second highest bid. However, because we expect peers with bursty traffic to occupy many slots in a neighbor's queue, the winner is likely to occupy both the first and second slots. A strict second price auction in that scenario will force most users to pay their bid, thereby giving them incentive to decrease their bids and pay a lower price. We avoid this by having the highest-bidding peer pay the highest bid that was not his. We believe this in fact maintains the spirit of the second price auction, which derives its incentive-compatibility in part because what peers are charged is not directly related to what they bid (only whether or not they win is affected by what they bid).

When a user bids in a Hoodnets per-packet auction, he does not know whether or not he will win. Other users with higher valuations for their packets may outbid the user and not allow his packets to be dequeued before they would be retransmitted. In the extreme, packets could live in neighbors' queues forever. To mitigate this, and to attempt to obtain timely packet delivery, Hoodnets bidders include a time expiry in their packets. When a per-packet auctioneer observes that she cannot meet a submitted packet's expiry, she

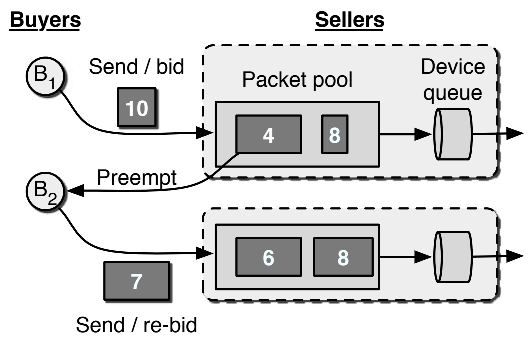


Figure 5.3: Per-packet auctions. B_1's large bid preempts B_2's packet at the topmost seller; B_2 then resubmits her bid to another seller.

informs the bidder that his bid has been rejected. This allows the peer to resubmit his bid, as demonstrated in Figure 5.3.

Per-packet auctions have several benefits that make them an important addition to Hoodnets. First, it allows peers who have bursty, short-lived flows to quickly purchase access to their neighbors' links. Second, it allows peers to *fast-track* individual, high-priority packets by simply placing a higher bid on those packets. For example, a peer could pay more for TCP SYN packets or while opening his TCP window, then scale back his bids once the connection is established.

The main downfall of per-packet auctions comes from their unpredictability and susceptibility to jitter. The increased jitter from being preempted out of a packet auction can induce reordering of TCP packets. This seems to be an inherent problem with per-packet auctions across multiple links. Hoodnets avoids these concerns by not relying on per-packet auctions alone.

5.4.3 Time-based spot markets

To address the concerns with per-packet auctions, we also consider a spot market in which hoodnet peers may buy and sell traffic forwarding capacity over a longer period of time.

Algorithm 6 Per-packet auction.

1. When a buyer wishes to send a packet, he determines a seller to send to, a time by which he wishes the seller to send his packet, and a bid to place for the packet.
2. When a seller receives a packet with a bid, she places it in her packet pool.
3. At any point, if a seller determines that she cannot meet a packet's expiry time, she informs the packet owner that the packet has been preempted.
4. The seller sends packets with the highest bids, and charges the winner the highest bid on a packet that did not originate from that sender.
5. Periodically, each seller broadcasts her average cost charged per packet and time taken to deliver.

The fundamental difference between this market and per-packet auctions is that sellers give buyers *leases* at their links: implicit promises of service over a period of time. Leases reduce jitter, and therefore reduce the chances of reordering.

The fairness properties we seek to maintain are as follows:

- If peer b_1 bids more than peer b_2 at the same hosts, then b_1 should get no worse service than b_2.
- If peer s_1 sets a lower reserve price than s_2, then s_1 should receive no less revenue than s_2.
- No seller should be required to provide a service for less than his stated reserve price, nor should any buyer have to spend more than their stated bid.

In many auctions, these properties fall out as a corollary to the auction being *strategyproof,* that is, an auction in which truthfully stating one's valuations is a dominant strategy. We emphasize that truth-telling is *not* one of our goals in lease negotiation. Negotiating a set of leases across multiple buyers and sellers is a combinatorial problem. Although there has been considerable work in developing strategyproof combinatorial auctions (§5.2), the auctions require considerable communication and computation overhead. These overheads are infeasible when establishing Hoodnets leases; peers may be resource-constrained smart-phones, and the auctions would have to be run every few seconds.

We make use of a simpler construct called a spot market [84]. We describe this market in Algorithm 7. The market matches the buyers who have the highest bids with the sellers who have the lowest *asks*. As a result, only the buyers who make the lowest bids obtain no service, and only the sellers who state the highest reserve prices earn no sales. The clearing price of the market (c) is the average of the smallest winning bid ($b^\star$) and the smallest losing ask ($a^\star$). By construction, $b^\star \geq c \geq a^\star$. That is, no bidder has to pay more than his bid, and no seller has to take compensation less than her ask. The spot market achieves our other desired fairness properties, as well. Any peer who bids more than another obtains no less service, and any seller who states a lower reserve than another obtains no fewer sales.

If a peer wishes to purchase (or sell) k leases, they simply act as k logical buyers (or sellers) in Algorithm 7. Even though each winning participant will be charged or paid the same amount, a peer may wish to submit bids of varying value to mitigate risk and to reflect the fact that increased bandwidth has marginal benefit. For example, a peer obtaining a video stream may wish to make large bids for enough leases to obtain a jitter-free stream rate, as well as smaller bids for enough leases to also improve the quality of the video.

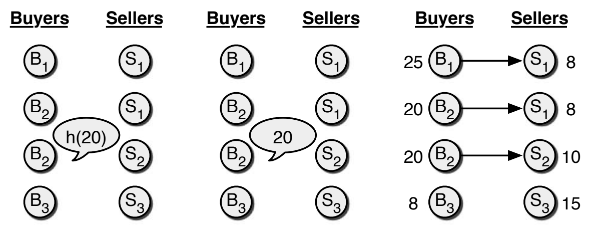
The spot market protocol differs fundamentally from per-packet auctions in terms of how it allows peers to negotiate resources. With bandwidth leases, there is no need to negotiate on packets that are sent within a lease's agreement. Time-based spot markets

Algorithm 7 Traffic-leasing spot market.

1. **Setting**: Peers may bid and sell uplink transit in a system-defined *quantum* of bytes. Each quantum is treated independently as a separate good; without loss of generality, assume that each seller wishes to sell a single quantum, and each buyer wishes to buy a single quantum.
2. **Simultaneous bidding**:
   - Each buyer B_i broadcasts the hash of his bid, $h(b_i)$, and each seller S_i broadcasts the hash of her ask, $h(a_i)$.
   - Each buyer broadcasts his bid, and each seller broadcasts her ask. All participants verify others' actual values match the hash from the prior step.
3. **Winner determination**: Each hoodnet member calculates locally:
   - Sort the buyers in *decreasing* order of bids to obtain, and sort the sellers in *increasing* order of asks; w.l.o.g., assume for all i that $b_i \geq b_{i+1}$ and $a_i \leq a_{i+1}$.
   - Determine the greatest X such that $b_X \leq a_X$.
   - Set the clearing price C to $\min\{\frac{1}{2}(b_X + a_X), a_{X+1}\}$.
4. For each $i \leq X$, buyer B_i wins the quantum at seller S_i at the clearing price C.

do not occur whenever a peer has a packet to transmit, like in per-packet auctions, but rather occur at fixed time epochs: the duration of the leases. Markets are pipelined: while negotiating for a lease of epoch $e + 1$, peers are also forwarding according to their leases of epoch e.

Negotiation in a Hoodnets spot market requires an implementation of a sealed-bid auction. To achieve this, Hoodnets exploits the broadcast nature of the wireless medium; no peer can equivocate, bidding one value at peer p and another at q. We achieve this with the protocol depicted in Figure 5.4. Submitting a bid or ask consists of two phases: (1) a *commit* phase, wherein the user does not broadcast his actual bid but rather commits to it by broadcasting the cryptographic hash of his bid, and (2) a *reveal* phase, where the peer broadcasts his bid. All peers wait to see all others' bid-commits before revealing their actual bid; otherwise, peers who learn of others' bids/asks before having to submit their



(a) Bid commit (b) Bid reveal (c) Goods allocation

Figure 5.4: Time-based spot market. Speech bubbles denote broadcast. Buyers with larger bids win capacity at sellers with smaller asks. This clears with each willing buyer paying 15.

	Per-packet	Time-based
Low jitter		✓
Fast-tracked packets	✓	
Low 802.11 overhead		✓
High cell-link-utilization	✓	

Table 5.1: Properties of the two hoodnet resource allocation methods.

own could easily game the system.

Time-based auctions have the benefit of being resilient to jitter, but introduce their own set of downfalls. Primarily, because peers must wait between auction epochs to submit their bids, they may not be able to obtain leases, and hence forward through others, when they have data to send.

5.4.4 Auction Management

We have presented the two Hoodnets markets, and discussed why we expect neither to be the clear winner, alone. We summarize the benefits and downfalls of our two mechanisms in Table 5.1. Rather than suffer from the downfalls of either mechanism alone, Hoodnets allows peers to use *both* auctions. In the remainder of this section, we discuss the details of how a peer runs his auctions, when he decides to instead use leases, and how he differentiates between his own packets and those he is forwarding for others.

Hoodnet nodes maintain multiple packet queues for managing packet transfer. The cell link is served using a Weighted Fair Queue scheduler. Each lease creates a queue at both the buyer and the seller node. The buyer's queue is served by a the 802.11 scheduler, whereas the seller's queue is added to the set of queues served by the cell-uplink WFQ scheduler. The weights for the weighted fair queue are selected using the bit rates at which leases are sold. For example, assume that a node's uplink is 400Kbps, and that the node has sold two leases of 100Kbps each. Then the two lease queues will get weight $(100/400 = 0.25)$ each, whereas the node will retain weight 0.5 for its own queue.

Nodes insert packets into the lease queue at a rate slightly faster (10%, in our implementations) than the lease rate. This ensures that the lease queue at the seller does not go idle, but also that it does not grow disproportionately.

Packets generated at a node are enqueued in an antechamber that can drain into the node's own WFQ queue or into a lease queue. The maximum size of the antechamber is determined using the node's uplink bandwidth: in our simulations, a node can buffer up to two seconds of data. A packet reaper thread periodically reaps packets that are more than three seconds old. Packets in the antechamber are kept sorted in order of valuation, and packets entering the antechamber may be selected for a packet auction (if enabled). A packet is auctioned, as opposed to buffered in the antechamber, if the product of the number of bytes to head of queue and the packet valuation is greater than the maximum size of the antechamber. This policy allows high valuation packets to be served locally (since they will occupy the head of the queue) and for low valuation packets to occupy the entirety of the antechamber before they are auctioned. Packets with medium valuation will be auctioned with relatively high frequency.

Packet Auctions

The bid on a packet is simply its valuation since the auctions are second price. The auctioning node enqueues auction packets ordered by valuation. Incoming auction packets are rejected only if the total size of packets to auction exceeds a threshold, set identical to the maximum antechamber size.

Auctions packets are transmitted in order of valuation (which is also their bid). The source is charged the maximum of the selling nodes reserve price or the price of any other auction packet *not* from the source.

Periodically, each node broadcasts a report listing the clearing prices for the last k packet auctions ($k = 8$ in our simulations). Bidding nodes merge all of the reports they receive, and choose the successive auction destinations in order of increasing clearing prices. To avoid oscillations and load imbalances, the choice is randomized at node: instead of picking the cheapest destination that has not yet been picked, a node chooses uniformly at random between the next three cheapest destinations.

Lease Auctions

A node chooses to offer a lease if the sum of the bytes buffered in its antechamber (b) and the number of bytes generated locally over the last lease interval (g) is less than half the number of bytes (C) that can be transmitted using its cell uplink in that interval. The number of lease slots is determined to be $\lfloor \frac{C-(b+g)}{q} \rfloor$, where q is the number of bytes that can be served using a single lease. If k leases are to be sold and the node's own reserve price is r, the selling prices are set as $r * 1.1^i$, where i ranges from $0 \ldots k-1$.

A node buys a lease if its own antechamber buffer size (b) and number of bytes generated (g) is greater than its capacity (C). The number of leases bid for is $\lceil \frac{(b+g)-C}{q} \rceil$. Let deficit ($d$) be defined as $(b + g) - C$. The node determines the lease bid values by considering the valuation of its lowest valued d bytes. This is because if a node is unable to get a lease, then these lowest valued bytes will not be sent. Suppose the number of leases needed is k. The k individual bids are set to be the average valuation of the lowest valued $q * i, i \in 0 \ldots k - 1$ bytes generated in the last interval and buffered in the antechamber. For this value to be computed, each node has to log the valuation of all packets generated in the last interval; in our simulation, we keep the valuations of the last 100 packets generated, and extrapolate based on this data.

5.5 Implementation

We have implemented a prototype hoodnet using laptops with dual radios. The laptops are equipped with both 802.11 interfaces as well as 3G EVDO cards for wide-area access.

Our implementation is in C++, runs on Linux, and we have tested it both on Emulab [122] and on commercial 3G networks. Our mobile nodes implement the entire Hoodnets node architecture as described in Section 5.3. We implement two proxy routing in our testbeds. The Hoodnets proxies maintain a memory for temporarily buffering out of order packets for a maximum of 20 milliseconds, after which time they are forwarded on to their destinations.

Hoodnets nodes use the Universal `tun/tap` driver to capture packets from all applications into userspace. The Hoodnets userspace code implements the antechamber and the auction and lease queues as specified in Section 5.3. Captured packets are encapsulated and transmitted according to local policy. The Hoodnets nodes use UDP, with a simple ARQ-based packet recovery algorithm, for forwarding within the 802.11 network. UDP, without retransmissions, is used for communication between the mobile nodes and their proxy. Message payloads are encrypted using AES-128 in CBC mode; messages are signed using 512-bit DSA keys.

Tunnel Flow Control We implemented a fixed-window flow control scheme to provide timely feedback about delivered throughput through both the direct and relayed connections. A node's proxy returns acknowledgments along the (direct return path only or through the relay that carried the original transmission). Acknowledgments include a byte sequence number. We currently fix the window size to 32 KB, which safely overestimates the buffering required for an 800 kilobit per second link with approximately 100ms round trip time, without the likelihood of overwhelming device queues.

This window-based flow control does not include retransmission functionality. Transmissions are pulled from the queue whenever there is space in the window (an acknowledgment is received or the window is not fully used) or when a one-second idle timer expires, to ensure that a link that re-connects is made active. The rate of returning acknowledgments during a round trip time provides an estimate of delivered throughput, precisely as estimated by TCP Vegas [15]. This estimate informs the Hoodnet scheduler whether a lease is delivering its promised performance and informs the Hoodnet auctioneer of the potential capacity to lease to others.

We avoided alternative approaches to rate control including DCCP [62, 63] for three reasons. First, we do not expect the cellular wireless bottleneck to be shared; we are comfortable with the assumption because our target domain is one in which we bond notably

§§	Result
5.6.1	Bonding cellular links on an 802.11 backplane is feasible.
5.6.3	Hoodnets yields 85% average throughput aggregation and effectively mitigates TCP reordering.
5.7	Per-packet auctions handle bursty traffic, lease-markets handle long-term, and combined they handle both.

Table 5.2: Summary of experimental results.

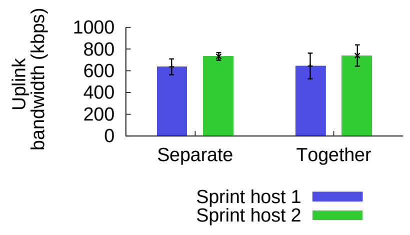
low-capacity channels. Second, we expect bandwidth to vary more from signal quality and auction-based allocation than from queue-based losses; responding to varied allocation quickly is possible with simple flow control, while TCP is designed to be cautious and skeptical of newly-available resources. Finally, we wanted to be able to defer decisions about the next packet to transmit as late as possible to improve the performance for the highest bidder, and not rely on in-kernel buffering to determine when another packet could be sent.

5.6 Hoodnets Evaluation

In this section, we present an evaluation of our Hoodnets implementation on real cellular radios. Our evaluation testbed consists of four laptops, each with a 3G EVDO card—two Sprint and two Verizon—and four Emulab hosts running as proxies. We list our main findings in Table 5.2.

5.6.1 Is Hoodnets feasible?

Hoodnets is built on two assumptions: that multiple cellular links can be accessed without adversely affecting one another, and that the 802.11 backplane has greater bandwidth



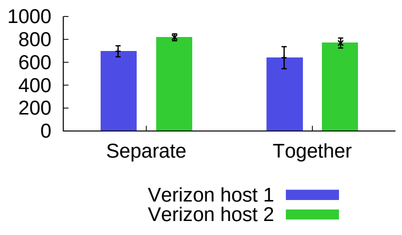


Figure 5.5: Hoodnets are feasible in that simultaneous transmitters do not necessarily impede one another.

than the aggregate of some number of cellular links. We first analyze the feasibility of Hoodnets by testing these two assumptions.

To determine whether two cellular links from the same provider are independent, we measure the bandwidth of hosts from both providers while sending separately and simultaneously. We present the results from this experiment in Figure 5.5, wherein the error bars denote 95% confidence intervals. These results indicate that links are indeed independent for both providers. There is no statistically significant difference in the average throughputs when two hosts transmit separately or together. However, there is a noticeable increase in the variance of the links' bandwidths when transmitting in tandem.

Figure 5.5 also indicates that an 802.11 backplane is sufficient for bonding uplink channels, to a point. If each link were running at its peak, the 802.11 backplane would have to provide no more than 2.4 Mbps in order to bond all four of these links. From these data, we believe that, in most scenarios, the bottleneck of a Hoodnets is the number of proximate users.

5.6.2 Routing overhead

We measure the overhead of routing via a Hoodnets node. Table 5.3 shows end-to-end TCP throughputs for two scenarios: (1) when the node a forwards packets natively using

	Throughput (Kbps)	Latency (msec)
Verizon (native)	794	77.0
Verizon (Hoodnet)	693	220.7
Sprint (native)	521	95.8
Sprint (Hoodnet)	428	93.6

Table 5.3: Mean Hoodnet Routing Overhead.

its EVDO link, and (2) when a second node b is forced to use a's link only to forward packets. In the first case, the nodes forward packets directly to the destination without using any Hoodnets software. In the second case, packets are forwarded to the same destination, but packets are first encrypted and signed. The packets then traverse the 802.11 network to reach node a, go up node a's cell uplink, traverse a fixed network between node a's and node b's proxy, and is then forwarded on to the destination. The throughputs were measured over 90 seconds using the `iperf` tool. We also tabulate the end-to-end latency, measured using `ping`, for the same configurations.

Table 5.3 shows the native bandwidth and latency of node a equipped with a Verizon EVDO card, and the latency and bandwidth obtained by a Hoodnets node when routed through that same node (a). The node's proxy and the destination were located within Emulab. For the Hoodnets experiment, both mobile nodes were hosted by the Verizon and Sprint EVDO networks.

These data demonstrate that Hoodnets traffic overhead is not unreasonable. Further, even over short intervals, the link capacity changes render this difference insignificant. The increase in latency on the Verizon card is attributable to the highly variable latencies we have

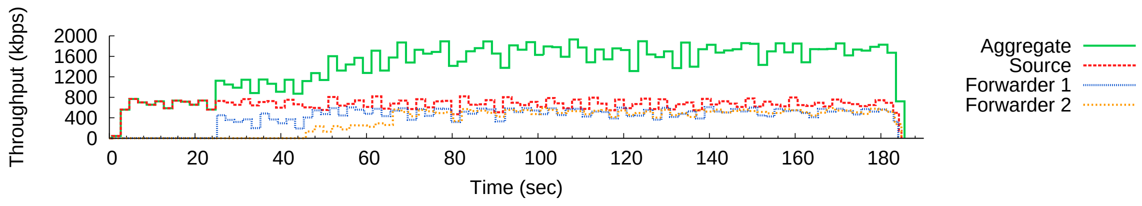


Figure 5.6: A single TCP flow in a hoodnet consisting of three peers.

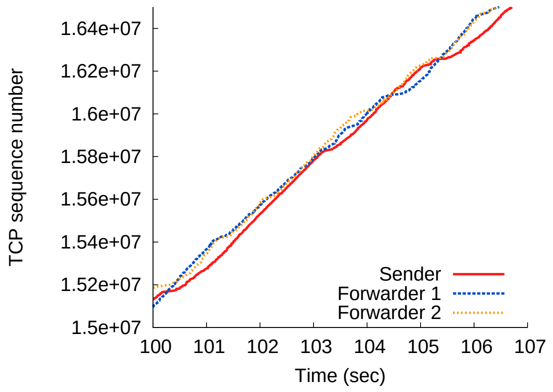


Figure 5.7: TCP sequence numbers received at the proxy. The hoodnet mitigates the sender's bandwidth variance by relying on others' links. (The x- and y-axes do not begin at zero.)

5.6.3 Bandwidth aggregation

We evaluate how well Hoodnets bonds multiple cellular links. The experiment consists of three peers, one of whom initiates a single TCP flow via iperf. Each of the participants is selfish; none of them agree to forward for

The baseline measurements of each of the peers were as follows. We equipped the source node with a Sprint card which we measured to have a throughput of 742 Kbps, while the other two peers, the "forwarders," each had Verizon cards which each obtained an average of 603 Kbps uplink before initiating our Hoodnets code. The maximum possible throughput the sender could achieve in this setting was therefore 1948 Kbps.

We present the results from a sample run in Figure 5.6. This figure captures the throughput Hoodnets obtains on each of the links, as well as the aggregate throughput of the connection; each of these measurements are reported by the source's proxy. In this experiment, Hoodnets obtained an overall average of 1426 Kbps. After the source obtained all of his neighbor's leases, the throughput stabilized to roughly 1675 Kbps: approximately 85% of the maximum aggregate throughput.

The source of the TCP connection forwards through his neighbors by purchasing leases on their links. Initially, the sender had no leases at either of the forwarders. Roughly 20 sec into the download, the sender begins obtaining from one forwarder, and two lease periods later begins obtaining leases at the other forwarder, as well. Figure 5.6 shows that the source gracefully incorporates his new-found bandwidth into his flow.

For each packet he sends, the source node chooses which on which link to send a packet: via his own cellular link, or via one of his neighbors. Our scheduling algorithm estimates which among them will deliver the packet to the source's proxy the earliest. To evaluate this, we plot in Figure 5.7 the TCP sequence numbers as they are received at the source's proxy, colored according to the Hoodnets peer who sent it over their cellular link. This is a small but representative snapshot of the run, taken after the source has obtained full leases at each of his neighbors. The Hoodnets forwarding scheduler successfully mitigates—but does not completely remove—TCP reordering at the proxy, despite the links having different bandwidths.

Figure 5.8 shows results from an experiment that integrates both packet and lease auctions. At the end of each lease period, if the sender still has queued packets in its own queue, then it invokes packet auctions. The dots on the upper plot show successful packet auctions. The experiment uses two Hoodnets peers, a sender and forwarder. The aggregate uplink bandwidth at both peers is approximately 1.4Mbps. Initially, the sender starts

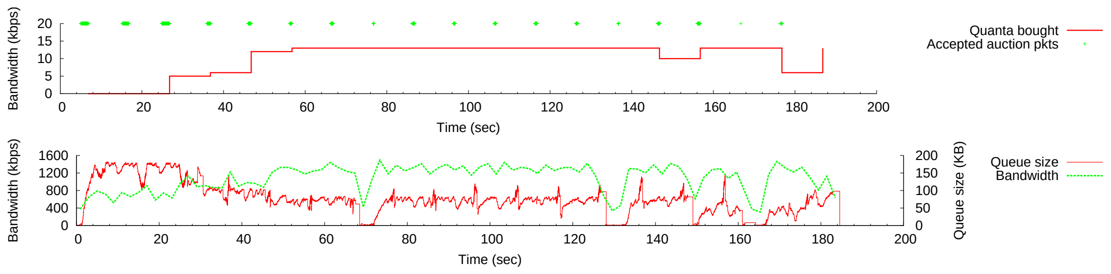


Figure 5.8: Hoodnets with both per-packet and lease auctions.

a 1Mbps UDP flow, which causes its send queue to build, and the sender invokes packet auctions to reduce its queue. Approximately 30 seconds into the experiment, we start a second 400Kbps UDP flow at the sender. Packet auctions are no longer sufficient, and the sender starts to lease bandwidth. The precipitous drop in locally queued packets and occasionally of throughput is a result of immediately loading all leases at the beginning of each epoch; after this experiment, we realize flow control must apply both to the direct link and to the relayed links.

These findings demonstrate that Hoodnets efficiently aggregates selfish neighbors' links. Furthermore, Hoodnets is able to do so without pinning connections; rather, Hoodnets exports a single, high-bandwidth link to its users. To the best of our knowledge, this is the first demonstration of a TCP flow being striped across multiple cellular links.

5.6.4 Load balancing with auctions

Next, we investigate the local state at a peer when deciding to bid on auctions. We plot in Figure 5.9 the number of packets in the sender's antechamber throughout the transfer described in §5.6.3. We overlay the number of quanta the sender purchases in total from his two neighbors. Initially in the transfer, the sender locally enqueues packets at a rate greater than he can dequeue, because he does not yet own any leases. This state

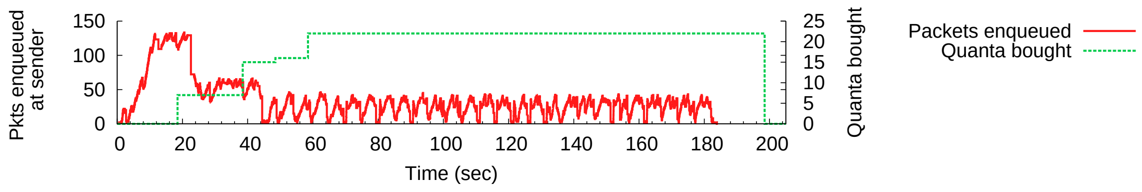


Figure 5.9: Hoodnet lease auctions react to, and alleviate, the number of bytes enqueued at the sender.

triggers the peer to begin bidding in lease auctions, which he wins roughly 20 sec into the trace. Obtaining these leases quickly mitigates the number of packets enqueued in the antechamber. As the peer continues to obtain more leases (until 60 sec, at which time he has purchased all of his neighbor's leases), the number of enqueued packets in the antechamber decreases.

5.7 Auction Analysis in Simulation

To better understand auction dynamics in settings that are both controlled and larger than our testbed, we have implemented a custom Hoodnets simulator in C. Simulated nodes implement the Hoodnets node architecture as described Figure 5.2, and the simulator allows customizable auctions with various bidding and pricing policies.

Each node in the simulator may host many simultaneously active packet generators. The packet generators should be thought of as different end-to-end flows active at a node. Each packet generator can be customized with the distribution of packet inter-arrival times, packet valuations, and packet sizes. (Recall that a packet's valuation is the maximum the source is willing to pay for that packet's transit through another node.)

The simulator includes both nodes and proxies. Data packets are always destined to a node's proxy, and can be routed directly over the cell uplink. Packets can also be auctioned; auctioned packets follow the two-proxy route as described in Section 5.3.

The link performance parameters are as follows. Nodes simulate uses 400Kbps cell uplinks; the one-way cell delays is set to 40 milliseconds. The hoodnet 802.11 network runs at 54Mbps, as does the inter-proxy network. Each packet is either delivered to the destination, or is reaped from the simulation after 4 seconds of simulated time. The simulated links do not model dynamic wireless channel conditions or drop packets. A packet is dropped only if:

- the packet cannot be accommodated by the local node's antechamber and it cannot be sent via other nodes. The latter happens if the source node does not have any active leases, and the packet has lost multiple auctions at its maximum valuation.
- the packet is reaped because it could not be transmitted for 4 seconds.

The simulator logs the lifecycle of each packet: when the packet is generated, what the valuation is, how it transits the system (directly to the destination or though a hood member due to an auction), and whether it is delivered or lost.

For each node, the simulator tallies the total valuations of the packets generated, the total valuations of packets that are delivered or are lost, the total amount the node earns and spends hosting and selling at different auctions.

5.7.1 Auction Interactions

Our goal with the Hoodnets simulator was to understand the answers to two related questions:

- When are two types of auctions necessary? (Alternately, is it sufficient to have only the packet/lease auctions?)

- (How) do different auctions and clearing mechanisms affect end-to-end performance?

We begin with a ten node Hoodnets (ten wireless nodes and ten proxies). For this initial example, we use a packet generator which generates 500 byte packets with packet inter-arrival times picked uniformly at random. The packet generator does not back-off when its packets are dropped in the simulation. Packet valuations are held constant.

Only two nodes, 0 and 1, generate data packets, and the other nodes sell their uplinks using auction strategies described in Section 5.4. The sources are over subscribed: without any auctions in the system, exactly 50% of the generated packets are lost.

With packet auctions enabled, no packets are lost in the system. Lease auctions require a round of observations to set bid and ask prices. During this initial interval, as expected, 50% of the packets are lost. Once lease auctions commence, the loss rates converge to that of packet auctions. Considering loss alone, there is not a strong case to advocate for either auction.

Figure 5.10 shows the difference in successive packet sequence numbers as they are delivered to the destination's proxy. The y-axis is in log scale. Packets received in sequence are contribute to the line at $x = 0$; the data shown is only for Node 0's generator. Node 1's data is similar. This data exposes a stark difference between auction performance: packet auctions introduce reordering, whereas leases provide much more stable link. The susceptibility of TCP to packet re-ordering is well documented [14] and indeed verified by us in our testbed. Using leases, a 12 packet reorder buffer is sufficient for 95% of all packets delivered.

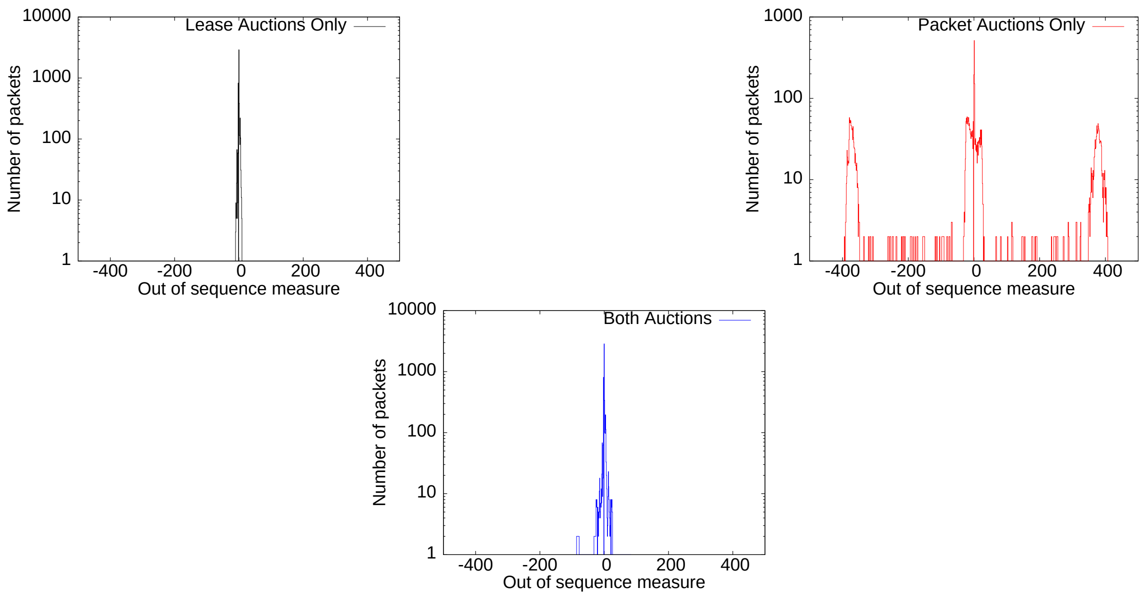
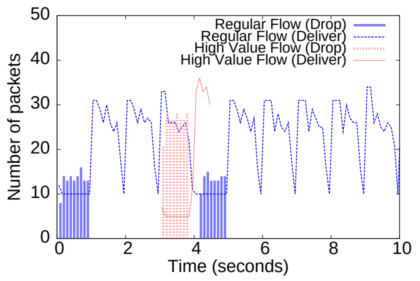


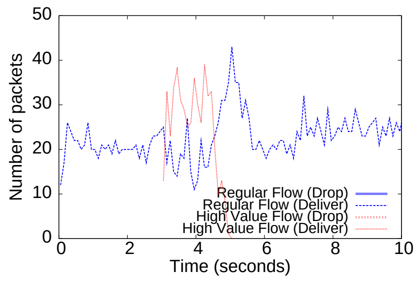
Figure 5.10: Out-of-sequence packets received at proxy. Y-axis is in log scale.

In contrast, the reorder data for packet auctions only shows a much different behavior. Packet auctions are invoked when the source node's antechamber cannot accept a packet. (The precise condition incorporates both the occupancy of the antechamber and the packet's valuation.) The packet generators in this scenario generate equal valuation packets, which implies that differing valuations do not cause reordering. These packets are then auctioned at other nodes, and these auctions are cleared in order of original bid (which in our case are all the same and hence do not cause any reordering). Each auctioning node rejects incoming auction packets if its auction queue is too full, and the packets can then be auctioned at other nodes. Each packet can be rejected in up to five auctions before it is discarded. The modes around 400 and -400 are for packets that win an auction, but are queued for a long time at different auction queues. Only 23% of the total packets are captured in the 12 packet buffer.

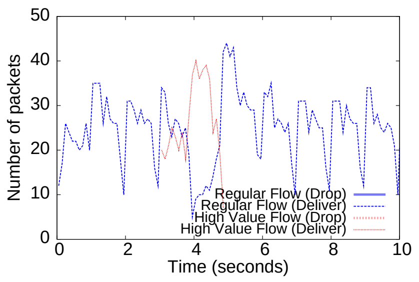
When both auctions are used, packet auctions clear the overload before leases can commence (1 second into the simulation), at which point each node is able to estimate its



(a) Bandwidth lease auctions alone



(b) Packet auctions alone



(c) Both auctions

Figure 5.11: Evaluation of burst tolerance of leases-only, packet-auction-only, and combined auction mechanisms.

demand and obtain appropriate leases. A 12 packet buffer, in this case, captures 94% of all packets.

Bursty Traffic Figure 5.11 shows the packet loss and delivery behavior when a bursty source is introduced at time 3 seconds. The source of the regular flow is Node 0. The bursty flow (from Node 2) is of higher bitrate and the burst packets have higher valuation. Figure 5.11 plots the number packets lost and delivered over time for packet-auctions only, lease-auctions only, and both auctions enabled. Each sample is the count of the packets lost or delivered over the last 100ms.

The lease-auctions only plot shows that packets are lost in the beginning (time < 1 second) before leases commence. The rate at which packets are delivered in the first second corresponds to the service rate of the source node only. Once leases are established, there are no more losses until time 3 seconds, when the bursty flow which has no leases starts. The periodic behavior of the delivered packets is due to the lease queues emptying as leases run out every second.

The high valuation burst starts at time = 3 seconds. The burst source immediately starts dropping packets since it has no leases in place. The service rate of the burst in its first second (between 3 to 4 seconds on the plot) is even less than the service rate obtained by the regular flow during time 0 to 1 second because the burst source had leased some of its own bandwidth out! After the first second, Node 2 (burst source) is able to obtain new leases, and the high valuation flow does not lose any more packets. However since the burst is of higher valuation, Node 2 is able to beat Node 0 on a few lease auctions, and Node 0 ends up getting fewer leases, and starts to drop packets during time 4 to 5 seconds. The burst ends during the fifth second after which time Node 0 does not lose any more packets.

The packet auctions are able to accommodate the burst without any loss. However, since burst packets are of higher valuation, they win most of the auctions while the flow is active, and the service rate to the regular flow's packets decreases (time between third and fifth second), and the regular flow packets queue up in the auction queues until the burst subsides.

The combined scheme does not lose packets either. Unlike the packet-only scheme, the regular flow's packets are not affected during second 3 when the burst starts since the regular flow has already purchased leases for that time period. The burst is cleared using packet auctions until second 4 when the burst source is able to lease sufficient bandwidth. Like with lease-auctions only, the burst source (Node 2) is able to obtain more leases and obtains more bandwidth during time 4–5 seconds. The regular flow recovers after the burst subsides, and obtains leases as necessary for the remainder of the simulation.

These two experiments demonstrate the necessity of our multi-auction model. Packet auctions are useful for clearing short unexpected traffic bursts, but high throughput stable flows are best served using leases. Sellers also have incentive to offer both types of auctions: lease-auctions provide guaranteed revenue, but may monopolize the seller's uplink capacity. If the seller reserves a fraction for its bandwidth for he own use, then packet auctions can "use up" this reserved bandwidth if it remains unused.

The system designer has to choose the length over which leases are served: a longer lease time allows more stable bandwidth allocation, but requires better prediction of traffic. Further, as lease times get longer, the amount of residual capacity in the system decreases; this can lead to lower overall utilization since the available bandwidth that can be used by packet auctions opportunistically decreases.

5.8 Discussion and Limitations

In this section, we describe limitations of the Hoodnets prototype we developed and speculate about the potential business model involved in providing Hoodnets software, accounting, and proxies as a service.

We use a simplistic model of packet valuation. In our prototype, packets within a flow are given monotonically decreasing value. This valuation is based on the assumption that the first few packets in a flow are likely to be performance sensitive, while later packets in long flows are less likely to be urgent. It could be, however, that the long transfer is important and short exchanges are not. A more realistic valuation of packets, one that takes user input, application state, and interactivity into account, is an open problem [108, 39]. We believe that Hoodnets provides a platform for experimentation with rules that assign value to packets.

Our prototype acts entirely at the user level, using the `tun/tap` interface to collect packets from the kernel. Migrating fast-path packet scheduling functionality into the kernel might avoid unnecessary user/kernel boundary crossings, which might reduce the computational effort on devices with limited processing power.

Our tunneling protocol uses a fixed window-based flow control approach for both direct and relayed connections. Ideally, this window would adapt to permit the decision to transmit a packet to be delayed until the last moment, preserving a nearly empty queue. Although we are aware of techniques that do so when link latency is fixed (e.g., the BaseRTT in TCP Vegas [15]), it is not clear how to best adapt to varying latency and throughput that are driven by changes in signal quality and allocation decisions rather than directly by contention.

The relationship between Hoodnets and 3G service providers is uncertain. Providers do offer wireless sharing devices (e.g., Verizon and Sprint offer the Novatel MiFi [92]) without inflated cost, suggesting that wireless service is not inherently priced per user. Extending the sharing concept to permit fine-grained resale of connectivity does not seem unlikely.

We expect that Hoodnets would be deployed as a service by a 3G service provider or by a third party that would implement the forwarding software and a payment service, comparable to that of FON [43]. Two users would be able to participate in such a Hoodnets if they both used the same Hoodnets provider. The Hoodnets provider could charge a subscription fee for participating in the Hoodnets, may charge a fee on each auction, or both.

Interestingly, it makes sense for Hoodnets to be offered as a service by the 3G provider themselves. Hoodnets provides a transparent and economical way for providers to improve user performance without deploying expensive new technology. Like third party Hoodnets providers, cell providers need not be restricted to Hoodnets that use only their own 3G links. Existing roaming relationships could be leveraged for using bandwidth from other providers. Such an arrangement might be particularly useful within public transit systems on which only a few providers provide data service.

5.9 Conclusion

Hoodnets permits users to share their low-capacity mobile links. To help selfish, independent agents distribute limited bandwidth resources with both responsiveness and predictable performance, Hoodnets uses two forms of auction: packet auctions and spot-market-allocated bandwidth leases. By using auctions to allocate scarce upload capacity, nodes can collect payment during their idle time by forwarding for others, and spend that credit when busy or when temporarily disconnected.

We implement Hoodnets using two-proxy routing. Each node has a proxy that acts as an Internet-side representative of the client for both privacy and accountability for malicious traffic, and acts as a wired-side relay station that can retransmit the packets of others, should another node's proxy fail to receive relayed traffic. To address the variability of performance characteristics of such networks, we use a simple window-based flow control approach with a scheduler that predicts which relay will best provide in-order delivery at the proxy.

In the future, we plan to develop both policy and mechanism. We do not specify an approach to providing a valuation on individual packets; we expect that simple models developed for in-network traffic prioritization [95] could be applied. On the auctioneer side, we have yet to investigate the characteristics of a Hoodnets market when one provider lacks coverage. Taken together, applications that can adapt to expensive connectivity may be promising future research.

Chapter 6

Punishment Pitfalls

The previous chapters' reward-based mechanisms provide incentives for proper behavior. Conversely, the incentive mechanisms we discuss in this chapter lead participants to punish one another when they detect acts of misbehavior, thereby providing disincentives for improper behavior. The primary difference between punishment and reward is that in the ideal case—that is, when no participant misbehaves—punishments would never be executed, whereas rewards would be constantly exchanged. There is therefore a systems-compatibility appeal to punishment; if we could expect that the mere threat of punishment would be enough to achieve proper behavior for the most part, then we would not have to pay the overhead of the incentive mechanism in the normal case. Furthermore, punishment is appealing in that it typically does not require an extensive infrastructure or high barrier of entry like those of a monetary system.

Although seemingly appealing, punishments introduce other sources of communication overhead, as well as a set of requirements from systems. For example, a system must be able to detect when one of its participants has misbehaved in order to determine when to enact punishment.

We study in this chapter whether or not punishment-based incentive mechanisms are systems-compatible. Specifically, we analyze what features a system must support in order to accommodate punishment as a disincentive for misbehavior. To drive our discussion, we use two representative systems. First, we consider a rather subtle form of punishment: propagating negative reputation information. This example demonstrates how to identify an act of punishment, and covers the properties that such an act must exhibit in order to be viable. Second, we present the ground-up design of a file-swarming system called FOX (Fair Optimal eXchange) that uses punishment to ensure that no peer can free-ride from others. The primary design challenge behind FOX, as in any punishment-compatible system, is the ability to detect and prove that a peer has not acted in accordance with the protocol. FOX introduces several mechanisms to support misbehavior detection that can be applied more generally.

6.1 Punishing with negative reputation

Reputation systems allow participants to exchange with one another information regarding the interactions they have had with others. When two peers interact for the first time, they can draw from others' experiences to infer how trustworthy one another is. There are two general forms of reputation: positive and negative. Peers have incentive to accrue positive reputation, so that they may leverage their good deeds in the past to curry favor from others with whom they have not interacted. Considerable work has gone into ensuring that they cannot arbitrarily inflate their positive reputation [58, 23].

In this section, we focus on negative reputation. When a peer p is mistreated by another peer m, p forms a local, negative view of m, and may either degrade service to m or cease communicating with m altogether. Sharing this experience with others, in other words establishing a negative reputation for m, is a powerful way to quickly weed out free riders and malicious peers from the system.

We study the feasibility of such an approach by first observing that providing negative reputation is a form of punishment. It is rather subtle, especially when compared to BAR gossip's blacklisting [76] or jamming a wireless channel [71]. However, it is indeed a form of punishment: peers implicitly threaten one another with posting negative "feedback," and having accrued a negative reputation can result in a loss of utility, such as fewer willing traders. As with any form of punishment, it is crucial to understand whether or not threats thereof are credible.

6.1.1 Credible threats

In a punishment-based mechanism, it is not the punishment itself that maintains cooperation. Rather, it is the *threat* of punishment that keeps selfish participants from defecting from the protocol. For a potential defector to take a threat from user u seriously, it must be clear to the defector that u would be willing to follow through with the punishment. In game theory parlance, this means that threats of punishment must be *credible*.

In other words, peers require incentive to punish one another. A mere act of "revenge" is not necessarily sufficient motivation to perform the work involved in doling out a punishment.

Further, the threat must typically be to punish for at least as much as the defector gained from performing a punishable act. For instance, if a peer cheated to obtain an additional 10 units of utility, then a subsequent punishment would have to incur a cost greater than 10 to ensure that the peer has enough disincentive to not cheat.

6.1.2 Is negative reputation a credible threat?

Suppose that peer p has been mistreated by peer m. Although p has implicitly made the threat of punishing m by reporting a negative reputation score, does p have incentive to follow through with this threat?

In many systems, peers compete with one another to obtain service from others. BitTorrent peers, for example, attempt to upload more to their neighbors than others do, so as to receive reciprocal bandwidth [30, 98]. If p realizes that m is a "lost cause"—that m does not return any data or that m consistently returns corrupted data—then p certainly knows to no longer upload to m. Instead, p will upload to a set of peers G he has observed to be good. However, p may wish for *other peers* to upload to m, as it diverts bandwidth they may have otherwise spent at peers in G.

In systems such as this, peers have a disincentive to report "bad deals," as it may draw greater competition for good deals.

This raises two natural questions: can systems provide peers incentive to truthfully report *all* interactions they have had with others, and can they do so in a systems-compatible manner? To the best of our knowledge, the most apropos systems that achieve this are accountability systems, such as PeerReview [52] and Nysiad [54]. These systems augment an existing protocol with an "accountability layer" that adds signed digests of messages sent in the underlying protocol. Both of these systems require considerable infrastructure, such as a PKI, and communication overhead, typically super-linear in the number of messages from the underlying protocol.

This example demonstrates that even a seemingly innocuous form of punishment like that of forwarding negative reputation can in fact require extensive supporting infrastructure to ensure truthful, cooperative behavior.

6.2 Grim triggers in file swarming

File swarming is a popular method of coordinated download by which peers obtain a file from an under-provisioned server. Critical problems arise within this domain when users act selfishly, yet most systems are built with altruism assumed. Working under the assumption that *all* peers are greedy, we introduce the Fair, Optimal eXchange (FOX) protocol. We prove that the fully-decentralized FOX overlay structure provides theoretically optimal download times when everyone cooperates. In the presence of free-riding, FOX offers two mechanisms: (1) a local, robust application of tit-for-tat when there are few free-riders, and (2) rate-limiting at the server only in the event of rampant free-riding (e.g., collusion) We show that this combination of two mechanisms ensures that a fair, subgame-perfect Nash equilibrium can always be obtained and subsequently maintained. We begin in an idealized setting and prove FOX's optimality and incentive properties, even when the network consists only of purely self-interested peers. We then present a deployable protocol that implements the FOX structure while maintaining its incentives and providing near-optimal performance.

6.2.1 Selfish bulk data transfer with FOX

In the remainder of this chapter, we consider the specific problem of some large number (N) of peers attempting to download the same file from a server. Our solution, called

FOX (Fair Optimal eXchange), creates an incentives framework by applying threats on a *structured* overlay topology, simultaneously providing both theoretically optimal download times and incentives to cooperate. We also introduce an algorithm that solves the so-called *last block problem* (in which peers leave the system as soon as they receive the last block, devolving into a complete lack of system-wide cooperation). The protocol we present for FOX requires very little work at the server and, unlike previous systems, is distributed, difficult to exploit, and provably fair.

Rather than assume that some subset of users are altruistic, we expect *each* peer to act in a purely greedy way. Specifically, we assume that *any peer p will help or hurt other peers only if it improves p's download time*. Within this domain of self-interested peers, the overarching goal of our work is to investigate what primitives are necessary to obtain provable fairness. In many cases, such guarantees may be overkill; it has been observed that some BitTorrent "communities" consist of a group of altruistic users who share a niche interest [7]. Our work focuses on the more general problem of ensuring users of system-wide fairness without any prior knowledge of the other participants.

There are of course many definitions of fairness. Exact fairness, in which each peer uploads exactly as much as it downloads, requires a degree of bookkeeping that is not possible to reliably implement scalably in large systems without adversely affecting peers' download times. Instead, the fairness we aim to provide can be described informally as follows: *gross violations will be detected and, more importantly, no rational peer will grossly deviate from the FOX protocol*. FOX achieves this property; in fact, in an idealized setting, FOX peers have no incentive to deviate at all. We present a formal, rigorous set of fairness properties in Section 6.2.5.

We consider the case where all peers are interested in downloading the file, and do not consider maliciousness. In most situations, this is a strong assumption, but we remind the reader that file swarming inherently has a single point of failure; a malicious node may simply attack the server.

Game Theory Concepts

We provide incentives to selfish peers to cooperate by employing game theoretic incentive mechanisms. Here, we provide a brief overview of the concepts of game theory that are pertinent to our presentation of FOX. For a more thorough treatment, please refer to [94].

Game Theory is a branch of microeconomics and mathematics that studies the interactions of two or more rational, self-interested players. Each player chooses an action, called a *strategy*, to take when playing a game. The set of chosen strategies is called a *strategy profile*. Upon completion of a game, each peer is awarded some *utility*, which is a player-specific function of the strategy profile. Peers are said to be rational if they always act in accordance with strategies that will yield them the greatest available utility. In *repeated games*, participants play the same game multiple times, keeping track of (some subset of) the history of the actions taken by players with which they have interacted and the resulting utilities.

One of the strongest results to have come from game theory is that of a *Nash equilibrium*. A strategy profile is said to be a Nash equilibrium if, for each player, no deviation would result in increased utility. A *subgame perfect Nash equilibrium* (SPNE) defines a set of strategy profiles over a repeated game such that, at each subgame (i.e., each iteration of the game), the strategy profile for that subgame is a Nash equilibrium. Another way to think of subgame perfect Nash equilibria is as a Nash equilibrium consisting only of *credible threats*, that is, to be a SPNE, no player can imply that they will defect from the Nash equilibrium at a later iteration of the game. Useful in formulating SPNEs are *incentive mechanisms*, which help guide players to an outcome that is as good as possible for all involved, or *socially optimal*. There are also incentive mechanisms that do not give rise to SPNEs, such as the *grim trigger strategy*, in which, when a player detects that it is being cheated by the other players, it causes all cooperation across the system to cease (in effect, the entire system collapses if a single peer is unhappy).

What Is Wrong with Tit-for-Tat?

Tit-for-tat (TFT) is a simple, intuitive incentive mechanism. In its simplest application to file swarming, peer p sends peer q a block but does not send more until q sends a block in return. If q wants more blocks from p, it has the incentive to act fairly and return p's favor. There are many obvious variations on this theme; p could simply reduce the quality of service it offers q until q sends blocks to p, or p could continue sending until q develops too large of a deficit. The former (reduced service) is similar to what BitTorrent currently uses in its choking algorithm [13]. TFT has found its way to such widely used systems, so why propose a change now?

Current file swarming systems do not provide the proper basis for applying TFT. For two-player repeated games, TFT can prove to be an effective mechanism. In particular, TFT forms a subgame perfect Nash equilibrium for many repeated games, the repeated prisoners' dilemma perhaps being the most well known. However, current file swarming systems allow peers to interact with any other peer in the system; greedy users may not be required to repeatedly interact with any other peer and therefore not give anything back to the system. It is therefore not clear what, if any, global properties can arise from applying TFT to systems such as BitTorrent, Bullet$'$, and Slurpie. Even less clear is whether a proof of the fairness properties TFT provides to these systems can be derived. Further, with unrestricted interaction, the amount of state peers must keep to maintain the history of their interactions can grow linearly with the number of peers.

The Need for Global Threats

FOX differs from previous file swarming systems in part because it reduces the set of other players with which a peer can interact. Peers must therefore play repeated games (with a substantial number of iterations) with one another, as opposed to moving from peer to peer throughout the system. We can therefore apply TFT with the assurance that it will provide incentives to act fairly.

However, TFT alone is not sufficient for ensuring fairness in FOX; in a sense, it is too localized. Peers could, for instance, choose to collude and isolate others from the swarm. To give the stranded peers a means of punishing these colluders, FOX also provides a system-wide threat model: peers may request the server to send at a slower rate, or *rate-limit*.

We introduce the FOX topology in Section 6.2.3. Within this hypothetical setting, we show in Section 6.2.5 that FOX is optimal and provably fair in that, by making use of incentive mechanisms introduced in Section 6.2.4, it provides incentives for all but a constant number of peers to upload as much as they download. It will become clear, however, that this hypothetical structure cannot immediately be realized in the Internet.

We present a FOX protocol in Section 6.2.6 that is easily implementable and show that it maintains the same incentives and expected efficiency. We also provide results from a proof of concept, message-level simulator. In Section 6.2.9, we present further relaxations to the global threat mechanisms we introduce.

6.2.2 Related Work

File swarming has received significant attention from researchers and users alike. We present an overview of the various works in terms of the incentive mechanisms they use. Many systems do not provide any incentive. For instance, application-level multicast systems [11, 20, 19, 24] can be considered some of the earliest work, but they generally assume that the participants are altruistic. At best, SplitStream [20] allows peers to specify some maximum amount of service they are willing to provide, but, without an explicit minimum amount of per-peer service, many users may potentially free-ride. In lieu of a multicast tree, some systems create a mesh, in which, in the ideal case, all nodes send and receive portions of the file (henceforth *blocks*), thereby “filling the pipe” with data to provide much better download times [30, 17, 64, 110].

In mesh-based systems, since each node trades blocks for new blocks, designers have attempted to apply TFT as an incentive mechanism. This is most evident in BitTorrent [30], which uses a variant of TFT in which the number of blocks a peer p sends to peer q determines the quality of service q provides to p. As described above, however, TFT does not provide strong global properties when applied to an unstructured system. Note also that, with TFT, blocks can be viewed as tradable currency. In general, without a means of verifying currency, forgery will occur; peers could simply forge garbage blocks and trade them for real ones. A means of verifying the currency in these systems is likely to involve additional work at the server (e.g., the server could sign each block). In Section 6.2.6, we present a protocol of FOX which requires the server to perform source coding, but we show in Section 6.2.5 that FOX peers do not have incentive to forge garbage blocks.

Other mechanisms use currency that is more closely analogous to money [16, 48, 127]. In these systems, peers can obtain digital currency by paying for it with real money or by providing service to other peers. Currency exchange between peers is a difficult problem to implement; existing systems require tamper-proof hardware [16] or a trusted, centralized authority [127]. Designing a distributed, easily deployable, altruism-free system that does not require a central trusted node (such as the issuer of currency) is the focus of our work.

Yet another general incentive mechanism is reputation, in which the more a participant aids others, the better reputation it receives. The better a node's reputation, the better service it receives. Conversely, malicious or greedy users obtain poor reputation, thereby reducing their quality of service. There are several systems which store reputation (or trust metrics) in a distributed, robust fashion [58, 68]. Of particular concern with these systems is the time to converge; when a node acts maliciously, it takes time for this information to be disseminated throughout the system. Greedy nodes could exploit the system during this potentially large window. Though reputation-based systems are useful for situations in which users are expected to stay for long periods of time, this exploit makes them unfavorable for applications in which users do not wish to linger in the system, as in file swarming. We avoid such issues by maintaining that reputation information need not be propagated; all reputation in our system is stored on a purely local, per-neighbor basis.

We use FOX's structured, cyclic topology to provide a means for peer to punish nodes both upstream and downstream from it. A similar approach is taken in [89], in which peers periodically reform a SplitStream structure, with the hopes that peers who were downstream from a cheater will later be upstream from it, thereby giving it the ability to punish. In addition to the overhead of reforming the structure, peers must wait for a new restructure to even have the possibility of punishing free-riders. FOX's structure allows nodes to punish free-riding as soon as they detects that someone is doing it, without requiring the topology to periodically be rebuilt.

FOX provides incentives while only requiring nodes to monitor their immediate overlay neighbors. Scrivener [86] employs a similar approach, but assumes the different model of distributing potentially many different files. Conversely, with FOX, we are interested in optimizing download times for a given popular file in addition to maintaining strict fairness properties.

6.2.3 The FOX Overlay Structure

In this section, we present the Fair Optimal eXchange structure, a P2P overlay that provides optimal download times to its peers. We use its structure as a basis for applying tit-for-tat. Coupled with the k-meltdown model, it provides incentive to peers to follow the protocol and thus act fairly toward others. To more easily intuit about the structure and peers' potential resulting behavior, we introduce FOX with some simplifying assumptions on the number of nodes in the system and their capabilities. Within this setting, we are able to prove FOX's optimality and the incentives it offers. In Section 6.2.6, we present one implementation of FOX that removes these assumptions.

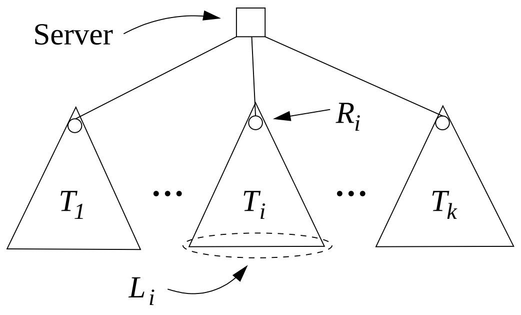


Figure 6.1: Notation pertaining to the tree structure.

Notation and Assumptions

There are N clients attempting to download a file of B blocks from one server. In this section, we assume that all nodes join the system before the server begins transmitting any blocks and none of them depart until they have completed the download. Per the k-meltdown model, the server and each client is assumed to have a maximum outgoing bandwidth of k blocks per time step. For ease of exposition, assume that there are enough nodes to fill a k-ary tree of height H rooted at the server, that is

$$N = \frac{k^{H+1}-1}{k-1} - 1 = \left(k^H - 1\right)\frac{k}{k-1}$$

Consider the tree in Figure 6.1. Let $R_i, 1 \leq i \leq k$, be the k peers connected to the server and let T_i be the subtree rooted at R_i. Lastly, let L_i denote the set of leaves of T_i. To arrive at our final structure, we will add loops to the tree in Figure 6.1, but will continue to refer to the members of the L_i's as leaves and $T_i \backslash L_i$ as internal nodes.

Filling the Pipe

To provide quick download times, we first fill the system's pipe, that is, we make use of each available outgoing link. We demonstrate how we do this in an iterative manner, beginning with the multicast tree in Figure 6.1. In a full, balanced, k-ary tree of height H, there are k^H leaves, thus with just the multicast tree, the fraction of nodes *not* sending data is

$$\frac{k^H}{N} = \frac{k^H}{k^H - 1} \cdot \frac{k-1}{k} > \frac{k-1}{k}$$

Clearly, the multicast tree alone is not an adequate use of the available resources. We therefore make use of the leaf sets' links as follows.

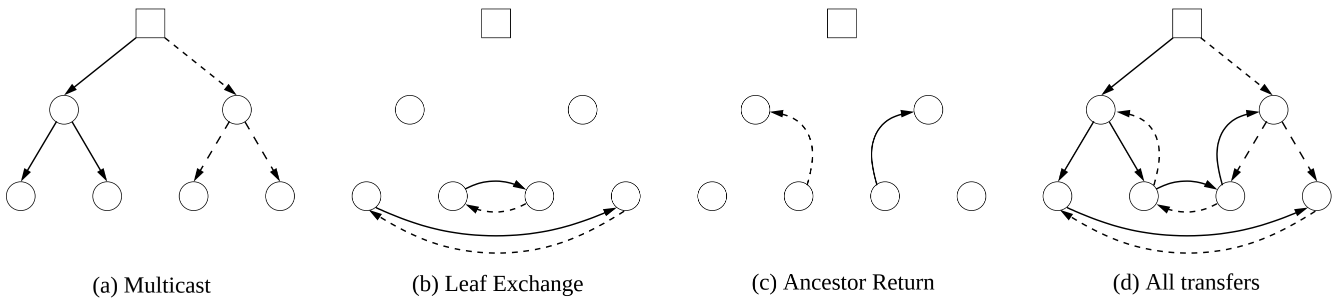


Figure 6.2: Example of FOX with $k = 2, N = 6$. Solid arrows denote the transfer of blocks b_{kt+1} and dashed arrows blocks b_{kt+2}, $0 \leq t \leq \left(\frac{B}{k} - k\right)$

Instead of obtaining the same block, each first-level child, R_i, requests a different block from the server; at time step $t \geq 0$, R_i retrieves the $(kt + i)$th block of the file. Note that, unlike most other file swarming systems, this requires no server involvement; the R_i's can choose their respective i's amongst one another. Let $\mathcal{B}_t$ be $\{b_{kt+1}, \ldots, b_{kt+k}\}$, the set of blocks uploaded by the server at time t. Once it receives b_{kt+i}, R_i multicasts it throughout T_i, as in Figure 6.2(a). When $v \in L_i$ receives block b_{kt+i}, it trades with some set of $(k-1)$ leaves, each from a different T_j. There are enough leaves to create pairings in which each leaf, in a single round, gets the $(k-1)$ blocks it lacks from the set $\mathcal{B}_t$ (we give one such method in our full protocol in Section 6.2.6). As shown in Figure 6.2(b), each leaf p only sends blocks to leaves which send blocks to p, as will be required by our incentives scheme (see 6.2.4).

Note that, at this point, each leaf has each block in $\mathcal{B}_t$ and each internal node $v_i \in T_i \backslash L_i$ has block b_{kt+i}. Further, observe that there are more than $(k-1)$ times as many leaves as internal nodes. We therefore have the resources to deliver to each such v_i the $(k-1)$ blocks it is missing, $\mathcal{B}_t \backslash \{b_{kt+i}\}$, in one additional step; the leaves simply connect directly to an appropriate ancestor. We show this in Figure 6.2(c) and the resulting structure in Figure 6.2(d). In a full k-ary tree, all but k leaves have a total of k peers to send to: the $(k-1)$ other leaf nodes and the 1 internal node. The k other leaves will have $(k-1)$ connections to other leaves but no ancestor pointer, as exemplified by the leftmost and rightmost leaves in Figure 6.2(d). This accounts for the fact that one node in the system does not require any in-degree — the server itself — while all other nodes have in-degree k.

The connections between members of L_i and the internal nodes of T_i follow the same paradigm as the inter-leaf connections: "send data to those who send data to you." Specifically, a node $v_i \in L_i$ sends one of the $(k-1)$ blocks it received from leaves in the other T_j's to one of the nodes on that path from v_i to R_i. This methodology forms the basis of an efficient use of tit-for-tat which we show below.

Avoiding Last Block Collapse

Suppose that the FOX structure as described above is formed before any nodes receive any blocks. Though unlikely to happen in a deployed system, this brings to light the interesting problem of *last block collapse*, in which nodes begin dropping out of the system when they finish, leaving the remaining nodes with no help to finish their downloads. As presented thus far, FOX is vulnerable to last block collapse; when each R_i gets its final block, it can disregard its respective T_i and instead trade the final block with the other R_j's. In doing so, each R_i saves the time it would take to propagate their final blocks down the tree, resulting in a download that finishes $O(\log_k N)$ steps faster. Note that they could not do this sooner than on the last block, since their successors could detect their defection in one time step and invoke a global punishment (e.g., rate-limiting).

Last-block collapse arises in FOX because internal nodes do not have vested interest in their descendants near the end of the download. We enforce participation by augmenting our block exchange with a novel encryption algorithm. At the end of the algorithm, both internal nodes and leaves have information that the other needs to complete its download. The problem of vested interest is therefore reduced to the exchange of these final pieces of information.

Block Encryption Algorithm The specific algorithm of encryption and dissemination is presented in Algorithm 8, where $[b]_K$ denotes block b encrypted with key K and bounds on t_0 are given in Section 6.2.3.

To summarize, the internal nodes (i.e., $\cup_i(T_i \backslash L_i)$) use a shared key to encrypt the final blocks of the file. The leaves, unable to decrypt them, trade these encrypted blocks during

Algorithm 8 Block encryption and dissemination.

```
1: The $R_i$'s agree on a public-private key pair, $(K_{pub}, K_{priv})$.
2: Simultaneously, $\forall i, \forall v \in L_i$, $v$ chooses its own (i.e., not shared with any other nodes)
   public-private key pair, $(K'_{pub}, K'_{priv})$.
3: The $R_i$'s multicast $K_{priv}$ to their respective $T_i$'s, but only to the internal nodes and
   leaves with no assigned ancestor.
4: **for** steps $t = t_0$ to $(\frac{B}{k} - k)$, in pipeline, **do**
5:     $R_i$ computes $[b_{kt+i}]_{K_{pub}}$.
6:     The internal nodes of $T_i$ multicast $[b_{kt+i}]_{K_{pub}}$ to all nodes in their tree.
7:     The leaves perform their exchange with the encrypted blocks, that is, they trade
       their $[b_{kt+i}]_{K_{pub}}$ for other leaves' $[b_{kt+j}]_{K_{pub}}$.
8:     In parallel, $\forall v \in \cup_i L_i$, $v$ uses its $K'_{pub}$ to calculate $[[b_{kt+j}]_{K_{pub}}]_{K'_{pub}}$ where $b_{kt+j}$ is
       the block $v$ is assigned to send to its ancestor.
9:     The leaf $v$ sends this doubly-encrypted block up to its assigned ancestor.
10: **end for**
```

their leaf exchanges. When sending blocks to their ancestors, each leaf uses its own key to encrypt the already-encrypted blocks. Thus, upon completion, internal nodes need some leaves' K'_{priv} and each leaf needs K_{priv}. These are the final pieces of information mentioned above. Observe that the algorithm does not require a PKI; the public-private key pairs can be generated on a per-file basis by the peers themselves.

Key Exchange After running the algorithm in Algorithm 8, each leaf has its own K'_{priv} but needs K_{priv} and that each internal node has K_{priv} but needs its appropriate set of $(k-1)$ different K'_{priv}'s. All that remains is the key exchanges. Each leaf is involved in one key exchange with its assigned ancestor, and each internal node is involved in $(k-1)$ exchanges. The simplest method for two peers to exchange keys is to alternate sending one bit of the key without letting the other get ahead by more than one bit. To ensure that no leaves get ahead of others, the internal nodes do not reveal a new bit until all $(k-1)$ of the leaves reveal their corresponding bit, resulting in $O(kM)$ total steps where M is the size of the keys in bits. In the worst case, one of the nodes in the exchange could get the final bit and leave the system (as they are, by that time, finished downloading). Thus, some nodes may finish their download at most one *bit* ahead of others, but the remaining node need only try both values. In essence, all participants finish their downloads simultaneously. A more advanced simultaneous key exchange protocol is presented in [107] which requires more communication than just the M bits of the keys but, during the exchange, verifies that the keys are correct.

The question remains: for how many time steps should the encryption process take place, i.e., what is the desired value of t_0 in Algorithm 8? If it were just for the final block ($t_0 = B/k$), for example, then the R_i's could simply exchange their final blocks instead of sending down the tree. If it were for less than $\log_k N$ steps ($t_0 > B/k - \log_k N$), the root nodes could exchange their final set of blocks with one another in less time than it would take for their blocks to reach the leaves. More generally, t_0 must be large enough so that, after receiving the last block from the server, the root nodes cannot trade with one another faster than it would take to complete the protocol. Thus, we require that blocks be encrypted for at least $\log_k N + kM$ steps ($t_0 \leq B/k - \log_k N - kM$), the number of steps from the time the R_i's get the block to the time the download is complete. Observe that, regardless of the value of t_0, only a single agreement at the R_i's and a single round of key exchanges are necessary. In the event that a node could fail, it may be helpful to use a weak encryption method, in which the time to crack the private key is less than the time to re-download the blocks but greater than the time to perform the key exchange itself.

In Section 6.2.6, we present one implementable protocol of our system, in which we place no restriction on when nodes can join or depart. However, it is likely that arrivals would be staggered enough to avoid last block collapse altogether.

6.2.4 Incentives for Cooperation

To ensure proper cooperation from each peer, we incorporate incentive mechanisms into the FOX structure. In this section, we propose two general types of incentive: localized threats, which provide incentive to forward in a TFT-like manner, and global threats which handle extreme cases such as collusion. Since FOX is intended to be a provably fair, *deployable* solution, we aim for incentive mechanisms that induce subgame perfect Nash equilibria.

Localized Threats

In addition to filling the pipe, the connections between peers described in Section 6.2.3 have two important properties. One, they exhibit a give-and-take symmetry; each of peer p's outgoing edges has a corresponding incoming edge which "returns the favor." Two, the connections are stable; as opposed to systems such as BitTorrent or Slurpie, a node maintains its set of neighbors for the entire download.

Due to these two properties, when a peer p fails to forward blocks to its downstream, it is in effect refusing to send blocks to nodes in its upstream. By not providing its upstream nodes with blocks to trade, p's upstream will have no blocks to send down to p. The repercussions from defecting are felt quickly; within $O(\log_k N)$ steps, the "bubble" p introduced into its downstream of blocks will be experienced along p's upstream. Other file swarming systems have no such guarantees, and a defection may not adversely affect p at all.

Global Threats

There are instances where localized, TFT-like threats are not sufficient to overcome selfish behavior. For instance, nodes close to the server in the FOX structure could collude and isolate nodes further from the server from receiving any blocks. The isolated nodes need some means of providing disincentive to such behavior; here, we consider two different mechanisms: grim triggers and rate-limiting at the server.

Grim Triggers Consider many peers competing for throughput at a server. In general, servers have some threshold for the amount of contention they can withstand, after which throughput tends toward zero for each connecting peer. We approximate this behavior in a simplistic way with the k-meltdown model, as follows:

> *As the number of connections to the server increases to some number, k, the throughput to each node stays constant. Once more than k clients connect, the server "melts down," that is, its total outgoing throughput drops immediately to zero.*

We assume the clients also have a meltdown point and that it is the same k. This is a rather strong assumption, as it implies homogeneous resources at the clients. However, it allows for a strong punishment mechanism; when p detects that there is some defection that a simple localized punishment has not remedied, p can connect to the server in an attempt to download more blocks. If the server does not already have k connections, then p will get blocks from the server. Otherwise, if the server already has k connections, then p should have been receiving blocks, and in this case will cause a meltdown at the server, cancelling the download for *all* peers.

Pulling this grim trigger of a meltdown is not a viable threat; when actually faced with the decision, why would a peer cancel the download forever given the alternative of waiting for the free-rider to finish downloading? It is precisely for this reason that any protocol with such a grim trigger is not a subgame perfect Nash equilibrium (SPNE). Protocols that exhibit SPNE are more desirable; it ensures that each peer will truly follow the protocol at each step in the repeated game.

Rate-Limiting Because grim triggers do not yield SPNE, it is not viable for a deployable solution To obtain a SPNE protocol, we introduce another mechanism: rate limiting at the server. When a peer p detects that some other peer has defected, and p is not able to resolve this locally, p may request the server to send at a limited throughput. The task of the server is to slow the download down for some duration of time, τ, such that the gain that a defector could have received is offset by this loss in throughput.

Let us now formalize this incentive mechanism. Suppose that any peer's defection can be detected in time $\Delta \in O(\log_k N)$ from when they defect, and that $G(\Delta)$ is the most gain that a defector can achieve as a function of the time it takes to get caught. The server's task is to induce a loss in utility, $L(\tau, \lambda)$, by limiting its throughput to $1/\lambda$, $\lambda \geq 1$, for some time τ, so that the resulting utility, $G(\Delta) - L(\tau, \lambda) < 0$, that is, the resulting utility is less than if the defector had simply followed the protocol all along.

The gain $G(\Delta)$ depends on the defection, but let us assume the worst case scenario: that the defector obtains all $k(\log_k N + 2)$ pending (i.e., "on-the-wire") blocks in one time unit, and then obtains each new k blocks introduced in each subsequent $\Delta - 1$ steps:

$$G(\Delta) \quad \leq \quad k((\log_k N + 2) + (\Delta - 1)) \quad = \quad k(\log_k N + \Delta + 1)$$

Hence we simply require that $L(\tau, \lambda) > k(\log_k N + \Delta + 1)$. The amount of blocks a node would get under a rate-limit factor of $1/\lambda$ for time τ is $k\tau/\lambda$, whereas a fully cooperative node would have received $k\tau$. Combining this, we get

$$\begin{aligned} L(\tau, \lambda) &> G(\Delta) \\ k\tau - \frac{k\tau}{\lambda} &> k(\log_k N + \Delta + 1) \\ \tau \cdot \frac{\lambda - 1}{\lambda} &> \log_k N + \Delta + 1 \end{aligned}$$

This shows that, regardless of the rate reduction ($\lambda \geq 1$), the punishment need only take place for $\Theta(\log_k N + \Delta)$ rounds for even the worst-case scenario.

The amount of time added to the system-wide download by both the defector and the punishment is $\Delta + \tau \cdot (\frac{\lambda-1}{\lambda})$. Let us define the social good as the amount of time for *all* peers to download the file, and let OPT denote the optimal time in doing so. Then the fractional loss in social good by this defection and punishment is

$$1 - \frac{OPT}{OPT + \Delta + \left(\tau \cdot \frac{\lambda-1}{\lambda}\right)}$$

As we will see in Section 6.2.5, this is dominated by OPT; the punishment, though a loss in efficiency, is a small price to pay to discourage nodes from further free-riding.

6.2.5 Analysis of the FOX Structure

The hypothetical system described above requires little state, provides asymptotically optimal performance, and provides incentive for participants to behave in accordance with the protocol.

Per-Peer State

Recall that each node stores k incoming connections, at most k outgoing connections, and one public-private key pair: (K_{pub}, K_{priv}) for all $v \in T_i \backslash L_i$ and (K'_{pub}, K'_{priv}) for all $v \in L_i$. Thus, the amount of state per peer is $O(k)$.

Download Time

There are two main properties of FOX to note: that each node receives the file in the same amount of time and that this time is nearly optimal. In our analysis, we will assume that each node has the same incoming and outgoing bandwidth resources. Without further loss of generality, we assume that each node can send out 1 block to each of its k links per time step. Thus, the server can send out only k different blocks each round, giving us the trivial lower bound of $\Omega(B/k)$ rounds until all nodes finish downloading. This lower bound does not take into account the time to propagate data throughout the system.

Theorem 1. *The number of steps until all FOX peers have completed the download is* $O(\log_k N + \frac{B}{k})$.

Proof. Initially filling the pipe takes $O(\log_k N)$ steps: $\log_k N$ to get down the multicast tree, 1 to swap blocks amongst leaves, and 1 to send back up to ancestors. Since no links are used more than once in these, we can pipeline the operation, requiring precisely $\left(\frac{B}{k} - 1\right)$ additional rounds. The key exchange at the end of the download consists of $O(kM)$ steps, each requiring 1 bit to be sent. If β is the block size sent each round during the file transfer then key exchange takes $O(\frac{kM}{k\beta}) = O(\frac{M}{\beta})$ time. It is reasonable to assume that the key size, M, is significantly smaller than the file's block size, β, hence $O(\frac{M}{\beta}) = O(1)$. Thus, the time for all FOX peers to complete downloading is $O(\log_k N + \frac{B}{k})$. $\square$

Incentives

Here, we discuss some of the incentives that FOX provides to its participants. Combined, these incentives indicate that the FOX protocol in is a stable equilibrium.

Sending Correct Data As discussed in Section 6.2.2, if there is no means of verifying an individual block's integrity, then using tit-for-tat in a file swarming system may give peers the incentive to generate garbage blocks, effectively forging the system's currency. In short, peers have the incentive to send data as soon as possible, garbage or not. For the remainder of this discussion, we assume that each peer perceives the task of sending data independent of the data itself; if it has a correct block to send, it will not go through the trouble of generating a garbage block to send in its stead.

Contrary to other tit-for-tat file swarming systems, we structure the flow of data in such a way that peers do not benefit from sending blocks before they have the correct data to send.

Claim 1. *Even without a means of verifying a block's integrity, no peer has the incentive to send forged blocks.*

Proof. Once peer p has begun receiving blocks, it has no shortage of correct data to send, so consider the case when p has yet to receive data. Suppose $p \in L_i$ for some i. Sending a garbage block to some $q \in L_j, j \neq i$, yields no reward, as q will not have a real block to trade until the same time p does. Similarly, sending forged data to an ancestor has no positive effect — receiving blocks from leaves does not make internal nodes change their behavior — it simply keeps them from invoking one of the global punishments from Section 6.2.4. Now consider $p \in T_i \backslash L_i$ for some i. Sending a garbage block to its children will give the leaves of the subtree rooted at p a block they can trade with leaves from other T_j's. However, the leaves in L_j will not have data to send in return until the first round of multicast is complete. Thus, peer p will not receive data in response to its garbage block any sooner than if it had followed the protocol exactly, and hence yields no benefit from such a deviation. □

Maintaining the Structure Unlike current file swarming systems, we provide a specific topology instead of allowing peers to arbitrarily match up. Here, we consider the desirability of our structure, in particular the underlying multicast tree. One could envision that a peer, attempting to minimize the amount of data it has to send, would maintain a single child. With the following, we show that internal nodes have the incentive to give as much to the system as possible.

Claim 2. *Every peer has the incentive to ensure that the underlying multicast tree is a balanced, full, k-ary tree.*

Proof. Observe that, for a tree of height H, the runtime in Theorem 1 is $O(H + \frac{B}{k})$, thus it is within every peer's best interest to minimize H. H is minimized when the tree is balanced and, in the k-meltdown model, when each internal node maintains as close to k children as possible.

Next, consider the leaves' outgoing edges. If not maintained precisely as described in Section 6.2.3, then there will be a peer that does not have k incoming edges and will therefore invoke a global punishment (e.g., rate-limiting). Thus, every peer has the incentive to maintain the structure. □

Maintaining Position The structure itself is desired by the participants, but that does not immediately imply that they will be willing to do their part. Instead, we must ask: are there certain positions within the structure that are preferable? If this were the case, the stability of the system would be in question as nodes would be vying for these better positions.

Observe, however, that since all nodes finish downloading the file at the same time, they have no incentive to change their position. This follows from our assumption that utility equals the time to download the entire file. If, however, a node perceived benefit in minimizing the time to download intermediate blocks, it may have the incentive to become a leaf (since they receive $(k-1)$ of the k blocks per time step earlier than the internal nodes). In this case, we could extend the encryption procedure to cover every step of the download, removing whatever use the nodes could have had for receiving the intermediate blocks sooner. This would not increase the number of messages; in Algorithm 8, only t_0 would change. It would, however, increase the amount of work performed at each R_i and each leaf, which is why we minimize the number of encryptions to $(\log_k N + kM)$.

Additional Threats A final concern is whether participants in FOX can use the k-meltdown model as a means of threatening other nodes into providing more resources than is strictly required by FOX. Note that, in any such threat, the threatening node must offer the nodes a means of completing the download. If instead they were offered nothing, then the meltdown would be an equivalent outcome and the threat would be baseless. A simple solution to this is for the threatened nodes to counter with another threat; any selfish activity will result in a meltdown.

6.2.6 A Deployable Protocol for FOX

FOX, as presented in Sections 6.2.3 and 6.2.4, is not immediately applicable to downloading on the Internet. A FOX protocol must take the following issues into account:

- In a deployed system, peers must be able to join the system at any time, which precludes any strict requirement on the number of nodes.
- To provide the incentive to not free-ride, the protocol must provide a means by which to restrict the locations in the tree where a node can reside.
- Since every node in the FOX topology plays a crucial role in system-wide data dissemination, the protocol must quickly recover from node failures. This must be done in such a way that failures or departures do not invoke any of the global punishments from Section 6.2.4.
- If, for some reason, a peer misses a block in the middle of the download, the protocol must provide a reasonable way for the peer to finish the download.
- Lastly, if possible, the protocol should allow for heterogeneous resources at the peers.

We employ two general approaches to address these points while still providing the same incentives and near-optimal download times.

- We introduce *leaf communities* as a means of allowing for any number of participants. Leaf communities are, logically, a single node in the idealized structure, but in fact consist of multiple peers.
- To ensure progress regardless of missed blocks or join times, we use source coding.

In this section, we present these approaches in detail.

Leaf Communities

The fundamental difference between our implementation and the FOX topology in Section 6.2.3 is the concept of leaf communities (Figure 6.3). We use the same structure as in the idealized case, but each leaf node is virtualized to a community of nodes, thereby improving FOX's flexibility. Leaf communities serve two roles: (1) they provide a means of abstracting the number of peers in the system from the FOX topology, and (2) they act as "backup pools" for internal nodes, in that leaf community members are responsible for replacing their ancestors if they leave the system.

Each community consists of one *community leader*, as many as $(k-1)$ distinct *community recipients*, and potentially other peers. The leader receives the native block from the community's parent (internal) node and sends it to (a) one node within its community and (b) one recipient in each of the $(k-1)$ leaf communities with which it shares. Community recipients receive blocks from other communities' leaders and propagate them within the community. The other peers in the community aid in propagation between community leaders, recipients, and handle the community's ancestor pointer. Thus, each

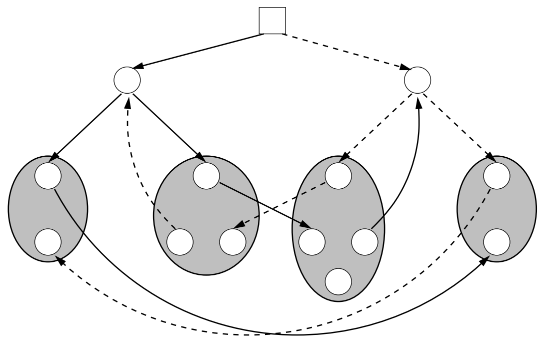


Figure 6.3: Leaf communities (shaded ellipses) may contain multiple peers. Notice that each leaf community has the same inter-community links as the example in Figure 6.2. Intra-community links are not shown.

leaf community receives and is responsible for the same incoming and outgoing links as the idealized structure: one ancestor link and $(k-1)$ lateral links. As long as there is at least one node in the community, it can uphold its communal responsibility of k total incoming and k total outgoing connections.

Additional work and control messages to propagate blocks within a given community are handled completely within that community. As we will show in Section 6.2.7, communities have a low expected size that is logarithmic in N. We therefore allow peers to store a complete list of each member of its community and to broadcast this list within the community. Once each community member has the same list of nodes, they each run a deterministic algorithm (Algorithm 9) locally to determine which blocks it is supposed to send to which nodes. Since it outputs a connected, directed graph with each node having equal in-degree and out-degree of k, this algorithm constructs a directed, Eulerian graph within the leaf community. This algorithm is similar to the optimal schedule described in [60], except nodes in FOX require bounded out degree by the k-meltdown model.

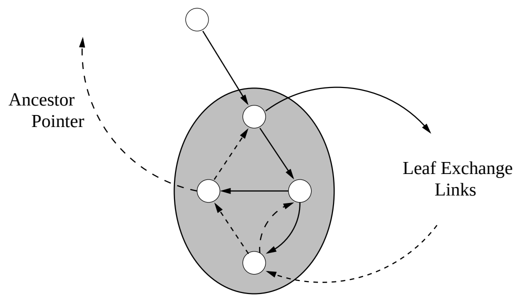


Figure 6.4: Intra-community links for a community of size 4 when $k = 2$. Note that the community leader is responsible for the community's outgoing links but other members are responsible for propagating blocks from incoming links.

By construction, increased community size results in greater resilience to node failure. This comes at the cost of increased state for community members.

Algorithm 9 calcSchedule(subtree id , nodes $p_1 \ldots p_c$)

```
    {Output a set of intra-community schedules $\bar{S} = \{S_1, \ldots, S_s\}$ such that $S_i(p,b)=q$
    denotes node $p$ always sends blocks from subtree $b$ to node $q$. Also output recv$_i$ , $i =
    0 \ldots k-1$, the community recipient that receives the blocks from the matching com-
    munity leader of the $i$th subtree}
    for $i \leftarrow id \ldots id + k - 1 (mod\ k)$ do
2: 1:  Output S($p_1$ , $id$)=recv$_i$ {Send native block to other matching community recipi-
       ents }
3:  end for
4:  for $b \leftarrow id \ldots id + k - 1 (mod\ k)$ do
5:     $p_i \leftarrow$ node with least schedules, tie break on lowest id
6:     Output $p_i$ is recv$_b$ for this community
7:     $\hat{i} \leftarrow i$
8:     for $j \leftarrow \hat{i} + 1 \ldots \hat{i} + c - 2 (mod\ c)$ do
9:        while $p_{\hat{i}}$ has less than $k$ schedules do
10:          Output S($p_{\hat{i}}$ , $b$)=$p_j$
11:       end while
12:       $\hat{i} \leftarrow \hat{i} + 1$
13:    end for
14: end for
```

Restricted Node Placement

As mentioned before, if peers are allowed to trade blocks with any other peers, there arises the possibility for nodes to join the system, free-ride, leave, and rejoin at a new position in the topology, effectively whitewashing without changing IDs. To avoid this problem, we could require nodes to share blacklists, but that in turn introduces additional overhead and state requirements. Instead, we restrict the connections that a given peer can make within the FOX topology, thereby reducing the number of nodes it could potentially exploit. Our solution makes use of the following observation found also in Pastry [105]: a base-k number defines a virtual path through a degree-k tree, the i^{th} digit denoting which of the k potential paths down the tree to take from a node at level i. Each peer locally computes its ID using a universal hash function that is known to all peers in the system (e.g., SHA-1

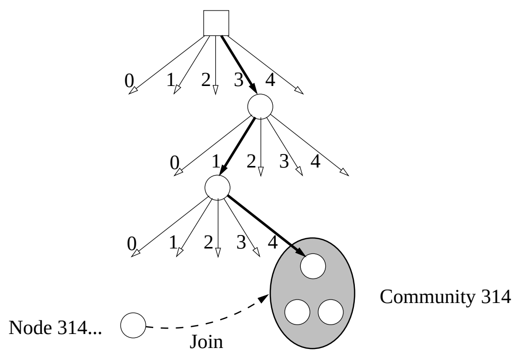


Figure 6.5: Example of placing a node with ID prefix 314 into a structure of height 3 with $k = 5$.

on its IP address). We require that each node be on this virtual path, as determined by its ID. In doing so, each peer can be mapped to exactly one leaf community at any given time, or one of the $(\log_k N - 1)$ internal nodes on the path from the leaf community to the server, given N peers in the system. For example, in Figure 6.5, when the structure is of height 3 and $k = 5$, a node with an ID prefix of 314 must join community 314, defined by the path from the server.

Peers will not allow others to join at parts of the tree that do not match its virtual path; otherwise, there is nothing binding the new peer to them, so it can free-ride and join somewhere else. With this in place, each peer p will have only $O(\log_k N)$ other peers willing to let it join (those on the virtual path defined by p's ID). Peers therefore do not have the incentive to free-ride, else they risk upsetting this small number of peers and getting disconnected from the tree. In our protocol, when a node joins, only its leaf community and the communities that trade with it need to change their connections. Note that, since each peer has precisely one community that it can join when it enters the system, peers gain nothing from free-riding, leaving the system, and returning under the same name.

Whitewashing (i.e., joining under a different name) is also not a viable strategy. To ensure that it joins a different part of the FOX topology, a greedy peer would have to generate a new ID. A simple balls-and-bins analysis shows that, to be able to go to all of the k^H leaf communities, a node would have to generate roughly $k^H(\ln k^H + 1)$ unique IDs. Lastly, at each join/departure, the node would not be able to exploit the system any more than a failed node would, else it would cause a global punishment (e.g., rate limiting) to be invoked. Given this, nodes do not have incentive to whitewash, assuming that generating a new ID takes even a nominal amount of time.

There is a trade-off between restricting node placement and maintaining network locality. We currently make no guarantees that a peer p be close, in terms of the underlying network, to any of the nodes on its virtual path. Our system may therefore result in potentially much worse latency than optimal (e.g., each node in New York could be connecting only to nodes in London rather than to each other). It is our observation that this trade-off is indicative of a much more general class of trade-offs between fairness and efficiency. We leave the study of such trade-offs, including that of network locality, for future work.

Lookups

There are several instances in our protocol when a peer must be able to locate a peer residing in a given community, the most obvious being when the node joins. Fortunately, there is a significant amount of structure already present in the FOX topology that makes this simple. We provide a recursive algorithm below. For ease of exposition, if p is a peer, let $p.parent$ denote p's parent, $p.child[i]$ denote p's i^{th} child, $p.id$ be p's ID, and for an

ID t, let $t[i]$ be t's i^{th} digit.[1]

Algorithm 10 lookup(Peer $curr$, ID $target$)

```
1:  if $curr.id = target$
2:      return $curr$
3:  Let $j$ be the smallest number such that $curr[j] \neq target[j]$.
4:  if $curr$ is in a leaf community
5:      Let $l$ be a leaf community member from tree $T_{target[0]}$ with which this leaf community exchanges blocks.
6:      return lookup($l, target$)
7:  else if $j <$ $curr$'s level
8:      return lookup($curr.parent, target$)
9:  else
10:     Let $m$ be $curr$'s level in the tree.
11:     return lookup($curr.child[target[m]], target$)
```

Joining Leaf Communities

Peers join the system by being added to the leaf community determined by the peer's ID. When a peer wishes to join the system, it first contacts a topology server. The topology server keeps k small caches (one for each of the different subtrees below the file server) of the most recent nodes to query it. When a node contacts it, the topology server returns a small list of nodes from the cache corresponding to the new node's intended subtree (as determined by the first digit of its ID). A list, as opposed to a single node, is returned so that, with high probability, the new node is supplied with at least one node that is still in the system, if any.

By our assumption on the topology server, if none of the nodes returned to the joining peer are still in the system (either because they failed or because they have completed their download), this means that it is the only node in its respective subtree and there-

[1]Note that Algorithm 10 can put unbalanced load on nodes near the top of the tree, especially at the server. To address this, we could make the FOX structure into a small world network by adding a small number of random "far" edges in addition to each peer's k "close" edges. Greedy routing in small world networks is both efficient and load-balanced [61]. We reserve such considerations as future work.

fore connects to the file server. Otherwise, it contacts one of the peers from the list and performs a lookup of its own ID. The lookup will terminate at the leader of the unique leaf community C whose ID which matches the longest prefix, which is precisely the leaf community on the new node's virtual path. Observe that, since peers' IDs are obtained from a universal hash function, each leaf community is equally likely to have a new node mapped to it.

When the lookup returns, the new node issues a join request message to the node it found in community C. If the request is accepted, C's community leader updates its local list of all community members to include the new node and then broadcasts this list to all of the members of the community. Each member (including the new node) uses the new list to determine which connections it should make. Once modified, C may have a new community leader and new community recipients. The new leader of C informs the leader of each community that sends blocks to C of the corresponding recipient. If the community leader does not accept the request, the new node can try to join again or it can invoke a global punishment, therefore providing peers with the incentive to accept new nodes into the system.

Recovering from Departures

Unlike previous file swarming systems, we do not require peers to perform any actions when they complete their download; they may (and are assumed to) simply leave the system as soon as they obtain the file. In our system, it therefore suffices to handle node departures and node failures in precisely the same fashion. When a node within a leaf community fails, the other community members that detect the failure inform the rest of the community with death certificates of the form “peer p is no longer in the community.” Upon receipt, each community member removes p from its membership list and locally recomputes the connections it should make. Observe that peers do not have the incentive to send death certificates when they are not true, otherwise p will be cut off from the community and will invoke a system-wide punishment.

By provisioning for each leaf community to have extra peers, we can effectively provide backup for internal nodes. When an ancestor a departs, the $(k-1)$ peers in leaf communities that had ancestor pointers to a detect it within roughly one RTT.

One such leaf node is responsible for replacing the ancestor. The replacing node, r, can leave its community without informing other community members; it will be handled as a node failure. Since r is replacing an internal node, it must discover the children it is adopting. There are several useful heuristics for this; each internal node could, for instance, keep state on its siblings and each leaf community could keep state on the $O(\log_k N)$ ancestors on its path to the root. In doing so, r would know one of its adopted children (the one on the path between r's community and the root), so r could ask it for its siblings. Handling the simultaneous departure of a chain of internal nodes is a non-trivial problem. A study into the trade-offs between overhead and fast recovery in the FOX topology is an avenue of future work.

Resizing the Structure

To scale with widely varying N, it is necessary that the overlay structure itself be dynamically resized. Otherwise, leaf communities could grow unbounded, requiring community members to maintain too much state. To address this, we provide means for growing the

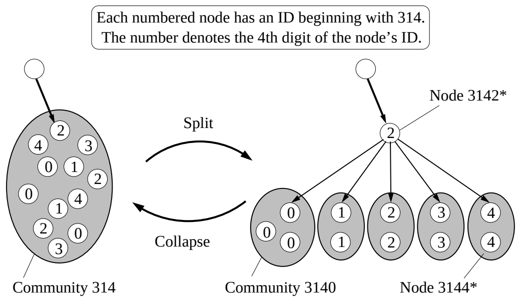

Each numbered node has an ID beginning with 314.
The number denotes the 4th digit of the node's ID.
Split
Collapse
Node 3142*
Community 314
Community 3140
Node 3144*

Figure 6.6: Example of splitting to a height-4 tree and collapsing to a height-3 tree.

structure with *splits* and shrinking the structure with *collapses*. Both of these operations maintain the structure's requirements, namely that it form a full tree of some height and that each leaf community have at least ℓ nodes.

Splits A leaf community splits by promoting its community leader to an internal node and partitioning its remaining members into k communities. The internal node takes these k communities on as its children and, in turn, $k-1$ of the communities send blocks back up to it via ancestor pointers. The k^{th} community maintains the responsibility for the original community's ancestor pointer. To maintain the properties of the overlay structure, each new community must contain only peers whose ID matches to $H+1$ digits; if the $(H+1)^{st}$ digit peer p's ID is i, then, if it does not become the internal node, it goes to the i^{th} new leaf community. For example, in Figure 6.6, nodes with ID 3144* in community 314 become members of community 3144 (the 5^{th} subtree of the node that became the internal node, 3142).

To preserve a full tree, all leaf nodes must split at the same time. Determining when to split involves feedback from the server. Periodically, each leaf community informs its parent (internal) node of whether or not it can split: that is, whether or not there are enough nodes whose $(H+1)^{st}$ digit is 0, 1, and so on. To reduce the probability of communities quickly splitting and collapsing repeatedly, community leaders do not report that they can split unless they have more than the minimum number required to sustain a community (e.g., only when they have three times this amount; we provide precise bounds in Section 6.2.7). Upon receiving such a report from each of its k children, the internal node informs its parent of whether or not all of its children are able to split. This continues up the tree, eventually resulting in the server learning whether every community is able to split.

If every community can split, the server multicasts a split message down the tree. When a leaf community receives a split message, each member locally calculates its position in the tree as well as the connections to make within its new community (or, if it becomes an internal node, which community leaders to connect to). Observe that, before the split message, every member in the community maintained state about every other member, so such computations can be purely local operations. The latency in reporting on splits and sending a split message down the tree is $O(\log_k N)$, so it is unlikely that a given community can no longer split once it receives a split message.

When a node joins the network, they become a member of the leaf community that matches the longest prefix of the node's ID. Since node IDs are generated with a universal hash function, each leaf community has equal probability of receiving a joining node. In Section 6.2.7, we bound the expected size of a leaf community before it splits.

Collapses When any leaf community is deemed to be too small, a collapse occurs. In addition sending up messages indicating whether it can split, each leaf community informs its parent (internal) node of whether it must collapse. Where split messages are logically and'ed as they propagate up the tree, collapse messages are or'ed; whenever a single community needs to collapse, the server will tell all communities to collapse.

Leaf communities combine by sharing their information with their parent, who will join the collapsed community as the leader (since it already has the connection to who will be the community's parent). The new leader concatenates all of the membership lists it has received and sends them to its children. As with splits, each member computes its position in the new leaf community and required connections.

Source Coding

In the idealized version of our protocol, all nodes join at the same time and do not fail or falter in delivering blocks to their prescribed peers. As such, no mechanism for keeping track of what blocks a node has or needs is necessary. However, to support realistic environments, we must allow for *trembles*: actions that are not in accordance with a peer's chosen strategy. In particular, if for some reason a peer p misses a single block, we ought not require p to (a) go to the server, (b) perform additional out of band work to obtain the block from someone else, or (c) participate in multiple downloads of the same file simply to get the few blocks it missed.

Instead, we employ source coding [18]. In lieu of sending out a pre-defined set of blocks, the server generates and sends only blocks that it has never sent before (or, at least, not for a long period of time), based on the original file. Downloaders retrieve any subset of some pre-determined size of these blocks (at least as many as the original file) and performs additional, local computation to "combine" the blocks into the desired file. Various source coding techniques exist with various trade-offs between the number of additional blocks to download and the amount of necessary post-processing. For instance, Tornado codes require less computation than Reed-Solomon codes, but at a cost of requiring more blocks to be downloaded. We make no explicit requirements on which type of coding to use, simply that peers need not monitor or inform others of what blocks it is missing. As a result, our protocol handles trembles and departures without requiring additional control overhead.

6.2.7 Analysis

Leaf Community Size Our protocol does not provide any "active" load balancing for leaf communities; each has equal probability of receiving a joining node and no leaves split until each of them has enough nodes to do so. This raises the question: how big can a given leaf community become before each leaf community is able to split?

Consider the case where the virtual structure is of height H and there are thus k^H leaf communities. Suppose each leaf community must have at least ℓ nodes to safely handle node departures. Then it is necessary, but not sufficient, that each leaf have $1 + k\ell$ nodes before it can split; after splitting, one node must become an internal node and each newly formed leaf must have at least ℓ nodes but, by our restriction on node placement, each member must have the same $(H + 1)^{st}$ digit. We show in Appendix A that, with high probability, the maximum number of nodes in any community is small, even with very large N.

Per-Peer State In addition to the amount of state stored in the structure from Section 6.2.3, nodes in leaf communities have to maintain the list of as many as M other community members, where M is small, even with large N (see Appendix A), resulting in $O(k + M)$ state.

Download Time Downloading a file involves two general steps: joining and maintaining the pipe during splits and collapses. The time to join is dominated by the time it takes to look up one's community, which, for a tree of height H, takes time $O(H)$, or $O(\log_k N)$. It is worth noting that this has small constants; if the minimum number of nodes in a community is at least 2, then roughly half of the peers do not contribute to the height of the tree. Once joined, the peer's pipe is full so, if no splits or collapses occur, it takes additional time $O(B/k)$ to download. Whenever a split or collapse occurs, each peer locally recomputes its intra-community connections (or becomes an internal node). The cost of this computation is negligible to download times. If nodes can fill the pipe of a new connection quickly (e.g., the time to open the TCP window to the size of the blocks is small), then additional time spent splitting and collapsing is also negligible. The overall download time for our presented protocol therefore remains $O(\log_k N + \frac{B}{k})$.

6.2.8 Simulation Results

In order to validate the system design, we implement the protocol in a message level discrete event simulator. The simulator models the node joins, data transfer, leaf clusters, and node recovery portions of the FOX structure, but not the dynamic splitting and joining of clusters. In absence of the splitting and merging protocols, the simulation still accurately models the system's steady state, that is, when joins and departures occur at roughly equal rates, thereby still providing insight into its performance. We present the results from the simulator below.

In all experiments, all nodes in the system have symmetric bandwidth of 1.5Mbps and 15ms latency, connected in a star topology. Each host downloads a 100MB file, in chunks of size 1460 bytes to simulate the payload of a TCP packet on a 1500 MTU link. In all cases, the absolute lower bound on the file transfer time is 533 seconds or about 9 minutes, and each node leaves the system as soon as it finishes downloading the file. Except where otherwise noted, we fix $k = 5$ for all experiments.

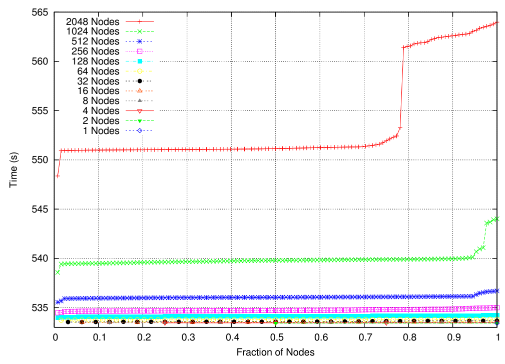


Figure 6.7: CDF of download times in seconds for $n = 1, 2, 4, \ldots, 2048$ peers concurrently download a 100MB file with $k = 5$. The optimal time to finish is 533 seconds shown as the x-axis.

Figure 6.7 shows the effect of flash crowds on the system. All n nodes, $n = 1, 2, \ldots, 2048$, join the system at time zero and leave immediately upon completion. The x-axis marks 533 seconds, the absolute lower bound on the transfer time. As shown in Figure 6.7, the transfer times for $n = 1 \ldots 128$ nodes are practically indistinguishable, i.e., they are independent of the system size with less than 1% overhead with respect to the lower bound. However, for larger n, the overhead from the large leaf community size starts to dominate the transfer time, resulting in 9% overhead in the worst case. However, we expect this worst case to be avoided in the full protocol because node splitting operations prevent communities from growing too big (AppendixA). The large gap in completion times for $n = 2048$ case around 79% represents the time to reconfigure and rebuild the tree after a

k	Avg.	σ
4	536.4	0.4
5	536.0	0.1
8	536.8	0.2
16	543.4	1.0
32	578.1	5.2
64	581.1	7.2

Table 6.1: Average completion times (seconds) with standard deviations for 512 nodes starting concurrently with variable k.

mass exodus of nodes that have finished downloading the file.

Next, we examine the parameter k's effect on performance. In Table 6.1, we fix the number of nodes to 512, test with $k = 4, 8, 16, 64$, and present the average completion times with standard deviations. As shown, if k is small, the transfer time is not effected by the out degree. However, as k grows, there is progressively more work that each leaf community must do to maintain its k pointers, so for large k, this extra overhead starts to affect the transfer time.

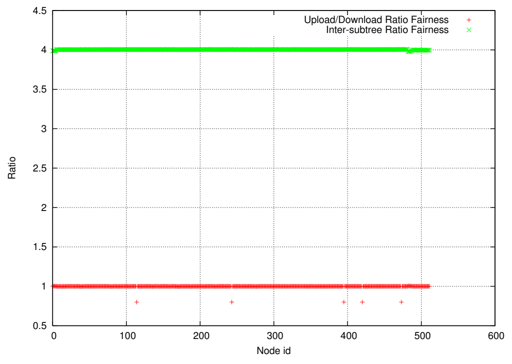


Figure 6.8: Ratio of blocks uploaded to downloaded, and non-native blocks to native blocks, $k = 5$, $n = 512$

Last, we demonstrate the fairness of the system in Figure 6.8. The top line is the ratio of blocks received from a node's $k - 1$ non-native subtrees to the number of blocks received from the native subtree. As expected, this ratio follows the line $y = k - 1 = 4$

very closely. The bottom line of Figure 6.8 is the ratio of uploaded blocks to downloaded blocks. Again, as predicted, this ratio very closely tracks the line $y = 1$, however note that exactly $k = 5$ nodes have sent slightly less then they have received. These are exactly the nodes that have the server as their ancestor (i.e., they are supposed to *upload* to the server) but do not need to send data on that one of their $k = 5$ links.

6.2.9 Discussion and Conclusions

FOX is intended to provide provable fairness in a way that can viably be deployed on today's Internet. By far, the largest obstacle in achieving this with the protocol we presented in this section is the use of system-wide punishments such as a grim trigger or, more reasonably, rate-limiting at the server. If we assume that it is difficult for peers to change their IDs (e.g., a mechanism such as those in [44] is in place), we may be able to remove global punishments (Section 6.2.4) altogether and simply handle cheaters the same way we handle node failures. Recall that, when a node fails, the system recovers by replacing it (if it was an internal node) or by simply recalculating links without it (if it was a leaf community member). If we are able to detect cheaters, then we can employ the same tactics; peers can simply recover from the Byzantine fault, store the cheater's IP address in a local blacklist, and not let the cheater rejoin the community. Currently, one cannot easily detect cheaters in FOX; if peer p does not receive a block from one of its neighbors when it should, it is not clear whether it was p's neighbor that cheated or someone further upstream. To handle this, FOX peers can send garbage data whenever they do not receive the real data they should have, thereby assuring their downstream peers that they are not cheating. As discussed in Section 6.2.5, rational players would not benefit from sending garbage data for any reason other than to prove to their downstream peers that they are not cheating.

Chapter 7

Truthfulness Pitfalls

In virtually all decentralized systems, peers maintain some *private information*: data or preferences that cannot be independently observed or verified by others. Examples of private information include how much a peer is willing to pay for someone to forward their packets like in Hoodnets, or how much bandwidth a peer is able to share, as in BitTorrent. One of the primary goals of economic mechanism design is to provide incentive to participants to truthfully report their private information [90]. We were able to achieve such *truth-telling* mechanisms in Hoodnets (Chapter 5) through the use of money. However, because money is typically not systems-compatible, we ask in this chapter whether or not truth-telling can be achieved in a decentralized system without the use of money.

Our focal system in answering this question is BitTorrent. We presented PropShare in Chapter 3, a trading mechanism that gives peers the incentive to provide as much bandwidth to the system as possible. PropShare effectively provides incentive for peers to truthfully report how much bandwidth they have; they are unable to over-report—that is, send more than their bandwidth cap—and under-reporting would increase their download times.

However, bandwidth is not the only piece of private information a BitTorrent peer maintains. Each peer, at any point in time, also maintains what pieces of the file they have downloaded. This information appears, at first glance, to be innocuous. Indeed, it has been widely believed in the literature that a BitTorrent peer cannot benefit from lying about how much of the file they have downloaded [111].

We demonstrate in this chapter that, counter-intuitively, BitTorrent peers have incentive to intelligently *under-report* what pieces of the file they have to their neighbors. Additionally, we propose an under-reporting strategy and show via implementation that it can lead to significantly improved download times.

Unfortunately, a systems-compatible solution to under-reporting does not appear to be possible, at least within the context of BitTorrent. We close this chapter with an overview of various solutions, and conclude that the infrastructure costs to ensure truthful mechanisms are too high for most systems.

7.1 Maximizing Interest in BitTorrent

In the auction in Section 3.4, a peer sends only to those in whom it is interested, and, equivalently, receives only from those interested in it. Clearly, a peer benefits from being interesting to as many of its interesting peers as possible. A peer's interest in one of its neighbors is based solely on what pieces that neighbor claims to have. In this section, we consider what piece revelation strategies a peer can employ to maximize the number of neighbors who are interested in it. That such strategies are feasible is counter-intuitive; it seems natural that peers should advertise all they have to offer to remain interesting. We show, however, that a peer can prolong interest by *under-reporting* what blocks it has.

A natural question follows: how successful can an under-reporting strategy be? Can a peer, for instance, obtain a piece monopoly? We show empirically that piece monopolies are infeasible. BitTorrent's rarest-first piece selection strategy is a viable deterrent to the style of piece *hoarding* that would be necessary to obtain a monopoly, even when a large number of leechers collude.

To the best of our knowledge, we are the first to consider non-trivial piece revelation strategies. BitThief's strategy is effectively to obtain as large a set of neighbors as possible and to reveal that it has no blocks of interest to any of its neighbors [79]. Shneidman et al. [111] suggest *over-stating* what blocks a peer has and uploading *garbage* blocks, but this is infeasible, as peers can easily detect and punish this defection. Other theoretical work assumes either an infinitely-sized file [124] or cooperative peers [83].

7.2 Leechers want all the attention

We begin our study of piece revelation strategies by considering what outcomes are preferable to leechers. In Figure 3.1, we show a leecher i's preferences over various scenarios, and denote preferences $\succeq_i$. Clearly, i prefers to have as many neighbors that are *interesting* to i to be *interested in* i as possible. This will result in more peers bidding at i, and thus faster download times. Thus, (a) $\succeq_i$ (c) $\succeq_i$ (e).

The degree to which leecher i's neighbors find *other peers* interesting affects i's future interest. If i's neighbors trade pieces that i has with one another, then i risks becoming uninteresting to its neighbors sooner than if i's neighbors *only* had interest in i. This gives us (b) $\succeq_i$ (d) $\succeq_i$ (f) from Figure 3.1.

Note that case (f) can only occur when i does not know at least one of j or k. To see this, suppose the converse: that j and k are interested in one another, that i has no interest in either, but that they are all neighbors with one another. Then j has some piece p_j that k does not have. Since i is not interested in j, i must have p_j. k would therefore be interested in i due at least to i having p_j, a contradiction.

Nonetheless, (e) $\succeq_i$ (f); i prefers the scenario in which *no* peers are interested in one another to the scenario in which *other* peers are progressing while i is not. This is because the sooner j and k finish downloading the file, the sooner they may leave the swarm, reducing the potential number of peers eventually bidding at i. This is an example of when selfish participants' preferences conflict with the social good.

In the remainder of this section, we consider strategies a leecher i can take to achieve preferable scenarios from Figure 3.1.

7.3 Forcing a block monopoly is infeasible

A tempting initial attempt at maximizing a peer's demand is to try to obtain a *block monopoly*. If p were the only peer in the system with some block b, then p could extort bandwidth from all other peers, giving out b to others at a rate so slow that p would be the only one making discernible progress toward completing the download. Of course, with a seeder, a single node obtaining a monopoly is virtually impossible.

We consider here the feasibility of a more relaxed form of monopoly: forcing the rarity of a block up by intentionally *hoarding* it, that is, refusing to give it out to peers until it becomes so rare that peers may be willing to pay a premium price for it. Clearly, BitTorrent is sufficiently robust to overcome a single hoarding node, but what about a

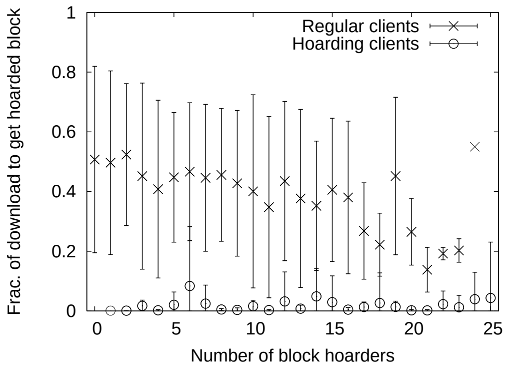


Figure 7.1: Rarest-first overcomes attempts at block monopolies.

colluding set of peers? We answer this by modifying the Azureus BitTorrent client to perform block hoarding. In our implementation, a hoarder knows the blocks he wishes to hoard *a priori*, attempts to obtain these blocks as soon as possible, and then refuses to inform others (with BitTorrent have messages) that he has the blocks. We ran experiments on a local cluster of 25 machines, varying the number of nodes hoarding the same piece.

Figure 7.1 shows that normal (non-hoarding, rarest-first) clients overcome block hoarding; that is, with an arbitrary fraction of hoarders, the normal clients obtain the block. Indeed, they obtain it more quickly; the expected point in the download at which a block is downloaded is 50% into the download, but the hoarders force the observed availability of the block down, causing normal clients to attempt to download it sooner. This is reflected in Figure 7.1; with no hoarders, the given block is centered around the expected value of 50%, but with more hoarders, normal clients tend to obtain the block sooner than halfway through the download.

Our experiments capture what happens when a node's hoarding strategy does *not* work, in the sense that at no point does a hoarder reveal its hoarded block and attempt to use it to download more from others. Figure 7.2 demonstrates that, when a hoarding

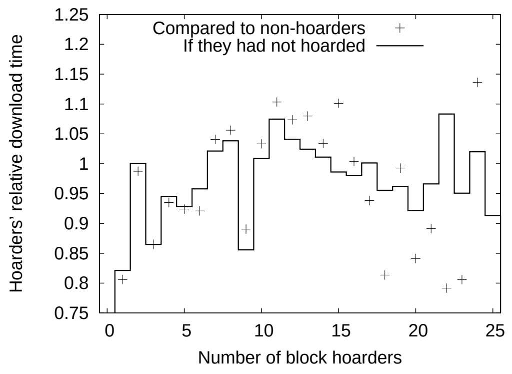


Figure 7.2: Hoarders' download times.

strategy does not work, it can have adverse effects on the peer's download time. The dots represent the relative download times to the other, non-hoarding clients, and the histogram represents the hoarders' download time relative to if they had not hoarded, i.e., relative to the download time when the number of hoarders is zero. One can interpret this figure as follows: it is in a peer's best interest to hoard blocks only when the dots are below the line.

There is a parabolic trend in Figure 7.2 that is consistent across all runs of this experiment. A few hoarders experiences faster download times than non-hoarding peers, slower when the number of hoarders is roughly equal to the number of non-hoarders. When hoarders far outnumber non-hoarders, hoarders perform comparatively faster, but, as shown by the solid line in Figure 7.2, harm overall performance.

We conclude that hoarding-based strategies *can* work, but (1) since it would require such widespread collusion, it is infeasible, and (2) given that it can *increase* download times if many peers perform it, it may not be a worthwhile risk for a peer to take.

7.4 Under-reporting to prolong interest

In both the BitTorrent and prop-share auctions, bidders consist solely of the peers *interested* in the peer running the auction. Although forcing a block monopoly appears to be infeasible, increasing the number of bidders—the number of interested peers—is possible with *strategic piece revelation*.

A piece revelation strategy dictates which blocks a peer claims to make available to its neighbors. Suppose β_i represents peer i's block bit-field, where $\beta_i(k) = 1$ if i has block k and 0 otherwise. i reports its bit-field to new neighbors, and sends updates in the form of have messages to existing neighbors. Let β'_i represent the bit-field i presents its neighbors, which need not be its *true* value β_i. If i *over-reports* block k, that is $\beta'_i(k) > \beta_i(k)$, then i's neighbors could easily detect this by not obtaining k upon request, and could subsequently punish i by allotting him less bandwidth. Hence, we can assume that $\beta'_i(k) \leq \beta_i(k)$, that is, that i will not over-report the blocks it has, but may under-report. Further, if i reports some block k and peer j requests it, then i must deliver that block. Otherwise, if i sends block k', it may hurt j's download time—j may have requested k' from another peer, and thus j could have spent time downloading two copies of the same block—thus j would have incentive to punish i. Hence, we can further assume that peer i delivers the blocks it reports, and that the only avenue for strategic piece revelation is under-reporting.

An under-reporting strategy

Why would i under-report its blocks? Certainly, i would not under-report to the point of becoming uninteresting to its peers, as the more interesting i is, the greater number of potential bids it can receive. A myopic peer might attempt to maximize its interest by declaring its true bit-field, as BitTorrent and BitTyrant currently do. However, a peer need not report *all* of the blocks it has to be equally interesting. Further, it is important to note that *the blocks that i gives out at round t affect how interesting other peers find i in future rounds*: Consider the simple motivating example in Figure 3.1(a), in which i has two neighbors j and k, both of whom find i interesting but do not find one another interesting. This scenario is preferable to i; both j and k will bid at i and i alone. Were i to truthfully report $\beta_i(t)$, j and k could request from i different blocks, which would make j and k interested in each other. This would then place i into scenario (b) or, if those were the only two blocks holding j and k's interest, scenario (e). Thus, i would under-report his blocks in order to maintain his neighbors' prolonged interest.

We conclude that peers indeed have incentive to strategically *under*-report what blocks they have. This leads us to the piece revelation Algorithm 11.

Algorithm 11 Strategic piece revelation. Run by i when peer j becomes uninterested in i.

```
1. Let $\beta'_j$ denote $j$'s bitfield, and $L_i(j)$ the list of pieces that $i$ has revealed to $j$.
2. If there does not exist any piece $p$ such that $\beta'_j(p) < \beta_i(p)$ then quit; $i$ cannot
   truthfully gain $j$'s interest.
3. Find the piece $p$ with $\beta'_j(p) < \beta_i(p)$ that *maximizes* the number of other neighbors
   $k$ for which (i) $k$ also has $p$: $\beta_k(p) = \beta_i(p)$, or (ii) $i$ has revealed $p$ to $k$: $p \in L_i(k)$.
4. Send a have message to $j$, revealing that $i$ has piece $p$, and add $p$ to $L_i(j)$.
```

Algorithm 11 is reactive; peer i only reveals that he has a piece to j when j loses interest. Instead of revealing the rarest piece he has, peer i reveals the *most common* piece he has that j does not. Providing j with a rare piece would make j more interesting to his neighbors, potentially removing some interest from i. This corresponds to i attempting to maintain the state in Figure 3.1(a) as opposed to (b), (d), or (f).

Evaluation of strategic piece revelation

We evaluate our strategic piece revelation on PlanetLab, and present the results in Figure 7.3. The experiment consists of two runs. In both, all but one peer start at the same time and run the standard BitTorrent client, revealing all of their pieces. The remaining peer runs one of the two strategies and starts 20 seconds after all the other peers, to test whether strategic block revelation can maintain the interest of peers who have many more blocks than he does. This result clearly shows the power in under-reporting one's pieces; the strategic peer is able to maintain others' interest over a prolonged period of time. Also clear from Figure 7.3 is the power of maintaining interest; the strategic peer downloads more quickly, as more peers place their bids throughout the download.

It seems intuitive that strategic piece revelation would induce a tragedy of the commons: that as the number of strategic piece revealers increases, system-wide download times would be severely increased. We ran experiments in which all peers strategically revealed their pieces. We saw on average a 12% increase in system-wide download times when *all* peers strategically revealed. That there is an increase in download times is unsurprising; a strategic piece-revealing peer does not know *which* pieces to offer to another strategic peer to garner its interest. The increase in system-wide download times is not much higher because peers continue to reveal pieces to their neighbors until they gain their interest.

These results indicate that selfish BitTorrent clients can benefit from under-reporting, but that as this practice becomes widespread, there may be an overall performance loss. There is therefore more to be understood about how strategic piece-revealers should inter-

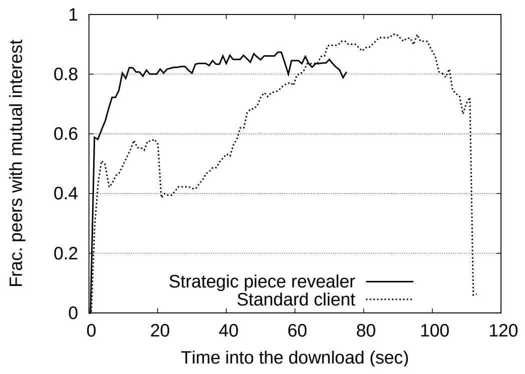


Figure 7.3: An example run, showing that strategic piece revelation can yield prolonged interest from neighbors, and hence faster download times in general.

act with one another. For instance, one area of future work is to consider strategic *interest* declaration, as a means of encouraging neighbors to reveal as many pieces as they truly have.

7.5 Solutions to under-reporting

Providing systems-compatible incentives for peers to truthfully report their bitfield is unfortunately an open problem. To understand why it has proven to be such a difficult problem, let us consider alternative, systems-incompatible solutions, so as to better understand the solution space.

A peer's bitfield is unlike other pieces of private information—such as bandwidth or how much the peer values the file—in that the bitfield is composed of observable events. Each positive entry in the bitfield was observed by another: the peer who uploaded the corresponding piece. Solutions to piece under-reporting leverage a collection of these small observations to achieve a verifiable bitfield.

Under-reporting is equivocation

We observed [72] that under-reporting is a form of attack called equivocation [25]. A peer *equivocates* by sending conflicting messages to two peers. Although quite simplistic, equivocation forms the basis of extremely powerful attacks. Chun et al. [25] proved that, were equivocation impossible, the Byzantine Generals Problem could be solved with $2f+1$ peers instead of $3f+1$, effectively demonstrating that equivocation is what makes Byzantine faults so detrimental.

To see why under-reporting is a form of equivocation, consider the following example. Peer u attempts to under-report to victim peer v; concretely, u states that he does not have piece p_ℓ, despite the fact that he downloaded the piece from peer ℓ prior to making that statement. Certainly, u is lying, but lying in and of itself does not indicate the conflict necessary for equivocation. When u downloaded p_ℓ, he must have acknowledged to ℓ that he received the piece. The subsequent bitfield not containing p_ℓ is a direct conflict with this prior acknowledgment, and thus an act of equivocation.

Identifying under-reporting as equivocation lends insight into the space of mechanisms that solve it. The classic solution to equivocation is the solution to the Byzantine Generals Problem by Lamport et al. [66], in which participants forward their statements to one another in an attempt to compare their own observations with the observations of others. Given a super-majority of honest participants—$3f+1$ participants total—the Lamport et al. show that they can achieve consensus. Equivocation is but one of the ways a dishonest participant can attempt to derail this consensus, and as mentioned earlier, it is the most powerful.

Accountability systems

PeerReview [52] and Nysiad [54] add Byzantine fault tolerance to arbitrary distributed systems. Both of these systems seek to obtain a verifiable set of actions on behalf of each participant, so as to detect equivocation and hold peers accountable for their actions. They achieve these verifiable logs by eliciting other peers to observe their actions. For example, PeerReview requires each peer to inform a set of witness peers whenever he sends a packet.

To apply these systems to BitTorrent, each peer p would be responsible for reporting to a pre-defined set of other peers all of p's actions as well as all of the actions p directly observed. Further, p would have to maintain and process queries on the logs corresponding to the peers whom p is responsible to monitor. Such systems are not compatible with BitTorrent in that they both introduce an enormous amount of communication overhead. Nysiad's introduces an order of magnitude increase in the number of messages sent, which degrades performance for a bandwidth-sensitive application like BitTorrent. These systems' communication overhead is in part due to the fact that a super-majority of honest peers must be included in the consensus because equivocation is possible.

Trusted hardware

We proposed TrInc [72] as an alternative solution to equivocation that incurs far less communication overhead. TrInc (Trusted Incrementer) is a simple primitive: a trusted, monotonically non-decreasing counter. Users provide a hash of some data to a TrInc-enabled piece of hardware—which we call a trinket—to obtain an attestation binding that data to a counter value. When obtaining an attestation, users specify a new counter value, and so long as that value is no less than the current value, the trinket performs the operation. Unlike the systems mentioned above, which detect equivocation, TrInc makes equivocation *impossible*. TrInc is designed such that a peer can equivocate only by causing a trinket to generate two conflicting messages using the same counter. Tamper-proof hardware makes this only as possible as one is unwilling to trust the hardware manufacture.

One can apply TrInc to make under-reporting in BitTorrent impossible as follows. Each user must be equipped with a trinket. A peer's TrInc counter value represents the number of pieces that peer has downloaded. When peer ℓ uploads piece p_ℓ to peer u, ℓ demands in return an attestation from u binding a new counter value to a bitfield containing the piece that ℓ just uploaded to u. Because the counter value cannot be decreased, this new counter value is forever bound to a bitfield containing this new piece. Hence, if peer v periodically requests u's most recent attestation, u cannot hide the fact that he has downloaded piece p_ℓ. Because the counter is monotonically increasing, a peer can only lie by stating that he has pieces which he does not, which is easily detected.

The benefits of TrInc over other accountability systems like PeerReview and Nysiad is that a set of witnesses are not necessary to detect if a peer is under-reporting. TrInc can be applied to BitTorrent by simply piggybacking small attestations to BitTorrent's standard messages, incurring only small additional communication overhead. Nonetheless, TrInc is not systems-compatible, as it requires a global deployment of trusted hardware. A major design goal of TrInc was ease of deployment. If the TrInc primitive sees such a large-scale deployment, then mechanisms that leverage it could arguably be systems-compatible. This is a prime example of why our definition of systems-compatibility must be flexible, so as to support future technologies as they become widely deployed.

These alternative solutions to under-reporting addressed the problem not by providing incentive to peers to truthfully state how many pieces of the file they have, but by making it impossible to state anything but the truth. This raises a question of how to distinguish between a problem solvable by incentives and one solvable only by security mechanisms. We discuss in Chapter 8 the importance of combining incentives with security protocols to ensure proper behavior across a wide range of attacks.

Chapter 8

Conclusion

8.1 Design principles

Throughout this dissertation, we have discussed mechanisms with varying degrees of systems-compatibility. We summarize our findings in this section by attempting to extract the design principles that have lead to successful, systems-compatible incentives.

8.1.1 Systems-compatibility

The fundamental finding of this dissertation is that by aligning the goals and assumptions of an incentive mechanism with the system to which it is being applied, we can ensure resilience against selfish manipulation in live, deployed systems. In addition to mitigating selfish attacks, we have also demonstrated improved performance; PropShare, for instance, improves download times in BitTorrent for all peers who employ it. We summarize this finding with the following design principle.

Design Principle 1. *Design and build mechanisms that are aligned with the given system's goals and assumptions. The assumptions may be extended so long as they yield a* *significant improvement to performance, as well.*

8.1.2 Money

In Chapter 2, we discussed the infeasibility of incorporating a "true" monetary system into a decentralized system, as well as the system performance costs of general-form money replacements. Successful alternatives for money have leveraged domain-specific properties, such as wireless nodes' limited battery life and desire to maintain connectivity [67]. Hoodnets (Chapter 5) demonstrates that money is not universally systems-incompatible, predominantly because the underlying system—cellular data access—already required user payment. This leads us to the following design principle.

Design Principle 2. *Avoid money when possible, particularly if the underlying system does not already incorporate it. Look for domain-specific replacements that are backed by an intrinsic good.*

8.1.3 Fairness

Within the context of file swarming, we considered two vastly different forms of fairness. FOX (Chapter 6) achieved perfect, bit-by-bit fairness; all but a small constant number of users upload exactly as much as they download. This is appealing from an incentives design perspective, in that it ensures that the addition of a user does not in any way decrease the utility of the others. However, it also means that the addition of a user does not *improve* the utility of the others. User capacities are heterogeneous [98]. There may be some users who are able and willing to upload more to the system than they download, yet FOX does not readily allow for this extra contribution. To do so, multiple FOX instances would have to run concurrently. In short, while FOX sets a lower bound on the amount a peer contributes, it does not provide incentive to peers to upload as much as possible.

Conversely, PropShare (Chapter 3) pursued a more relaxed form of fairness: the more a peer gives, the more that peer gets. This fairness property is more relaxed in the sense that it does not specify any peer's return on investment; one PropShare peer could yield two file pieces for every one he uploads, whereas another could experience the inverse. This is less appealing than bit-by-bit fairness in that a peer may not be able to estimate precisely what performance to expect at any given time. On the other hand, because it provides incentive to all peers to upload as much as possible, PropShare will typically outperform FOX.

This indicates that fairness constraints induce a trade-off between system performance and the ability to hold participants accountable for their actions. We propose the following design principle:

Design Principle 3. *Choose the fairness constraint that matches the system's trade-off between performance and accountability. More relaxed fairness may lead to better performance.*

In the context of file swarming, accountability is a secondary concern to performance, thereby making PropShare's fairness more suitable than that of FOX.

8.1.4 Prolonged interaction

In systems with asymmetric interest, such as PeerWise, peers must be provided incentive to interact with one another over a prolonged period of time. As we showed in Chapter 4, the nature of such interaction benefits from being rigidly defined so peers know what is expected from them and what to expect from others. We propose the following design principle.

Design Principle 4. *Make explicit and in-band the conditions of a long-lived interaction under non-simultaneous interest, and define actions to take in the event that the agreement is broken.*

This is analogous to the design principle proposed by Clark et al. [27], who argue that resolving "tussles" between self-interested principles in the Internet merit moving out-of-band agreements—such as how much one ISP pays another—into explicit protocol messages. Our observation that incentives-related agreements should be in-band is one extension of this argument.

8.1.5 Ground-up incentives design

Bandwidth auctions have been studied by networking researchers and algorithm game theorists for decades, but unfortunately none have experienced wide-spread deployment. Within the context of Hoodnets (Chapter 5), we demonstrated that bandwidth auctions merit a ground-up design of an end-hosts' forwarding architecture. For example, because a Hoodnets peer may cease forwarding for one of its neighbors if there is another neighbor willing to pay more, each peer must be able to quickly switch their forwarding paths. This is what merited a forwarding architecture capable of forwarding packets corresponding to a single TCP connection across multiple paths, which in turn led us to design an incentives mechanism capable of supporting this traffic pattern. We summarize this finding with the following design principle.

Design Principle 5. *Incorporate incentives decisions into the early design stages of a system. The underlying architecture may need to be tailored to the incentive mechanisms.*

Additionally, we observed with Hoodnets that realistic traffic patterns made no single incentive mechanism sufficient. Incorporating the underlying architecture's demands and capabilities is crucial to designing the set of incentives necessary to support it.

Design Principle 6. *If the underlying system must support multiple different inputs, then one incentive mechanism may not be enough. Consider incorporating incentive mechanisms that compose well together to address the system's multiple operating points.*

8.1.6 Punishment

Extreme forms of punishment have been demonstrated to be powerful in a theoretical sense, as in FOX (Chapter 6) and in wireless channel jamming [71]. But such punishments may present opportunities for acts of malice. In Chapter 6, we demonstrated that there are also more subtle forms of punishment, such as negative feedback in a reputation system. The necessity of making a threat of punishment credible leads us to the following design principle:

Design Principle 7. *Prefer carrots to sticks when possible. Expose subtle forms of punishment, and ensure the threats thereof are credible.*

8.1.7 Truthfulness

In virtually all decentralized systems, peers maintain private information that they are expected to report, yet few provide incentives for peers to do so truthfully. We observed within the context of under-reporting in BitTorrent that this lack of incentives can lead to a degradation of system-wide information. We propose the following design principle.

Design Principle 8. *For each piece of private information that a peer is expected to truthfully report, ensure that the peer has incentive to do so. Understating their capabilities may be at least as detrimental as overstating.*

Unfortunately, the mechanisms necessary to do so may not be systems-compatible. A system designer must evaluate the trade-offs between incorporating a systems-incompatible mechanism—such as an accountability system like PeerReview, Nysiad, or TrInc—and the degradation caused from a lack of truthful incentives. This motivates the design of easily deployable mechanisms like TrInc [72] that could extend the set of systems-compatible accountability mechanisms.

8.2 Open problems in systems-compatible incentives

This dissertation has demonstrated that systems-compatible incentives can bring both resilience to selfish manipulation as well as performance improvements. Nonetheless, there remain several important open problems in ensuring proper behavior of all Internet participants.

We uncovered one such open problem in Chapter 7; equivocation, what one might consider a security or accountability problem, is not readily solved by incentives alone. This motivates what I find to be the most important open problem in this space: designing a combination incentive and security mechanisms to provide incentives when possible, and to make impossible selfish or malicious acts that incentives alone cannot solve.

Example problem: Privacy-preserving targeted advertising

A concrete example of the necessity of both incentives and security is online advertising. Advertisements fuel the economy of the Internet. Users are able to enjoy free online services such as Google search or Facebook because these services obtain revenue from advertisements. Furthermore, in pursuit of greater conversion rates—that is, the likelihood that a user shown an ad will click on and purchase the advertised product—online advertising is increasingly becoming highly *targeted.* An advertisement selection algorithm is targeted if it takes into account user information, such as search history, birth date, sexual preference, purchasing history, and so on, when choosing an advertisement to show to the user. One could argue that targeted advertising is of benefit to users, as, in the ideal case, it presents to them precisely the product they wish to purchase precisely when they want to purchase it. However, today's targeted advertising algorithms require access to users' personal information. In effect, users "pay" for free online services with their private data.

But is this a necessary cost for users to pay? Can we design accountability and incentive mechanisms that restrict online service providers' access to personal information while maintaining the ability to provide targeted advertisements? There have been some recent mechanisms that address this problem [51, 56]. However, the solutions proposed to date require modifications to the current economic model of online targeted advertising, such as the introduction of a trusted third part [51] or the restriction of what ads can be displayed [56]. We suspect that modifying the economic model would lead to deployment hurdles; what incentive would Google have to deploy a privacy-preserving advertising architecture if some of their revenue had to be shared with a third party?

We posit that the design principles outlined in this chapter can be applied to guide the design of a privacy-preserving targeted advertisement architecture that can be deployed today. Of utmost priority is maintaining the assumptions and goals of today's advertising system: online services keep all of the revenue and are able to target ads with arbitrary precision. Without maintaining this principle, deployment would be difficult at best. Second, our design principles dictate that we take into account the demands of the system. Jules [56] suggests a computationally intensive means of retrieving advertisements, but if typical user patterns involve obtaining many ads over a short period of time, such a mechanism would not be compatible.

Online advertising represents but one example where incentives built into a protocol may not be sufficient to halt all acts of malice and misbehavior. Further understanding the space of what incentives do and do not make possible, and where security mechanisms are necessary, is in my opinion the most important open problem in aligning the competing interests of the Internet's participants.

Appendix A

Community sizes in FOX

Recall that, a leaf community C can only split when it has one peer to remain as the internal node, and ℓ nodes for each of the k new communities. Consider the worst case, that is the number of nodes in the community, $|C|$, is 1, and let $S > \ell$ be the number of nodes that must be added to C such that, for some $\delta < 1/k$,

$$\Pr[C \text{ cannot split } \mid |C| = S + 1] \ \leq \ k\delta$$

If n_i is the number of nodes that will go into the i^{th} leaf community after splitting, $i = 1, \ldots, k$, then the expected value of n_i is S/k. We can therefore express the above probability as

$$\Pr\left[\bigvee_{i=1}^{k}(n_i \leq \ell - 1)\right] \ \leq \ \sum_{i=1}^{k} \Pr\left[n_i \leq \ell - 1\right]$$

It suffices to have, for an arbitrary $i \in [k]$,

$$\Pr\left[n_i \leq (1-\epsilon)\frac{S}{k}\right] \;\leq\; \delta$$

where $0 < \epsilon = 1 - k\ell/S < 1$. Applying a Chernoff bound and the definition of ϵ, we obtain $\exp\left[-S\epsilon^2/(2k)\right] \leq \delta$

$$\begin{aligned} S &\geq \frac{2k}{\left(1-\frac{k\ell}{S}\right)^2}\,\ln\left(\frac{1}{\delta}\right) \\ \Rightarrow S &\geq k\left(\ell + \ln\frac{1}{\delta} + \sqrt{2\ell\ln(1/\delta) + \ln^2(1/\delta)}\right) \end{aligned} \tag{A.1}$$

Given a failure probability (δ), degree (k) and the minimum desired number of nodes in a leaf community (ℓ), we can calculate the minimum size of a community that will allow the leaf community to split (S). For example, if $k = \ell = 5$, we need roughly 114 nodes to be able to split with probability 99.9% (observe that this is $114/26 < 4$ times the minimum number we would need).

We now consider the upper bound, $M > S > \ell$, for the size to which any community can grow. Let $\mathcal{E}$ be the event that, when all communities are able to split, no community has more than M nodes, and let the probability of event $\mathcal{E}$ be at least $(1-\alpha)$. Let σ be the rhs of Equation A.1. Define X to be the number of joins necessary such that all k^H leaf communities receive at least σ joins with probability at least $(1-\alpha/2)$. Clearly, $X \geq k^H\sigma$, so there exists some $0 \leq \epsilon_1 < 1$ such that $X(1-\epsilon_1)/H = S$. With X joins, each community receives an expected X/k^H nodes. Thus, applying a Chernoff bound and bounding the probability of failure to $\alpha/2$, we get:

$$\frac{\epsilon_1^2}{1-\epsilon_1} \geq \frac{2\ln\left(\frac{2k^H}{\alpha}\right)}{H\sigma}$$

Turning to M, there must exist some ϵ_2 such that $X(1+\epsilon_2)/k^H = M$. Applying a Chernoff bound, keeping in mind that the expected size of each community is still X/k^H, we obtain

$$(1+\epsilon_2)\ln(1+\epsilon_2) - \epsilon_2 \geq \frac{k^H}{X}\ln\left(\frac{2k^H}{\alpha}\right)$$

$\square$

For example, if $k = \ell = 5$ and $N = 1,000,000$, then with at least 99.9% probability, the largest a community will grow is at most 285.

Bibliography

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

[2] H. Adiseshu, G. Parulkar, and G. Varghese. A reliable and scalable striping protocol. In *Proceedings of the SIGCOMM Conference on Data Communication*, 1996.

[3] M. Adler and D. Rubenstein. Pricing multicasting in more practical network models. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2002.

[4] M. Afergan. Using repeated games to design incentive-based routing systems. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*, 2006.

[5] A. Akella, S. Seshan, R. Karp, S. Shenker, and C. Papadimitriou. Selfish behavior and stability of the Internet: A game-theoretic analysis of TCP. In *Proceedings of the SIGCOMM Conference on Data Communication*, 2002.

[6] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. Resilient overlay networks. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[7] N. Andrade, M. Mowbray, A. Lima, G. Wagner, and M. Ripeanu. Influences on cooperation in BitTorrent communities. In *Proceedings of the Workshop on Economics of Peer-to-Peer Systems (P2PEcon)*, 2005.

[8] R. J. Aumann and M. Maschler. Game theoretic analysis of a bankruptcy problem from the Talmud. *Journal of Economic Theory*, 36(2):195–213, 1985.

[9] R. Axelrod. *Evolution of Cooperation*. Basic Books, New York, 1984.

[10] A. Balasubramanian, R. Mahajan, A. Venkataramani, B. N. Levine, and J. Zahorjan. Interactive WiFi connectivity for moving vehicles. In *Proceedings of the SIGCOMM Conference on Data Communication*, 2008.

[11] S. Banerjee, S. Lee, B. Bhattacharjee, and A. Srinivasan. Resilient multicast using overlays. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2003.

[12] A. Bharambe, J. R. Douceur, J. R. Lorch, T. Moscibroda, J. Pang, S. Seshan, and X. Zhuang. Donnybrook: Enabling large-scale, high-speed, peer-to-peer games. In *Proceedings of the SIGCOMM Conference on Data Communication*, 2008.

[13] BitTorrent. `http://www.bittorrent.com/`.

[14] E. Blanton and M. Allman. On making TCP more robust to packet reordering. *Computer Communication Review (CCR)*, 32(1):20–30, 2002.

[15] L. Brakmo, S. O'Malley, and L. Peterson. TCP Vegas: New techniques for congestion detection and avoidance. In *Proceedings of the SIGCOMM Conference on Data Communication*, 1994.

[16] L. Buttyán and J.-P. Hubaux. Stimulating cooperation in self-organizing mobile ad hoc networks. *ACM/Kluwer Mobile Networks and Applications (MONET)*, 8(5), 2003.

[17] J. W. Byers, J. Considine, M. Mitzenmacher, and S. Rost. Informed content delivery across adaptive overlay networks. In *Proceedings of the SIGCOMM Conference on Data Communication*, 2002.

[18] J. W. Byers, M. Luby, M. Mitzenmacher, and A. Rege. A digital fountain approach to reliable distribution of bulk data. In *Proceedings of the SIGCOMM Conference on Data Communication*, 1998.

[19] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communication (JSAC)*, 20(8), 2002.

[20] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: High-bandwidth content distribution in a cooperative environment. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[21] R. Chakravorty, S. Agarwal, S. Banerjee, and I. Pratt. MoB: A mobile bazaar for wide-area wireless services. In *Proceedings of the ACM Conference on Mobile Computing and Networking (MobiCom)*, 2005.

[22] D. Chaum, A. Fiat, and M. Naor. Untraceable electronic cash. In *International Cryptology Conference (CRYPTO)*, 1990.

[23] A. Cheng and E. Friedman. Sybilproof reputation systems. In *Proceedings of the Workshop on Economics of Peer-to-Peer Systems (P2PEcon)*, 2005.

[24] Y.-H. Chu, S. G. Rao, and H. Zhang. A case for end system multicast. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2000.

[25] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[26] Cisco Systems Inc. Inverse multiplexing over ATM (IMA) on Cisco 2600 and 3600 routers, 2005.

[27] D. D. Clark, J. Wroclawski, K. R. Sollins, and R. Braden. Tussle in cyberspace: Defining tomorrow's Internet. In *Proceedings of the SIGCOMM Conference on Data Communication*, 2002.

[28] E. Clarke. Multipart pricing of public goods. *Public choice*, 11, 1971.

[29] B. Cohen. Blog entry regarding avalanche. Online: http://bramcohen.livejournal.com/20140.html?thread=226988.

[30] B. Cohen. Incentives build robustness in BitTorrent. In *Proceedings of the Workshop on Economics of Peer-to-Peer Systems (P2PEcon)*, 2003.

[31] J. Contreras, O. Candiles, J. De La Fuente, and T. Gomez. Auction design in day-ahead electricity markets. *IEEE Transactions on Power Systems*, 16(1), 2001.

[32] J. Corbo, S. Jain, M. Mitzenmacher, and D. C. Parkes. An economically principled generative model of AS graph connectivity. In *Proceedings of the Joint Workshop on the Economics of Networked Systems and Incentive-Based Computing (NetEcon+IBC)*, 2007.

[33] P. Cramton. Electricity market design: The good, the bad, and the ugly. In *Proceedings of the Hawaii International Conference on System Sciences*, pp. 1–8, 2003.

[34] P. Cramton, Y. Shoham, and R. Steinberg. *Combinatorial Auctions*. MIT Press, 2006.

[35] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A decentralized network coordinate system. In *Proceedings of the SIGCOMM Conference on Data Communication*, 2004.

[36] J. R. Douceur. The Sybil attack. In *Proceedings of the Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.

[37] J. R. Douceur and T. Moscibroda. Lottery trees: Motivational deployment of networked systems. In *Proceedings of the SIGCOMM Conference on Data Communication*, 2007.

[38] C. Dwork, M. Naor, and H. Wee. Pebbling and proofs of work. In *International Cryptology Conference (CRYPTO)*, 2005.

[39] C. Estan, A. Akella, and S. Banerjee. Achieving good end-to-end service using bill-pay. In *Proceedings of the Workshop on Hot Topics in Networks (HotNets)*, 2006.

[40] FCC. FCC auctions. http://wireless.fcc.gov/auctions.

[41] J. Feigenbaum, C. Papadimitriou, R. Sami, and S. Shenker. A BGP-based mechanism for lowest-cost routing. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, 2002.

[42] M. Feldman, K. Lai, and L. Zhang. A price-anticipating resource allocation mechanism for distributed shared clusters. In *Proceedings of the ACM Conference on Electronic Conference (EC)*, 2005.

[43] FON. http://www.fon.com/.

[44] E. J. Friedman and P. Resnick. The social cost of cheap pseudonyms. *Journal of Economics & Management Strategy*, 10(2):173–199, 2001.

[45] P. Garbacki, D. H. Epema, and M. van Steen. An amortized tit-for-tat protocol for exchanging bandwidth instead of content in P2P networks. In *IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, 2007.

[46] F. D. Garcia and J.-H. Hoepman. Off-line karma: A decentralized currency for peer-to-peer and grid applications. In *Proceedings of the Applied Cryptography and Network Security Conference (ACNS)*, 2005.

[47] E. Gerding, R. Dash, A. Byde, and N. Jennings. Optimal strategies for bidding agents participating in simultaneous Vickrey auctions with perfect substitutes. *Journal of Artificial Intelligence Research*, 32:939–982, 2008.

[48] P. Golle, K. Leyton-Brown, and I. Mironov. Incentives for sharing in peer-to-peer networks. In *Proceedings of the ACM Conference on Electronic Conference (EC)*, 2001.

[49] T. Groves. Incentives in teams. *Econometrica*, 41, 1973.

[50] K. P. Gummadi, H. Madhyastha, S. D. Gribble, H. M. Levy, and D. J. Wetherall. Improving the reliability of Internet paths with one-hop source routing. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[51] H. Haddadi, S. Guha, and P. Francis. Not all adware is badware: Towards privacy-aware advertising. In *IFIP Conference on e-Business, e-Services, and e-Society (I3E)*, 2009.

[52] A. Haeberlen, P. Kuznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[53] J. D. Hartline and T. Roughgarden. Optimal mechanism design and money burning. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, 2008.

[54] C. Ho, R. van Renesse, M. Bickford, and D. Dolev. Nysiad: Practical protocol transformation to tolerate Byzantine failures. In *Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI)*, 2008.

[55] R. W. Irving. An efficient algorithm for the stable roommates problem. *Journal of Algorithms*, 6:577–595, 1985.

[56] A. Juels. Targeted advertising ... And privacy too. In *Proc. Conference on Topics in Cryptology: The Cryptographer's Track at RSA*, 2001.

[57] S. Jun and M. Ahamad. Incentives in BitTorrent induce free riding. In *Proceedings of the Workshop on Economics of Peer-to-Peer Systems (P2PEcon)*, 2005.

[58] S. Kamvar, M. Schlosser, and H. Garcia-Molina. The EigenTrust algorithm for reputation management in P2P networks. In *Proceedings of the International World Wide Web Conference (WWW)*, 2003.

[59] S. Kandula, K. C.-J. Lin, T. Badirkhanli, and D. Katabi. FatVAP: Aggregating AP backhaul capacity to maximize throughput. In *Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI)*, 2008.

[60] R. M. Karp, A. Sahay, E. E. Santos, and K. E. Schauser. Optimal broadcast and summation in the LogP model. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 1993.

[61] J. Kleinberg. The small-world phenomenon: An algorithm perspective. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, 2000.

[62] E. Kohler, M. Handley, and S. Floyd. Datagram congestion control protocol (DCCP), 2006. IETF RFC 4340.

[63] E. Kohler, M. Handley, and S. Floyd. Designing DCCP: Congestion control without reliability. In *Proceedings of the SIGCOMM Conference on Data Communication*, 2006.

[64] D. Kostić, R. Braud, C. Killian, E. Vandekieft, J. W. Anderson, A. C. Snoeren, and A. Vahdat. Maintaining high bandwidth under dynamic network conditions. In *Proceedings of the USENIX Annual Technical Conference*, 2005.

[65] K. Lai, L. Rasmusson, E. Adar, S. Sorkin, L. Zhang, and B. A. Huberman. Tycoon: An implemention of a distributed market-based resource allocation system. *Multiagent and Grid Systems*, 1(3):169–182, 2005.

[66] L. Lamport, R. E. Shostak, and M. C. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[67] S. Lee, D. Levin, V. Gopalakrishnan, and B. Bhattacharjee. Backbone construction in selfish wireless networks. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2007.

[68] S. Lee, R. Sherwood, and B. Bhattacharjee. Cooperative peer groups in NICE. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*, 2003.

[69] A. Legout, G. Urvoy-Keller, and P. Michiardi. Rarest first and choke algorithms are enough. In *Proceedings of the Internet Measurement Conference (IMC)*, 2006.

[70] R. LeMay. BitTorrent creator slams Microsoft's methods. *ZDNet Australia*, 2005.

[71] D. Levin. Punishment in selfish wireless networks: A game theoretic analysis. In *Proceedings of the Workshop on the Economics of Networked Systems (NetEcon)*, 2006.

[72] D. Levin, J. Douceur, J. Lorch, and T. Moscibroda. Trinc: Small trusted hardware for large distributed systems. In *Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[73] D. Levin, K. LaCurts, N. Spring, and B. Bhattacharjee. BitTorrent is an auction: Analyzing and improving BitTorrent's incentives. In *Proceedings of the SIGCOMM Conference on Data Communication*, 2008.

[74] D. Levin, R. Sherwood, and B. Bhattacharjee. Fair file swarming with FOX. In *Proceedings of the Workshop on Peer-to-Peer Systems (IPTPS)*, 2006.

[75] D. Levin, A. Bender, C. Lumezanu, N. Spring, and B. Bhattacharjee. Boycotting and extorting nodes in an internetwork. In *Proceedings of the Joint Workshop on the Economics of Networked Systems and Incentive-Based Computing (NetEcon+IBC)*, 2007.

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

[77] J. Liang, N. Naoumov, and K. W. Ross. The index poisoning attack in P2P file sharing systems. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*, 2006.

[78] N. Liogkas, R. Nelson, E. Kohler, and L. Zhang. Exploiting BitTorrent for fun (but not profit). In *Proceedings of the Workshop on Peer-to-Peer Systems (IPTPS)*, 2006.

[79] T. Locher, P. Moor, S. Schmid, and R. Wattenhofer. Free riding in BitTorrent is cheap. In *Proceedings of the Workshop on Hot Topics in Networks (HotNets)*, 2006.

[80] C. Lumezanu, D. Levin, and N. Spring. PeerWise discovery and negotiation of faster paths. In *Proceedings of the Workshop on Hot Topics in Networks (HotNets)*, 2007.

[81] J. K. MacKie-Mason and H. R. Varian. Pricing the Internet. In *Public Access to the Internet*, 1993.

[82] J. Martin and A. Nilsson. On service level agreements for IP networks. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*, 2002.

[83] L. Massoulié and M. Vojnović. Coupon replication systems. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2005.

[84] I. Molho. *The Economics of Information: Lying and Cheating in Markets and Organizations*. Wiley-Blackwell, 1997.

[85] A. Nakao and L. Peterson. Scalable routing overlay networks. In *ACM SIGOPS Operating Systems Review*, 2006.

[86] A. Nandi, T.-W. Ngan, A. Singh, D. Wallach, and P. Druschel. Scrivener: Providing incentives in cooperative content distribution systems. In *Proceedings of the ACM/IFIP/USENIX International Middleware Conference (MIDDLEWARE)*, 2005.

[87] G. Neglia, G. L. Presti, H. Zhang, and D. Towsley. A network formation game approach to study BitTorrent tit-for-tat. In *Proceedings of the Workshop on Network Control and Optimization (Netcoop)*, 2007.

[88] T.-W. J. Ngan, D. S. Wallach, and P. Druschel. Enforcing fair sharing of peer-to-peer resources. In *Proceedings of the Workshop on Peer-to-Peer Systems (IPTPS)*, 2003.

[89] T.-W. J. Ngan, D. S. Wallach, and P. Druschel. Incentives-compatible peer-to-peer multicast. In *Proceedings of the Workshop on Economics of Peer-to-Peer Systems (P2PEcon)*. Cambridge, Massachusetts, 2004.

[90] N. Nisan and A. Ronen. Algorithmic mechanism design. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, 1999. ISBN 1-58113-067-8.

[91] N. Nisan, T. Roughgarden, and Éva Tardos. *Algorithmic Game Theory*. Cambridge University Press, 2007.

[92] Novatel. http://www.novatelwireless.com/.

[93] A. Odlyzko. The history of communications and its implications for the Internet. *AT&T Labs – Research*, 2000.

[94] M. J. Osborne and A. Rubinstein. *A Course in Game Theory*. The MIT Press, 1994.

[95] Packeteer. Technology for intelligent bandwidth measurement. http://www.packeteer.com/technology/technology.htm.

[96] B. Parno, D. Wendlandt, E. Shi, A. Perrig, B. Maggs, and Y.-C. Hu. Portcullis: Protecting connection setup from denial-of-capability attacks. In *Proceedings of the SIGCOMM Conference on Data Communication*, 2007.

[97] M. Piatek, T. Isdal, A. Krishnamurthy, and T. Anderson. One hop reputations for peer to peer file sharing workloads. In *Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI)*, 2008.

[98] M. Piatek, T. Isdal, T. Anderson, A. Krishnamurthy, and A. Venkataramani. Do incentives build robustness in BitTorrent? In *Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI)*, 2007.

[99] D. Qiu and R. Srikant. Modeling and performance analysis of BitTorrent-like peer-to-peer networks. In *Proceedings of the SIGCOMM Conference on Data Communication*, 2004.

[100] A. Qureshi, J. Carlisle, and J. Guttag. Tavarua: Video streaming with WWAN striping. In *ACM Multimedia*, 2006.

[101] V. Rai, S. Sivasubramanian, S. Bhulai, P. Garbacki, and M. van Steen. A multiphased approach for modeling and analysis of the BitTorrent protocol. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, 2007.

[102] A. Ramachandran, A. D. Sarma, and N. Feamster. BitStore: An incentive-compatible solution for blocked downloads in BitTorrent. In *Proceedings of the Joint Workshop on the Economics of Networked Systems and Incentive-Based Computing (NetEcon+IBC)*, 2007.

[103] P. Rodriguez, R. Chakravorty, J. Chesterfield, I. Pratt, and S. Banerjee. MAR: A commuter router infrastructure for the mobile Internet. In *Proceedings of the ACM Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2004.

[104] M. H. Rothkopf, A. Pekeč, and R. M. Harstad. Computationally manageable combinatorial auctions. Tech. rep., DIMACS Technical Report 95-09, 1995.

[105] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proceedings of the ACM/IFIP/USENIX International Middleware Conference (MIDDLEWARE)*, 2001.

[106] S. Saroiu, K. P. Gummadi, R. J. Dunn, S. D. Gribble, and H. M. Levy. An analysis of Internet content delivery systems. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[107] B. Schneier. *Applied Cryptography*. John Wiley & Sons, 2nd edn., 1996.

[108] S. Shenker, D. Clark, D. Estrin, and S. Herzog. Pricing in computer networks: Reshaping the research agenda. *Computer Communication Review (CCR)*, 26(2):19–43, 1996.

[109] R. Sherwood, B. Bhattacharjee, and R. Braud. Misbehaving TCP receivers can cause Internet-wide congestion collapse. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2005.

[110] R. Sherwood, R. Braud, and B. Bhattacharjee. Slurpie: A cooperative bulk data transfer protocol. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*, 2004.

[111] J. Shneidman, D. C. Parkes, and L. Massoulié. Faithfulness in Internet algorithms. In *Proceedings of the ACM Workshop on Practice and Theory of Incentives in Networked Systems (PINS)*, 2004.

[112] M. Sirivianos, J. H. Park, R. Chen, and X. Yang. Free-riding in BitTorrent networks with the large view exploit. In *Proceedings of the Workshop on Peer-to-Peer Systems (IPTPS)*, 2007.

[113] M. Sirivianos, J. H. Park, X. Yang, and S. Jarecki. Dandelion: Cooperative content distribution with robust incentives. In *Proceedings of the USENIX Annual Technical Conference*, 2007.

[114] K. Sklower, B. Lloyd, G. McGregor, D. Carr, and T. Coradetti. The PPP multilink protocol, 1996. IETF RFC 1990.

[115] Skype. `http://www.skype.com/`.

[116] A. Snoeren. Adaptive inverse multiplexing for wide area wireless networks. In *Proceedings of the IEEE Global Communications Conference (GLOBECOM)*, 1999.

[117] E. Tan, L. Guo, S. Chen, and X. Zhang. CUBS: Coordinated upload bandwidth sharing in residential networks. In *Proceedings of the IEEE International Conference on Network Protocols (ICNP)*, 2009.

[118] K. Varadhan, R. Govindan, and D. Estrin. Persistent route oscillations in inter-domain routing. *Computer Networks*, 32(1):1–16, 2000.

[119] W. Vickrey. Counterspeculation, auctions, and competitive sealed tenders. *Journal of Finance*, 16, 1961.

[120] V. Vishnumurthy, S. Chandrakumar, and E. G. Sirer. KARMA: A secure economic framework for P2P resource sharing. In *Proceedings of the Workshop on Economics of Peer-to-Peer Systems (P2PEcon)*, 2003.

[121] J. Weesie. Incomplete information and timing in the volunteer's dilemma. *Journal of Conflict Resolution*, 38(3), 1994.

[122] B. White, *et al.* An integrated experimental environment for distributed systems and network. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[123] B. Wong, A. Slivkins, and E. G. Sirer. Meridian: A lightweight network location service without virtual coordinates. In *Proceedings of the SIGCOMM Conference on Data Communication*, 2005.

[124] F. Wu and L. Zhang. Proportional response dynamics leads to market equilibrium. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, 2007.

[125] X. Yang. Auction but don't block. In *Proceedings of the Workshop on the Economics of Networked Systems (NetEcon)*, 2008.

[126] B. Zhang, T. E. Ng, A. Nandi, R. RIedi, P. Druschel, and G. Wang. Measurement-based analysis, modeling, and synthesis of the Internet delay space. In *Proceedings of the Internet Measurement Conference (IMC)*, 2006.

[127] S. Zhong, J. Chen, and Y. R. Yang. Sprite: A simple, cheat-proof, credit-based system for mobile ad-hoc networks. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*, 2003.