The Dissertation Committee for Harry Chu-Kit Li
certifies that this is the approved version of the following dissertation:

Robust Peer-to-Peer Systems

Committee:

Lorenzo Alvisi, Supervisor

Michael Dahlin

Lili Qiu

Robbert van Renesse

Petros Maniatis

Robust Peer-to-Peer Systems

by

Harry Chu-Kit Li, B.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

May 2009

To Meg, Dad, and Mom

Acknowledgments

Seven years ago, I received advice about selecting a graduate school: "You don't need to be friends with your advisor." While that advice may still hold, my time in Austin would have been significantly less fulfilling if my advisor, Lorenzo Alvisi, were not both my mentor and friend these past few years. His constant support both intellectually and emotionally has made this dissertation possible. I look back upon my years in Austin and remark on how smooth many of the transitions have been. From taking classes to do doing research and from proposing to finishing a dissertation, it seems that each achievement has been a natural step from the one before. I know that such painless progress is due in no small part to a great advisor. I am forever thankful for his guidance in turning me into the researcher I am and will take memories of countless discussions about being a professor, music, film, art, and food with me to the Bay Area.

Mike Dahlin has been a source of inspiration in distilling challenging problems down to obscenely simple statements. His talent for doing so extends well beyond computer science. Comments such as "Of course it scales! Just throw more money at it." or "There are three numbers: one, two, and many." are classic examples. Perhaps the one that stands out most in my head is that people can endure a lot more pain when armed with an easy way out. I am deeply grateful for his unique perspective on life and research and how to balance the two.

I thank Petros Maniatis immensely for serving on my thesis committee. In

this past year, I have confirmed the widely held belief that Petros is a really nice guy. He has a knack for listening intently while asking critical questions without being judgmental, a talent that I hope to have one day. Let me also thank Robbert van Renesse whose confidence in this work and feedback have been invaluable in its progression.

My path in computer science would never have led to graduate school were it not for my time at Brown University with my mentors, Shriram Krishnamurthi and Kathi Fisler. They taught me what research is, how it gets done, and how to make it better. I value my friendship with them greatly and remember my first steps in research with fondness. A noteworthy one is the single line email feedback I received on a revised draft of my honors thesis introduction: "Globally better, but locally awful in whole new ways." I should have realized back then that praise and criticism are intimately linked in a successful research career.

I have been fortunate in graduate school to make three best friends: Allen Clement, Taylor Riché, and Jeff Napper. I am grateful to Allen for not only putting a roof over my head for two years, but also for his companionship as we grew up as researchers together. I have spent many hours of my life arguing with him about some niggling semantic difference that neither of us would surrender. His intelligence and critical eye have made my work orders of magnitude better. His company is something I miss dearly and I hope he stays in my professional and personal life for many years to come. Taylor has done a great job setting me straight when I inevitably go on a rant about something. His perspective has kept my head squarely on my shoulders. I truly miss Jeff's humor since his departure to the Netherlands. Despite the distance, I still solicit his wisdom gathered from many many years of experience. I hope he comes back to the States soon.

Let me thank several colleagues for making graduate school a joy in which to spend seven years. Serita Nelesen bore the brunt of getting me to select UT Austin

over my other graduate school options. I think her sacrifice in seeing Blade II with me should be amortized over the length of our continuing friendship. Edmund Wong has been of enormous help both in research and in knowing the current state of the stock market. Eric Rozner has been the butt of many Wisconsin jokes and has taken them in stride. His deadpan humor is unforgettable. Don Porter is perhaps the best cubicle mate of all time. His intelligence, friendliness, and absolutely inappropriate sense of humor has made each day at the office memorable.

My parents have been an inspiring story of what hard work can accomplish and I hope that I can do the same for my future family. They have been an enormous source of strength, love, and support as I made my way through college and then graduate school. In addition to wonderful parents, I also have the best older brother anyone could ask for. Thank you, Ray. As I search for jobs post-Ph.D., his faith in me despite a massive recession is reassuring. I look forward to the day when we are both living in the same place again.

And lastly, let me thank my best friend, Meg. She is my support network of one. She challenges me to be more while loving me for who I am. And she makes it obvious that despite a finished dissertation being a very important milestone, the things that matter most can never be written down. I am eternally grateful that she could share my journey in producing this document.

Harry C. Li

The University of Texas at Austin

May 2009

Robust Peer-to-Peer Systems

Publication No. _____

Harry Chu-Kit Li, Ph.D.
The University of Texas at Austin, 2009

Supervisor: Lorenzo Alvisi

Peer-to-peer (p2p) approaches are an increasingly effective way to deploy services. Popular examples include BitTorrent, Skype, and KaZaA. These approaches are attractive because they can be highly fault-tolerant, scalable, adaptive, and less expensive than a more centralized solution. Cooperation lies at the heart of these strengths. Yet, in settings where working together is crucial, a natural question is: "What if users stop cooperating?" After all, cooperative services are typically deployed over multiple administrative domains, and thus vulnerable to Byzantine failures and users who may act selfishly.

This dissertation explores how to construct p2p systems to tolerate Byzantine participants while also incentivizing selfish participants to contribute resources. We describe how to balance obedience against choice in building a robust p2p live

streaming system. Imposing obedience is desirable as it leaves little room for peers to attack or cheat the system. However, providing choice is also attractive as it allows us to engineer flexible and efficient solutions.

We first focus on obedience by using Nash equilibria to drive the design of BAR Gossip, the first gossip protocol that is resilient to Byzantine and selfish nodes. BAR Gossip relies on *verifiable pseudo-random partner selection* to eliminate non-determinism, which can be used to game the system, while maintaining the robustness and rapid convergence of traditional gossip. A novel *fair enough exchange* primitive entices cooperation among selfish peers on short timescales, thereby avoiding the need for distributed reputation schemes. We next focus on tempering obedience with choice by using approximate equilibria to guide the construction of a novel p2p live streaming system. These equilibria allow us to design incentives to limit selfish behavior rigorously, yet provide sufficient flexibility to build practical systems. We show the advantages of using an ϵ-Nash equilibrium, instead of an exact Nash, to design and implement FlightPath, our live streaming system that uses bandwidth efficiently, absorbs flash crowds, adapts to sudden peer departures, handles churn, and tolerates malicious activity.

Contents

List of Figures

Chapter 1

Introduction

In 1999, the infamous file-sharing program Napster [55] demonstrated that peer-to-peer (p2p) applications can be a very effective and disruptive technology. Since then, greater accessibility of broadband connections, decreasing storage costs, and more powerful microprocessors have made p2p an increasingly popular way to deploy services. P2P approaches are attractive because they can be highly fault-tolerant, scalable, and adaptive. Moreover, a p2p design can be less expensive than a traditional client-server approach, thereby lowering the barriers to entry in markets such as content distribution, telecommunications, and audio/video streaming. BitTorrent [18], KaZaA [36], Limewire [44], Skype [72], and LiveStation [48] are a small sample of today's well-known p2p applications.

Of course, different p2p services can take vastly different technological approaches in their applications. Yet, no matter how traditional or innovative we are, all p2p systems rely on a shared principle: cooperation. Peers pool their resources to achieve a common goal. Cooperation gives p2p systems their vaunted strengths. But those advantages are only as solid as the foundation on which they rest, leading to a simple question: What if peers stop cooperating?

1.1 Motivation

An application may never see the typical p2p benefits—such as scalability and adaptability—if it does not tolerate Byzantine users who may disrupt the service or selfish ones who may try to use it without contributing their fair share. Designing a practical p2p system that deals with both Byzantine and selfish users is challenging. First, the system needs to work well in a range of settings. For example, in a file-sharing application, users should typically see high download speeds despite the inherent unpredictability of the Internet. Second, the system should continue to work well despite the presence of Byzantine users. Byzantine behavior is inevitable in large-scale systems: computers crash, programs contain bugs, people misconfigure their systems, and some users will attack the system. Third, the system should ensure that selfish participants actually cooperate. Selfish peers are an unfortunate reality of p2p systems. Adar and Huberman estimate that nearly 70% of Gnutella users share no files [1]. When widespread, such behavior can severely hurt any service whose foundation relies on cooperation.

Several deployed applications [18, 36] and research prototypes [19, 31, 33, 57, 64, 63, 70] have recognized the need to curb selfish actions. Despite these systems' built-in incentives and punishments, selfish participants still frequently find ways to cheat. For example, consider the KaZaA network [36] in which it is estimated that nearly half the peers use KaZaA Lite [37], which falsifies users' contributions. The popular application BitTorrent uses a tit-for-tat based mechanism to encourage cooperation. However, Schneidman et al. [68] identify several ways a client could cheat, weaknesses that have since been exploited in recent BitTorrent clients [49, 64, 70].

Levin et al. [41] bring to light BitTorrent's ad hoc design by debunking the popular belief that BitTorrent uses a tit-for-tat mechanism. They show that BitTorrent actually uses an auction-based scheme, a design decision that explains

BitTyrant's [64] success in uploading only enough to trigger reciprocation. These vulnerabilities and misunderstandings in BitTorrent illustrate a broader point in system design: almost no p2p system *rigorously* shows that its built-in incentives and punishments are enough to induce cooperation.

We therefore take a formal approach to building a system to tolerate both Byzantine and selfish participants. We base our approach on the BAR model [3] in which Aiyer et al. combine game theory with traditional Byzantine fault-tolerance in building a cooperative backup system. This dissertation begins by leveraging techniques pioneered in the original BAR work and applying them in a new setting: p2p live streaming.

1.2 Contributions

This dissertation focuses on constructing p2p live streaming systems that are supported by a rigorous theoretical foundation. Live streaming is an increasingly useful application at several scales of deployment. For example, the first 2008 presidential debate was watched by over 1 million people online. At smaller scales, such as academic conferences or local concerts, there may be interest in providing a live stream, but little existing infrastructure to support it.

At all these scales, a p2p streaming solution is an intriguing alternative to traditional methods. For example, such a solution could absorb the impact of an unexpected flash crowd. Furthermore, large-scale content providers may adopt p2p-based solutions to shift costs (like bandwidth) to clients, and small-scale providers might find it simpler to use a self-organizing network instead of provisioning and maintaining a large dedicated server.

We have built two systems, BAR Gossip and FlightPath, that tolerate Byzantine and selfish peers. We base both systems on gossip protocols [10, 21]. As with all gossip protocols, peers in our protocols periodically exchange recently received data

with random neighbors. Birman et al. [10] show how such a simple communication pattern yields a robust and highly reliable multicast. For this reason, some have used gossip-based mechanisms as part of their streaming systems [32, 33, 60, 63]. Yet, to our knowledge, no one has considered how robust these mechanisms actually are with Byzantine and selfish peers. The key challenge we face in building a p2p live streaming system is in how to induce cooperation in a practical system.

BAR Gossip: A Purist Approach BAR Gossip consists of two novel gossip protocols, Balanced Exchange and Optimistic Push. We construct Balanced Exchange to be a Nash equilibrium [56]—a condition in which no selfish peer has an incentive to act alone in trying to cheat the system. In designing Balanced Exchange, we immediately face two practical hurdles. First, gossip's robustness stems from its randomness, a trait that makes it difficult to enforce obedience since peers can mask suspicious behavior as the product of improbable though not impossible events. We overcome the randomness problem by creating a verifiable pseudo-random algorithm. Second, the fair exchange of data between two peers is provably impossible to achieve without a trusted arbiter [25], which can be a bottleneck at large scales. We sidestep the impossibility result by designing an exchange primitive that is *fair enough*, guaranteeing that selfish peers do not attempt to cheat others in trades.

Our desire to ensure that cheating is not beneficial results in a theoretically appealing Balanced Exchange protocol that performs poorly. To fix this performance issue, we introduce the second half of BAR Gossip, Optimistic Push. This protocol is similar to Balanced Exchange, except that it allows peers to trade data with the hope, rather than the certainty, that they will receive the same number of useful data packets in return. Together with Balanced Exchange, the Optimistic Push protocol achieves good performance but has a theoretical drawback: We cannot prove that the Optimistic Push protocol is a Nash equilibrium, although our experiments suggest it is not easy to cheat the protocol.

FlightPath: A Practical and Rigorous Approach BAR Gossip's resilience to Byzantine and selfish peers is a good start towards designing p2p live streaming systems under the BAR model. However, our experience suggests that strict equilibria (e.g., Nash equilibria) may not provide the flexibility necessary to achieve great performance in p2p systems (whether for live streaming or not). We therefore turn to approximate equilibria, and in particular, ϵ-Nash equilibria [15] in which selfish peers expect to gain at most a factor of ϵ from unilateral deviations. We use these equilibria to trade resilience to cheating for practical concerns like performance and low overhead.

FlightPath—our streaming system based on ϵ-Nash equilibria—significantly improves stream quality over Balanced Exchange and Optimistic Push. The strength of these equilibria is that they allow us to balance enforcing obedience against providing enough flexibility to adapt to challenging situations. An example of this balancing act is in how we craft FlightPath's partner selection algorithm to both limit the number of partners that Byzantine peers can simultaneously contact while offering enough choices for other peers to select good trading partners. In addition to these practical benefits, ϵ-Nash equilibria also have theoretical ones. We can prove that FlightPath is an ϵ-Nash equilibrium without relying on experiments that only suggest this condition as we did when assessing whether Optimistic Push is a strict Nash equilibrium.

1.3 Roadmap

This dissertation consists of seven chapters. In the first, we provide motivation and an overview of the entire document.

Chapter 2 summarizes related work. Most readers may skip this chapter to more quickly reach the technical contributions of this dissertation. This chapter explains

where this work fits in the existing body of research on content distribution, gossip protocols, game theory and mechanism design, and p2p streaming systems.

Chapter 3 provides background on relevant game-theoretic concepts used in this dissertation. We use the framework already established by game theory to reason about how a selfish user would behave. This chapter also explains our assumptions regarding communication channels and node failures. In short, it defines our system model.

Chapter 4 describes the first half of BAR Gossip, Balanced Exchange, detailing the verifiable pseudorandom partner algorithm and the fair enough exchange mechanism. In addition, we prove in this chapter that the Balanced Exchange protocol is a Nash equilibrium.

Chapter 5 addresses some of the practical concerns inherent to Balanced Exchange. We introduce BAR Gossip's second half, Optimistic Push, and discuss its similarities and differences with Balanced Exchange. This chapter also compares BAR Gossip to a traditional gossip algorithm, assesses the robustness of Optimistic Push to cheating strategies, and evaluates BAR Gossip under attack.

Chapter 6 proposes approximate equilibria as a new way to design p2p services. We design and implement FlightPath, a novel p2p live streaming system based on ϵ-Nash equilibria. This chapter describes the basic design of FlightPath and the heuristics we use to improve overall performance. Additionally, we also discuss implementation details of the FlightPath prototype and an evaluation of FlightPath in several settings.

Chapter 7 highlights open questions related to designing robust p2p services and concludes this document.

Chapter 2

Related Work

This dissertation builds on a broad set of contributions in gossip protocols, bulk file transfer, and p2p streaming protocols. In the rest of this section, we summarize the works that have influenced BAR Gossip and FlightPath the most, drawing comparisons when appropriate.

2.1 Gossip Protocols

In BAR Gossip and FlightPath, we incorporate many techniques from several works in the gossip literature. In their seminal work, Demers et al. [21] show how to apply concepts from epidemiology to maintain consistency across replicated databases. Demers et al. resolved the scalability and performance problems of maintaining consistency across Xerox's Clearinghouse servers [58] by using epidemic algorithms to provide a weak form of consistency. In time, epidemic protocols also came to be known as gossip protocols. Years later, Petersen et al. [62] extend epidemic techniques from databases to filesystems. The resulting weakly consistent system, Bayou, was important in its conceptually simple approach—pair-wise update propagation—leading to a filesystem that could support a diverge range of networking environments.

Birman et al. [10] propose Bimodal Multicast, going beyond using gossip techniques for just maintaining replica consistency. They design a highly reliable multicast protocol that works in two phases. The first phase uses a light-weight best-effort multicast, such as IP Multicast, to carry out the initial spread of data in the system. The second phase leverages gossip as a repair mechanism, informing those nodes still ignorant of the latest updates. Bimodal Multicast ensures that nodes receive updates with high probability, establishing a middle ground between 100% reliable multicast protocols and best-effort approaches—the former scaling poorly and the latter providing disappointing guarantees.

In Astrolabe [74], van Renesse et al. show how to use gossip in yet another area: distributed monitoring and management. Astrolabe uses gossip-based techniques in a broader infrastructure for information management. Gossip provides a scalable way for nodes to exchange state with another, enabling a number of Astrolabe's applications: publish-subscribe, resource location, attribute aggregation, and large-scale state monitoring. Moreover, gossip provides a mechanism by which Astrolabe can manage membership information, incorporating new nodes when appropriate and removing very old entries that probably correspond to failed computers.

Eugster et al. [23] improve gossip's scalability in their work on light-weight probabilistic broadcast protocols (lpbcast). They propose a better message buffering algorithm that prioritizes gossiping more rare updates over those that are more well-known. In addition, Eugster et al. show that nodes do not have to maintain full membership lists to retain the robustness of gossip. A node can maintain a partial view that is selected uniformly at random from the full list. Their algorithm to maintain partial views appears to provide views that are close to random. Ganesh et al. [24] extend this partial view technique in Scamp, where they develop an algorithm to grow and shrink view sizes in a decentralized manner relative to system size.

Although techniques to maintain partial membership views strive to retain the robustness properties inherent to true randomness, both lpbcast and Scamp have not been thoroughly analyzed to determine whether gossip's robustness has indeed been kept. In Cyclon [77], Voulgaris et al. propose a way for nodes to shuffle their partial views so that the resulting communication graph imposed by those views possesses many desirable properties of random graphs, namely low diameter, low clustering, highly symmetric node degrees, and remarkable resilience to failures. Araneola [52], by Melamed et al., also improve upon overlay construction. Their approach allows nodes to dynamically build and maintain k-regular graphs in which every node has either degree k or $k + 1$. By controlling the randomness in graph construction, Araneola is able to improve load, reliability, and latency compared to standard gossip-based multicast approaches.

Johansen et al. [35] depart from traditional gossip-based membership management by aiming for a protocol that tolerates Byzantine participants while maintaining full membership lists. They argue that memory overheads are not the problem. Rather, overheads stemming from join and leave events can cause a high amount of network traffic. Johansen et al. demonstrate that FlightPath can maintain full membership lists and tolerate Byzantine members, while incurring modest network overheads. To our knowledge, Brahms [11], by Bortnikov et al., is the latest in the series of gossip-based membership protocols. This system is unique in its ability to tolerate Byzantine members while maintaining small view sizes. Key to their approach is a technique that enables nodes to independently and uniformly sample from the space of existing members. This achievement is a fundamental advance as previous techniques could only provide small views that empirically appeared to be random, whereas Brahms does so in theory as well.

Although gossip is appealing because of its simplicity and increased robustness, those benefits come with a cost. At large-scales, its uniformly random nature

can generate a large number of packets that congest common links. A few have proposed techniques to lift gossip's ignorance of the underlying topology. In their Leaf Box Hierarchy, Gupta et al. [30] organize peers into leaf boxes, which reside at the leaves of a tree. A member gossips with a neighbor with decreasing probability the farther away a member is in the tree. Provided that the mapping from members to leaf boxes reflects the proximity of nodes in the network, this gossiping algorithm reduces stress on common links that connect clusters of peers. Note that Gupta et al. leave a good mapping unspecified. In Directional Gossip [45], Lin and Marzullo adopt a different approach, advocating the use of two gossip protocols to cope with wide area network settings. They propose using a standard gossip algorithm within LANs and a separate algorithm across LANs. Nodes in different LANs gossip with one another with probability decreasing the better connected those two nodes are to one another, where connectivity is measured according to the size of the minimum link-disjoint set between those two nodes. The intuition behind this approach is that nodes which are not well-connected should seize the opportunity to gossip with one another.

To our knowledge, only two gossip protocols have taken advantage of network coding: Slingshot [7] and Ricochet [6] [1]. Both systems use receiver-based network coding as a repair mechanism in case an initial unreliable multicast fails to reach every node. Receiver-based network coding allows rapid recovery from message losses in time critical environments. Their authors intend Ricochet and Slingshot to be used in data centers where message loss occurs at the nodes and the networking infrastructure is assumed to be nearly infallible.

[1] As "Balakrishnan et al." could refer to either Slingshot or Ricochet, we refer to each system by name instead of by the authors.

2.2 File-sharing and Incentives

Perhaps the most well-known file-sharing program today is BitTorrent [18]. Its incentive structure, wide-availability, and numerous implementations have made it one of the most effective ways to transfer large files for both lawful and less lawful purposes. Those incentives suggest a robustness not seen in previous file-sharing applications such as Limewire [44] or Napster [55]. In this section, we summarize recent works that exploit weak points in BitTorrent's design, weaknesses that illustrate the dangers of informal incentive-based approaches. We begin with a summary of BitTorrent's incentive structure.

A BitTorrent client has incentive to upload to other clients as such action may trigger others to respond in kind. By default, a client chokes its uploads to all other clients. Periodically, a client chooses 4 clients to be unchoked. The 3 clients who have uploaded the most to a client comprise 3 out of those 4 unchoke slots. A client chooses the last slot at random. A BitTorrent client divides its upload bandwidth equally across the 4 partners that it has unchoked.

Shneidman et al. [68] identify several ways to game the BitTorrent protocol. Using a technique they name *manual backtracing*, Shneidman et al. propose four modifications to a BitTorrent client that can decrease download time or let a user free-ride off others.

1. The BitTorrent protocol is susceptible to a Sybil attack [22]; a client wishing to free-ride can create multiple identities to increase the chances of being optimistically unchoked.

2. There is little incentive for a peer to use more upload bandwidth than is necessary to be one of the top 3 uploaders for a neighbor. That extra bandwidth can be conserved or diverted to be a top 3 uploader for a different neighbor.

3. Once an optimistically unchoked client is again choked, that client has an

incentive to disconnect from its partner and reconnect. This exploit is an artifact of the original BitTorrent implementation.

4. A client can be credited with uploading data even if the data it uploads is garbage.

Although Shneidman et al. identify these weaknesses, they ignore the degree to which each can be exploited. Some may be theoretically possible but of little consequence practically. Uploading garbage data is clearly a practical weakness and is addressed in later BitTorrent clients such as Azureus [5]. Recent works have shown the remaining three are also weaknesses that can be gamed in practice [41, 49, 70].

In BitThief [49], Locher et al. instrument a client to reconnect to the tracker several times to obtain a large list of possible partners. A BitThief client then contacts all of these partners and is therefore known by many more peers than normal. Although not a Sybil attack, the underlying motivation is the same—increase the probability of being optimistically unchoked. Almost concurrently, Sirivianos et al. [70] develop an identical attack and term it the *large-view exploit*. Sirivianos et al. also include the third technique by Shneidman et al. to take advantage of the BitTorrent specification. Both works demonstrate free-riding is quite feasible in real BitTorrent swarms.

In BitTyrant [64], Piatek et al. explore how a strategic client can alter how it portions out upload bandwidth to increase aggregate download rates. A BitTyrant client estimates how much bandwidth is needed to be considered a top 3 uploader with respect to its neighbors. Any extra upload bandwidth is used to attract other neighbors. BitTyrant's heuristics show that a clever client can significantly improve its performance by being more strategic in who it uploads to and how much it uploads.

Levin et al. [41] dispel a common belief that BitTorrent's incentives stem from a tit-for-tat strategy. Although this observation is not new [68, 70], Levin et

al. show that BitTorrent's structure is essentially an auction that awards the top 3 bidders. They improve on BitTorrent's incentive structure by redesigning auctions such that the rewards are proportional to the amount of upload bandwidth each client bids. Framing BitTorrent's incentive structure cleanly captures the success of Piatek et al. with BitTyrant. Moreover, Levin et al. reveal a fifth way to game the protocol. A clever client can selectively lie about not having a file block to prolong beneficial relationships.

BitTorrent illustrates the power of using a p2p approach to deploy services. However, its weaknesses, as highlighted by several researchers, point to the dangers of an informal approach in dealing with selfish behavior. These shortcoming have triggered a series of proposals for more robust incentives in file-sharing systems [42, 54, 71, 75]. We elide a thorough discussion of incentives in file-sharing systems, focusing instead on incentives in streaming applications.

2.3 Streaming and Incentives

BAR Gossip and FlightPath are the latest in a series of p2p streaming proposals that include incentives to encourage cooperation. Our work differs from previous approaches in two important ways. First, we require our systems to tolerate Byzantine behavior. Second, we focus on showing formally that our incentives are enough to induce obedience. Existing p2p live streaming systems either crumble with the existence of even one malicious peer or fail to justify that the incentives provided are indeed enough. In this section, we begin with a brief summary of some streaming systems that do not incorporate incentives. Afterwards, we discuss systems that provide incentives.

Because IP multicast has failed in gaining widespread adoption across Internet service providers, many have designed application-level multicast protocols using overlay networks. Overcast [34] is one of the first systems to explore using

overlay networks to disseminate data. In Overcast, Jannotti et al. design heuristics to build trees optimized for high-bandwidth data dissemination.

SplitStream [13] builds a forest of trees to disseminate data. In this system, Castro et al. leverage the mechanisms in Pastry [66] and Scribe [12] to construct trees that spread the work of forwarding data evenly across participating nodes. Their techniques ensure that no node serves as an internal node in more than one tree. Ngan et al. [57] take advantage of the SplitStream design and propose a punishment scheme to discourage free-riding behavior. Their approach penalizes internal nodes who decide to not forward data by periodically restructuring the tree, allowing the descendants a selfish user may have hurt to respond in kind. This scheme quickly loses its appeal when we consider that a malicious peer can trigger frequent restructurings.

Instead of focusing on punishments, some works incentivize cooperation by offering peers who contribute increased resilience to benign failures. Habib and Chuang [31] employ a scoring system that tracks a peer's contributions. A peer with a high score has more options in choosing where to obtain data, and therefore, more resilience in case some peers crash or are unreachable. In Climber [61], Park et al. initially organize peers into a tree and then insert redundant links to a form a mesh where peers who contribute more are rewarded with more redundant links.

PULSE [63] rewards users who contribute by placing them closer to the source with respect to how data is distributed. The assumption is that a user benefits from less delay between when a live event occurs and when it appears on a user's screen. Liu et al. [47] adopt a BitTorrent-like strategy, specifying that each peer should upload to those partners who have contributed the most. This scheme also incorporates an optimistic unchoke technique, making their system vulnerable to the free-riding strategies that plague BitTorrent. In short, although many of the above works propose incentives to encourage cooperation, these approaches provide no

justification that the inherent rewards are compelling enough to induce cooperation.

Chu et al. [17] sidestep the worry that users may cheat by simply positing that users are incapable of modifying the software. They assume that a peer's behaviors are limited to changing how much to contribute, ignoring more sophisticated and subtle strategies that may indeed be better. Chu et al. focus on designing a linear taxation model that requires those with more available bandwidth to contribute more than those with less available bandwidth. Their work focuses on what the taxes should be for different upload capabilities, not how to enforce the collecting of such taxes.

Equicast [38], by Keidar et al., is the first work to formally deal with selfish users in a p2p streaming setting. They structure the system such that it is in a peer's best interest to follow the protocol, assuming that the peer can only change how many connections to maintain and how many packets to send on each connection. Although theoretically valuable, Equicast relies on a number of impractical assumptions, such as a peer being hurt infinitely much if it misses even one block of the stream. Further, Equicast is a purely theoretical work, without simulations or a prototype to assess practicality. Lin et al. [46] similarly take a formal approach, providing a game-theoretic framework in which one could analyze a p2p live streaming application. However, they leave open how to actually design a system to be a Nash equilibrium.

To our knowledge, SecureStream [32] is the only other work designed to tolerate malicious behavior in a p2p live streaming setting. Leveraging an intrusion-tolerant membership protocol, Fireflies [35], SecureStream is resilient to denial of service attacks, forgery, and several other attacks. Haridasan et al. [33] later extend the SecureStream system by augmenting it with an infrastructure to audit peers' contributions. Although effective in simulation, the resulting protocol lacks a formal analysis bounding the gains from attempting to cheat.

While there have been several works that use incentives in file-sharing and streaming settings, few approaches also deal with Byzantine behavior. The work in this dissertation draws from the contributions made by Aiyer et al. [3] in proposing the BAR model to cope with Byzantine behaviors and selfish actions. Aiyer et al. identify that existing models are ill-equipped to deal with both kinds of deviations. In the traditional Byzantine failure model, any deviation is considered a fault, meaning that system designers may have to consider situations in which nearly every participant is Byzantine—a characterization that perhaps overstates the problem. Traditional economics models are well-armed against selfish participants, but provide little relief against participants who may break incentive structures by acting irrationally. Aiyer et al. use the BAR model to design a cooperative backup system based on a replicated state machine architecture that enforces obedience while tolerating a bounded number of Byzantine failures. BAR Gossip and FlightPath adopt a very different approach by leveraging gossip's scalability to disseminate data.

Chapter 3

Background & Model

In this chapter, we provide background on relevant concepts from game theory and their analogues in distributed system design. Afterwards, we describe the setting in which we plan to stream data.

3.1 Game Theory Background

We can frame many interactions, both in real life and the online world, as games in which players participate to receive payoffs. In these interactions, game theory provides a framework to reason about strategic behavior and lets us predict the outcomes when players seek to increase their payoffs perhaps at the expense of other players. As even a basic game theory primer could consume an entire dissertation, we include only those topics relevant to understanding our contributions. Where appropriate, we relate game theoretic definitions to their analogues in distributed systems.

A game consists of a set of *players* $\mathcal{N}$, a set of *strategies* S_i for each player i, and a set of *utility functions* $\{u_1, \ldots, u_n\}$, where $n = |\mathcal{N}|$. Each player participates in the game by taking individual actions. A player's strategy determines which

actions a player takes in different situations. Since players typically interact with one another, it is useful to refer to a *strategy profile* $\mathbf{s} = (s_1, \ldots, s_n)$, identifying which strategy each player pursues. Given a profile $(s_1, \ldots, s_n)$ and a player i, we measure $i's$ payoff or its utility using $u_i(s_1, \ldots, s_n)$. Greater utilities correspond to more effective strategies. We occasionally use the notation $(\mathbf{s_{-i}}, s_i^*)$ to refer to the profile in which player i is pursuing a strategy s_i^* different from its strategy in the profile $\mathbf{s}$.

We consider games in which the participants are strategic or *rational* and seek to increase their own utility. Several contributions in game theory focus on predicting what rational players will do when presented with a game. If every player is rational and believes all other players to be rational, what strategy will each player adopt? Often, the reasoning can be confusing with loops such as "If I play strategy A, then her best response is strategy B in which case I would play C. However, she knows that I would play C if she played B, so she would play D to counter my C. Since I know that she would play D to counter my C, then my best response to D is to play strategy E, and so on." A desirable property is when such reasoning results in a stable strategy profile, or an equilibrium.

Equilibria are important in game theory and the closely related subfield, *mechanism design*. Whereas game theory focuses on predicting a game's outcome, mechanism design strives to design games to achieve a particular outcome. We declare success when we can design mechanisms so that the outcome we want is the same as the outcome that game theory predicts. However, aligning our desires with game theory's predictions is usually difficult. We may assign strategies to players to achieve an outcome. Yet, a rational player may attempt to increase its own utility by switching to a different strategy. In this case, we say a player disobeys its assigned strategy or *defects*. Our goal is to design a strategy profile that would achieve our desired outcome while also being an equilibrium, providing no incentive

for any player to defect.

Distributed system design shares many concepts with game theory and mechanism design. Nodes assume the role of players and are assigned the strategy of obeying the protocol. Therefore, a protocol specification implicitly defines a strategy profile. When a node deviates from the protocol, that action is analogous to a player defecting. We consider a protocol to be an equilibrium if the protocol defines a strategy profile that is an equilibrium. Such protocols are important in p2p systems because they define stable situations in which rational participants obey the protocol.

In this dissertation, we design p2p protocols to achieve a very specific outcome: reliably streaming live data. We craft mechanisms so that it is in every peer's interest to obey the protocol and use game theory to justify that claim. In Chapters 4 and 5, we design protocols to be a particular kind of equilibria, Nash equilibria [56] (defined in Chapter 4). In Chapter 6, we create a protocol that is an ϵ-Nash equilibria [15] (defined in Chapter 6).

3.2 System Model

We consider the problem of streaming a live event over the Internet to a set of clients (or peers). A *tracker* maintains the set of peers that subscribe to the live event. A *source* divides time into rounds that are `r_len` seconds long. In each round, the source generates stream packets that expire after `deadline` rounds. The source multicasts each packet to a small fraction `seed_frac` of random clients.

Clients work together to disseminate those packets throughout the system. When a stream packet expires, all peers that possess that packet deliver it to their media application. If a peer delivers fewer updates than it should in a round, we consider that round jittered and our goal is to minimize such rounds. We define the jitter of a live stream to be the ratio of rounds jittered to the total number of rounds.

Similarly, we define the reliability as the ratio of packets delivered on time to the total number of packets. This reliability metric is analogous to SecureStream's [32] continuity index.

We assume that the source and tracker run as specified and do not fail, although we could relax this assumption using standard techniques for fault-tolerance [67, 14]. Peers, however, may deviate from the specification.

We use the BAR model [3] to classify peer behaviors as Byzantine, altruistic, or rational. The premise of the BAR model is that when nodes can benefit by deviating, it may be untenable to bound the number of deviations to a small fraction. Thus, we desire to create systems that continue to function even if all participants are rational and willing to deviate for gain.

While many nodes behave rationally, some may be Byzantine and behave arbitrarily because of a bug, misconfiguration, or ill-will. We assume that the fraction of nodes that are Byzantine is bound by $F_{byz} < 1$. We also assume that rational peers expect negative utility from communicating with peers known to be Byzantine. Altruistic peers obey the given protocol.

Non-Byzantine peers maintain clocks synchronized with the tracker. Nodes communicate over synchronous yet unreliable channels. We assume that each peer has exactly one public key bound to a permanent id. In practice, we can discharge this assumption by using a certificate authority or by implementing recent proposals to defend against Sybil attacks [78].

We assume that cryptographic primitives—such as digital signatures, symmetric encryption, and one-way hashes—cannot be subverted. Our algorithms also require that private keys generate unique signatures [9]. We denote a message m signed by peer i as $\langle m \rangle_i$. Non-Byzantine nodes ignore messages that cannot be properly authenticated.

Finally, we hold peers accountable for the messages they send. We define a

proof of misbehavior (PoM) as a signed message that proves a peer has deviated from the protocol. A PoM against a peer is sufficient evidence for the source and tracker to evict a peer from the system, never letting that peer join a streaming session with that tracker or source in the future. We assume that eviction is a sufficient penalty to deter any rational peer from sending a message that the receiver could present as a PoM.

Chapter 4

Balanced Exchange: A Purist Approach

In this chapter, we present Balanced Exchange, the first gossip protocol designed for a setting in which some peers may be Byzantine while the remaining can be rational. Despite this adversarial environment, Balanced Exchange provides a mechanism to disseminate information that ensures steady throughput.

The defining characteristic of gossip protocols is that each peer exchanges data, or gossips, with randomly selected peers. It is precisely this randomness that gives gossip protocols their enviable robustness, a trait that we want to maintain in Balanced Exchange. However, from the perspective of designing protocols in a Byzantine and rational setting, randomness can be a headache: in fact, *any* source of non-determinism is hard to deal with because it gives opportunities for rational users to hide deviations in the guise of legitimate non-deterministic behavior.

In Balanced Exchange, we overcome this difficulty by using *verifiable randomness* [53]. In particular, we leverage the properties of pseudo-random number generators and unique signature schemes to build a verifiable pseudo-random algorithm. Although our algorithm does not provide the theoretical properties of

verifiable random functions as seen in the number theory literature, it performs well in practice. Furthermore, to our knowledge, there exists no implementation of those more sophisticated schemes, while our algorithm is easily implemented given existing cryptographic libraries. Our verifiable pseudo-random algorithm eliminates the main source of non-determinism in traditional gossip—randomness in partner selection—yet maintains the unpredictability and rapid convergence of traditional gossip. In combination with a simple *fair enough exchange* primitive, our partner selection algorithm is effective in encouraging peers to trade updates with one another.

In the remainder of this chapter, we describe the design of Balanced Exchange and prove that it is a Nash equilibrium. At the end of this chapter, we include simulation results indicating that, despite its theoretical appeal, Balanced Exchange falls short of our practical goals. In the next chapter, we introduce Optimistic Push to address these shortcomings.

4.1 Assumptions

We model a live event as an infinite game. Rational peers are interested in acquring stream updates that will ultimatley be delivered to their media applications. We assume that a rational peer's benefit increases proportionally to the number of unique stream updates it acquires. A rational peer's cost increases according to how much upload and download bandwidth that peer uses. Each unit of upload and download bandwidth consumed costs a peer c_u and c_d, respectively.

Because we only consider live events that have no end times, we can ignore end game strategies that rational peers may employ. Such strategies seek to cheat at the end of a game because doing so is less risky than cheating at the beginning or middle.

Additionally, rational peers never risk eviction; they never send a proof of

misbehavior (PoM), which could result in their own eviction from the system. In our model, an evicted rational peer is actually a contradiction. Recall that it is rational to act in ways to increase utility (calculated as benefits minus costs). Yet, an evicted peer stops receiving stream updates, and in the limit receives 0 benefit.

We curb rational deviations in our cooperative system, by designing Balanced Exchange to be an ex ante Nash equilibrium [56].

Definition 1 *A protocol is an* ex ante Nash equilibrium *if it defines a strategy profile such that each rational player expects no gain from unilaterally deviating. More formally, a strategy profile* $\mathbf{s} = (s_0, \ldots, s_n)$ *is an ex ante Nash equilibrium if for every player* i *there exists no strategy* s_i^* *such that* i *expects utility* $u_i(\mathbf{s}_{-i}, s_i^*) > u_i(\mathbf{s})$ *from following* s_i^**.*

In the rest of this dissertation, we elide 'ex ante' for brevity. In the framework of Nash equilibria, we assume that rational peers only consider strategies that maximize the number of useful updates received in each exchange independent of concurrent or future exchanges. This greedy strategy is reasonable in a streaming setting where there is a limited amount of time to obtain useful updates. However, it is possible that more sophisticated strategies optimizing over multiple exchanges can achieve greater utility. We also assume that it is always in a rational peer's interest to participate in exchanges provided it is missing at least one update.

4.2 Design

The Balanced Exchange protocol describes a method for a source to send a stream of data to a set of clients. Streaming a live event requires Balanced Exchange to ensure that clients who obey the protocol deliver only authentic stream packets and that such clients receive nearly all stream packets in a timely manner. Although we can provide the first property in all situations, the second property is elusive,

and will require the addition of the Optimistic Push protocol described in the next chapter.

Before the start of a live event, each peer generates a session key pair consisting of a public and private key. Peers sign up for the event by divulging *both* keys to the tracker. The tracker then verifies the keys, closes the sign up service, and posts a list that contains each peer's identity, address, and public key. Peers sign protocol messages using their private keys to provide authentication, integrity, and non-repudiation of message contents.

During a live event, the source divides the stream into discrete fixed-size *stream updates*. In each round, the source multicasts each of `ups_per_round` updates to a fraction `seed_frac` of random clients. The source signs each of these updates so that clients can verify the stream data's authenticity.

Since each peer is unlikely to receive every update directly from the source, peers rely on Balanced Exchange to garner the remaining. The Balanced Exchange protocol allows clients to trade equal numbers of updates with one another. That is, if client S has ten updates to offer client R and R has only five to offer in return, then S and R trade five updates in each direction. We construct each balanced exchange so that it is a *Nash equilibrium*, thereby motivating rational clients to conduct the trade faithfully. For reference, Figure 4.1 illustrates a balanced exchange between two peers.

4.2.1 Overview

Balanced Exchange provides a mechanism for peers to exchange updates without worrying that rational peers will cheat their trading partners. In a balanced exchange, each party determines the largest number of new updates it can trade on a one-for-one basis. While a client initiates an exchange with another client, it also responds to Balanced Exchange requests. An exchange consists of four phases.

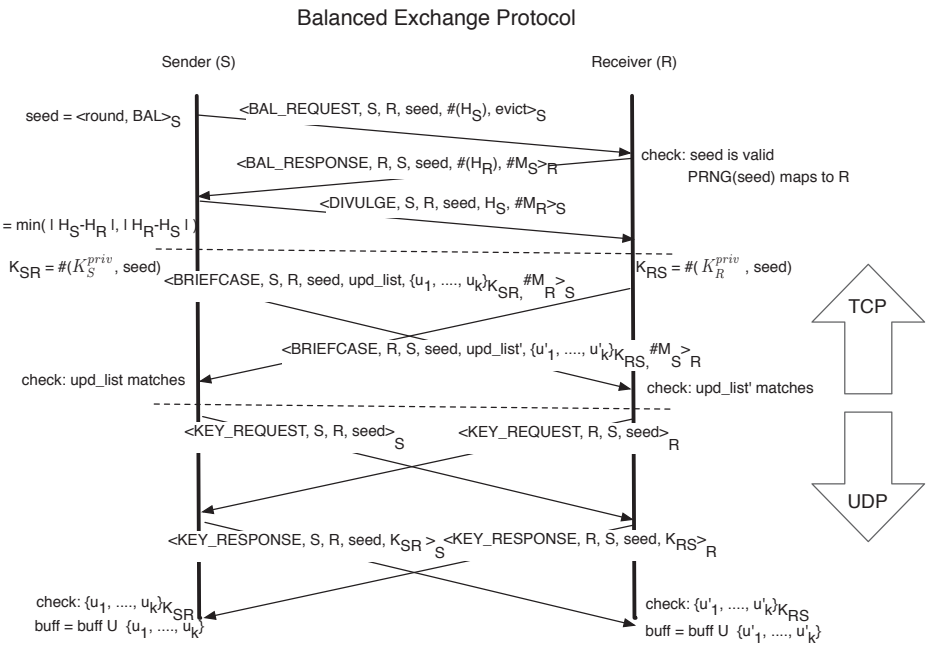

Figure 4.1: Balanced Exchange Protocol for client S contacting client R.

Partner Selection: A peer selects another peer with whom to trade using a verifiable pseudo-random algorithm.

History Exchange: The two parties learn about the unexpired updates the other party holds and determines the largest number k of updates that can be exchanged equally.

Update Exchange Each party deterministically generates an encryption key based upon its private key and a per-exchange seed value. Each party then encrypts its k most recent exchangeable updates and copies the encrypted updates into a *briefcase* that is sent to the other party.

Key Exchange The parties swap keys and decrypt the contents of received briefcases. If a peer does not receive a decryption key in a timely manner, that peer sends a key request to its partner. After decrypting updates, peers update

their histories to reflect the new updates they have just received.

A peer ends an exchange early in the history exchange phase if that peer realizes the exchange will ultimately trade no updates. An exchange *completes* if both clients execute all four phases or if one of them ends the exchange early as allowed by the protocol. Peers use TCP to send packets reliably during the first three phases and switch to UDP for the fourth. We discuss this design decision later when we prove that each balanced exchange is a Nash equilibrium.

In each exchange, we discourage cheating by designing the protocol so that peers who blatantly cheat are caught and evicted from the system. If the tracker sees a *proof of misbehavior* (PoM) against a peer, the tracker removes that peer from the system. A PoM is a sequence of inconsistent messages signed by a peer. We intertwine the history and update exchange phases so that blatant cheating efforts generate PoMs. Each message in our history and update exchanges includes a cryptographic hash of the previous message sent in the trade: if a client sends a briefcase whose contents differ from the agreed upon updates of the history exchange, then the history exchange messages plus the briefcase constitutes a PoM. These cryptographic hashes are missing from messages in the key exchange phase. We later explain this omission.

We use the tracker to audit possible PoMs. In every round, the tracker polices the system by ordering a fraction of random peers to supply suspected PoMs against other peers. If a queried peer does not have a suspected PoM against another peer, then it replies with a dummy message. We specify all audit responses to be of equal size, thereby removing a rational peer's incentive to cover up PoMs against others. The tracker treats peers that ignore audit requests as it would peers that have provably misbehaved. To reduce false positives because of transient network failures, the tracker allows sufficient time for a peer to respond to audit requests. The auditor evicts a misbehaving peer by sending a signed eviction notice to the

broadcaster who embeds all eviction notices in every update and stops sending updates to evicted peers. We discuss in Section 5.1.2 how to bound the overhead of eviction notices.

In the next section, we prove that each balanced exchange is a Nash equilibrium. Our approach to provide this property follows two principles: *delayed gratification* and *restricted choice*. We delay gratification to postponse a peer's receipt of useful updates until the last phase, thereby forcing rational peers to participate until the end of a trade. We then restrict choice so that such participation follows the protocol specification. In the next section, we discuss the Balanced Exchange Protocol in greater detail.

4.2.2 Details

To review, in a protocol that is a Nash equilibrium, rational peers expect no gain from unilateral deviations. As stated earlier, we assume rational peers only consider strategies that maximize the number of useful updates received in each exchange independent of concurrent or future exchanges. The Balanced Exchange Protocol guarantees the following property:

Theorem 1 *If two rational clients participating in a balanced exchange with each other seek to maximize the utility of that exchange independent of concurrent or future exchanges, then the Balanced Exchange Protocol is a Nash equilibrium.*

In the following sections, we show that each phase of a balanced exchange is a Nash equilibrium, implying that the whole balanced exchange is also a Nash equilibrium. In each phase, we show that if a rational peer assumes its peers obey the protocol, then following the Balanced Exchange protocol is in that peer's best interest. Note that the assumption that remaining peers obey the protocol is an artifact of the Nash equilibria proof technique and is not a requirement of our protocol.

We simplify the presentation of the subsequent lemmas and proofs by treating any client that has issued a POM against itself as evicted. We use the following property regarding eviction in later proofs and include it here for reference.

Lemma 1 *A rational peer S only pursues strategies that ignore evicted peers.*

Proof:

1. Suppose for contradiction that there exists an evicted peer R that S does not ignore.
2. S participates in an exchange with R only if S expects positive utility from the interaction.
3. Because R has already been evicted, R is Byzantine, as it can be neither altruistic nor rational
4. S therefore expects negative utility from communicating with R.
5. This reasoning brings the proof to its contradiction in that S expects both positive and negative utility from communicating with R.
6. Therefore, S ignores all evicted peers.

Phase 1: Partner Selection

The first phase, partner selection, highlights a fundamental difference between traditional gossip and gossip in a BAR setting. In a traditional gossip protocol, each peer periodically selects a partner using a pseudo-random number generator (PRNG) and contacts that partner to request an exchange. Each peer also accepts every request it receives. Random partner selection provides robustness against crashed peers and link failures. Yet, in a BAR setting, the freedom to choose partners allows rational peers to select partners not at random, possibly dissolving gossip's robustness.

This vulnerability stems from the non-determinism inherent to random partner selection. Furthermore, malicious peers could attempt to monopolize all gossiping opportunities.

We eliminate this weakness by replacing the non-deterministic seed, typically system time, used in PRNGs with a per round deterministic yet unpredictable value. A peer S uses its signature $\langle r, \text{BAL} \rangle_S$ as its seed for round r, where BAL is the string "BAL". With this seed, S then deterministically maps numbers generated by the PRNG to entries in the membership list. S continues looking for entries until it finds the first non-evicted partner R. This partner selection is deterministic, but unpredictable because no peer other than S can generate $S's$ signature for a seed value. As Figure 4.1 illustrates, to initiate a gossip request to R, S includes the seed and all eviction notices for peers that S could have selected before deciding on R. R then determines whether the exchange request is *valid*.

Definition 2 *A balanced exchange request* $\langle BAL\_REQUEST, S, R, seed, \#(H_S), evict \rangle_S$ *is* valid *if and only it it satisfies the following conditions:*

- *The seed is S's signature of* $\langle r, BAL \rangle$*.*
- *All included eviction notices are properly signed by the tracker.*
- *The seeded PRNG selects R as the first non-evicted peer.*

Note that invalid balanced exchange requests are also PoMs. R accepts a balanced exchange request if the request is valid, the seed is for the current round, and this is the first time that S has presented this seed value to R. Otherwise, R ignores the request.

Lemma 2 *Rational peers only send gossip requests to and accept gossip requests from peers as determined by valid requests.*

Proof:

1. A rational peer S may communicate with the targets as sanctioned by the protocol, an evicted peer, or some other peer.

2. By Lemma 1, S does not send balanced exchange requests to evicted peers.

3. S only sends balanced exchange requests to partners as sanctioned by the protocol, as it expects other peers to reject inappropriate requests.

4. S rejects balanced exchange requests from peers not sanctioned by the protocol, as responding to the request would generate a PoM against S.

5. S therefore only communicates with targets as sanctioned by the protocol.

We note that the argument against peers participating in unsanctioned exchanges is buttressed by the specific tangible concern that such exchanges would be done without the recourse of sending a PoM to the auditor if either peer in an exchange were (quite rationally) to cheat its partner by sending a different briefcase than the one agreed upon.

Phase 2: History Exchange

After a peer S selects a partner R, they exchange histories—a history defines a set of update ids—using three messages. As Figure 4.1 illustrates, S provides in the first message a hash of its history H_S and the PRNG seed value (as discussed earlier) to R; the hash commits S to send H_S to R. After R accepts S's request to trade updates, R returns its current history H_R. In the final message, S divulges its actual history, H_S, to R who checks that H_S is consistent with the previously sent hash. Note that each peer sends a history before learning its partner's history: S does so by sending a hash first and R by sending its actual history while possessing only an irreversible hash. This design makes it difficult for a Byzantine peer to maximize network traffic during the update exchange by tailoring a history to its partner's, e.g., by responding with a history that is the exact complement of its partner's.

Lemma 3 *A rational peer S does not divulge a history that does not match the original hash.*

Proof:

1. Suppose S sent a signed message that divulged a history whose hash does not match the one sent in an exchange request.
2. Together, the request and divulge messages constitute a PoM.
3. Rational peers do not sent PoMs.

Lemma 4 *A rational peer S terminates any exchange with a peer R who divulges a history whose hash does not match the original sent in a request message.*

Proof:

1. Suppose S does not terminate the exchange.
2. Because R sends these internally inconsistent messages, R generates a PoM against itself.
3. S considers R to be an evicted peer since R has generated a PoM against itself.
4. S does not continue communicating with evicted peers.

The restrictions we have imposed on history exchanges forces rational peers to, in essence, obey the letter of the law. The spirit, however, is that rational peers report their histories honestly. We now discuss the incentives for a rational peer to be honest in claiming which updates it has and does not have. Our first technique draws from the principle of *balanced cost* [3]. We define all histories to be of equal size, thereby removing any incentive S may have to save a few bytes by sending smaller histories. Therefore, when exchanging histories, the only remaining way for

S to obtain greater utility is by increasing the number of useful updates received by the end of the exchange.

We show that a rational peer S has no incentive to lie about its history. The proof relies on two lemmas. The first lemma states that S should not lie about missing any updates. The second states that any situation in which S benefits from claiming to have more than it actually does also requires S to risk eviction. Together, these two lemmas imply that S should be honest in reporting its history.

Lemma 5 *Consider a rational peer S, its history H_S, and an unexpired update $U \in H_S$. If S participates in an exchange, then S expects nothing to gain from claiming to not possess U, i.e., reporting a history H'_S such that $U \notin H'_S$.*

Proof:

1. Recall that the number of updates to be exchanged k is the minimum of how many updates each party reports that it has that the other reports it does not.
2. Suppose that S reports H'_S as its history.
3. We consider two cases: S's partner either possesses U or does not.
4. If S's partner does not have U, then S can only keep k the same or increase k by claiming to have U.
5. If S's partner does have U, then it is possible that S can increase k by 1
6. However, if k increases, it means that S would expect to receive U again

Lemma 6 *Consider a rational peer S, its history H_S, and an unexpired update $U \notin H_S$. Any exchange in which S benefits from reporting a history H'_S, such that $U \in H'_S$, requires S to upload U.*

Proof:

1. Again, k is the minimum number of how many updates each party reports that it has that the other does not.
2. Suppose that S reports a H'_S as its history
3. We consider two cases: S's partner either possesses U or does not.
4. If S's partner possesses U, then by being honest and not claiming to have U, S either increases k or leaves k unchanged.
5. If S's partner does not possess U, then by being honest, S risks decreasing k by 1.
6. Note that k is decreased only if S would be obligated to upload U as part of the exchange.
7. However, S does not possess U and claiming to upload it in a briefcase would generate a PoM, something that rational peers do not do.

Phase 3: Update Exchange

After the history exchange commits S and R to sending the k most recent updates each possesses but the other lacks, S and R send the corresponding updates in signed briefcase messages. Each briefcase contains *i)* the seed identifying this exchange, *ii)* a plaintext description of k update ids, and *iii)* the corresponding k updates encrypted with the hash of both the sender's private key and the exchange's seed value. The sender signs the briefcase, promising that the encrypted contents match the description. If either the received briefcase's seed value does not match the seed identifying this exchange or the briefcase's update list does not match the k expected updates, the receiver aborts the exchange without sending its decryption key.

Lemma 7 *Rational peers do not send briefcases that contain inappropriate seed values or inappropriate plaintext descriptions.*

Proof:

1. Suppose that a rational peer S includes either the wrong seed value or the wrong plaintext description in a briefcase message.
2. S expects its partner to reject the briefcase and abandon the exchange.
3. S's action is therefore not rational.

Lemma 8 *Rational peers reject briefcases that contain inappropriate seed values or inappropriate plaintext descriptions.*

Proof:

1. Suppose that a rational peer S accepts a briefcase with either a wrong seed value or wrong plaintext description.
2. Such a briefcase along with the previous history messages constitutes a PoM against the sender of the briefcase.
3. S therefore considers its partner an evicted peer.
4. S does not communicate with evicted peers.

Lemma 9 *If a rational peer S sends a briefcase message, then the encrypted contents correspond to the briefcase's plaintext description.*

Proof:

1. Suppose that a rational peer S sends a briefcase whose contents differ from the plaintext description.
2. Such a briefcase constitutes a PoM.
3. Yet, S is rational and does not send PoMs.

Key Exchange

A peer who is satisfied with its partner's briefcase enters the key exchange phase. In this phase, the peer sends via UDP a key request that contains the seed value used to initiate the exchange. A peer replies to a key request with a response that contains that peer's seed-specific decryption key.

As redundant as it may seem to read, the goal of the key exchange phase is for peers to actually exchange keys. There should be no expected benefit for a rational peer to withold its key. However, as with the exchange of any items, guaranteeing fairness requires a trusted third party [25, 59], where fairness means that either both peers in a balanced exchange obtain what they are seeking or neither peer does.

Introducing a third party to mediate potentially every exchange can be a bottleneck at large-scales. We therefore design a mechanism that allows altruistic and rational peers to trade keys without concern that these peers will attempt to cheat. The linchpin in providing the incentives for exchanging key is to use a *credible threat.* A peer repeatedly sends key requests, up to some constant number of times, until it obtains a key response from its partner. Note that it is possible to tune the size of key requests to offset any asymmetry between download and upload capacity.

Lemma 10 *If a rational peer S replies to a key request, then S's response contains the appropriate symmetric key.*

Proof:

1. Suppose that S sends a key response with an inappropriate key.
2. S's partner discards the response and acts as if S never sent the message.
3. Therefore, S wasted the effort in sending its response, an action that rational peers do not indulge.

Lemma 11 *If a rational peer S does not receive a key response from its partner R, S resends its key request.*

Proof:

1. Suppose S does not send the key request.
2. S then does not expect to get a key to decrypt updates received from R, and gets no benefit from this exchange.
3. Recall that S is rational.
4. If S were to resend the request, S would expect to get a response, decrypt updates, and benefit from the exchange.
5. S therefore sends the request.

The final lemma states that S will indeed respond to key responses from R. The proof relies on the unpredictability of exchanging messages over an unreliable channel. By using UDP to exchange keys, we take advantage of $S's$ uncertainty in guessing whether R will send further key requests. More concretely, if S is certain with probability p that R is finished sending key requests, then $R's$ key requests should be at least a factor of $\frac{c_u}{c_d(1-p)}$ as large as key responses, where c_u and c_d are the costs of uploading and downloading a single byte, respectively. In our implementation, we assume that S always believes another key request will arrive, $p = 0$.

Lemma 12 *If S is certain with probability p that R has sent its last key request, then S replies to $R's$ requests provided that those requests are more than a factor of $\frac{c_u}{c_d(1-p)}$ as large as key responses.*

Proof:

1. Suppose S ignores a key request from R and therefore saves the cost, $c_u \times$ `response-size`, of sending a key response.

2. S expects to receive a key request from R with probability $1-p$, which would cost $c_d \times$ `request-size` to download.

3. S therefore expects to incur $(1-p)(c_d \times$ `request-size`$)$ cost from ignoring R's earlier request, which is greater than $S's$ savings of $c_u \times$ `response-size`, as requests are more than a factor of $\frac{c_u}{c_d(1-p)}$ as large as response.

4. Recall that S is rational and deviates only if doing so increases expected utility.

5. Therefore, it is irrational for S to ignore $R's$ key request.

4.3 Discussion

The Balanced Exchange protocol is the result of our efforts to create a protocol for which cooperation is in a rational peer's best interest. To enforce that cooperation and retain the robustness inherent to gossip protocols, we design a verifiable pseudo-random partner selection algorithm and a novel fair enough exchange mechanism. These contributions let us construct a gossip protocol founded on an appealing solution concept, Nash equilibria.

However, Balanced Exchange's theoretical attractiveness is diluted by its practical shortcomings. We simulate 500 peers using Balanced Exchange to trade updates. In our simulation, the broadcaster sends 50 updates to 25 random peers in every round. Each update expires 10 rounds from the round in which it was sent. In Figure 4.2, we plot the average percentage of updates a peer misses and show that peers miss over 1% of updates sent by a broadcaster[1]. This small deficit is a

[1]Additionally, we see that reliability improves with time and levels off after approximately 30 rounds. This improvement is because there is more diversity in updates for later rounds, resulting in more effective bilateral trades.

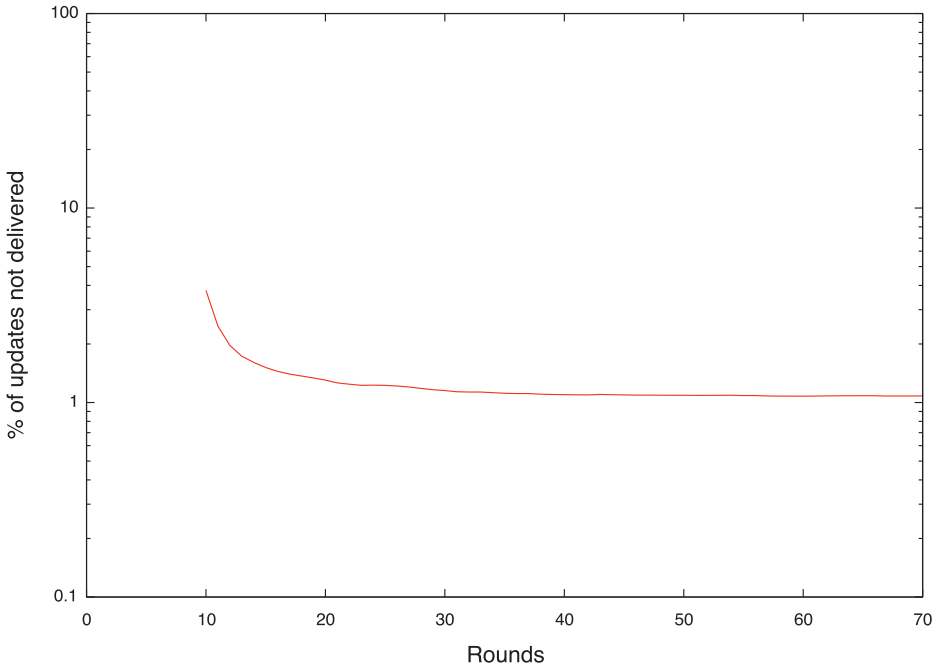

Figure 4.2: Percentage of updates missed on a round by round basis. Note that on this graph we use a log-scale y-axis and that a curve closer to the x-axis is desirable.

consequence of restricting data dissemination to exactly balanced trades. Although small, this unreliability introduces several noticeable artifacts in a video stream. In the next chapter, we explore how to augment the Balanced Exchange protocol to address this performance concern.

Chapter 5

BAR Gossip: Making it Work

In this chapter, we augment Balanced Exchange to yield BAR Gossip, the first p2p live streaming system that tolerates both Byzantine and rational peers. BAR Gossip consists of two protocols: Balanced Exchange and Optimistic Push. In the previous chapter, we presented Balanced Exchange, which enables rational peers to trade updates with one another in a provably fair manner. However, while fairness is desirable, Balanced Exchange achieves it while sacrificing performance (see Figure 4.2). BAR Gossip's second half, Optimistic Push, addresses Balanced Exchange's practical shortcomings.

This second protocol offers another avenue for peers to trade updates. The main difference between the two protocols is that an optimistic push lets a peer trade updates with the hope, rather than the certainty, that a peer will receive an equal number of updates in return. Although we show that optimistic pushes address practical shortcomings, they have a theoretical drawback: We cannot *prove* that the Optimistic Push protocol is a Nash equilibrium, although our experiments suggest it is not easy to cheat.

In our evaluation, we demonstrate that Balanced Exchange and Optimistic Push can together achieve 99.9% reliability, surpassing Balanced Exchange's 98.7%.

We also assess how robust Optimistic Push is to attempts at cheating. Finally, we examine how resilient BAR Gossip is to colluding and Byzantine behaviors.

5.1 Design

The Balanced Exchange protocol's strength is also its weakness. We crafted each balanced exchange so that a non-Byzantine peer could trade a set of updates for an equal number of updates. This design works well if peers can consistently find partners who have the right set of updates—an exact complement is ideal. If a peer has little to offer in an exchange with its partner, then few updates would be traded.

Although uncommon, these occurrences make it more likely that a peer will fail to deliver updates before they expire. Worse, when a peer begins to fall behind in acquiring updates, it becomes harder for that peer to get caught up since every exchange is balanced. This observation suggests that a second way to trade updates may be useful, one that allows peers that have fallen behind to quickly obtain missing updates than can be used in future trades.

5.1.1 Optimistic Push

The Optimistic Push Protocol provides a safety net for peers to acquire missing updates without giving back a set of updates of equivalent value. Optimistic pushes follow the same structure as balanced exchanges; we provide Figure 5.1 for reference. Partner selection is nearly identical. In round r, peer S uses $\langle r, \text{OPT} \rangle_S$ to seed the PRNG and selects a partner R in the same way as in the Balanced Exchange protocol.

The main difference between Balanced Exchange and Optimistic Push lies in what the parties disclose to each other during the history exchange phase and in how they determine the content of their respective briefcases during the update exchange phase. To exchange histories, S sends to R two lists: a *young list* and an *old*

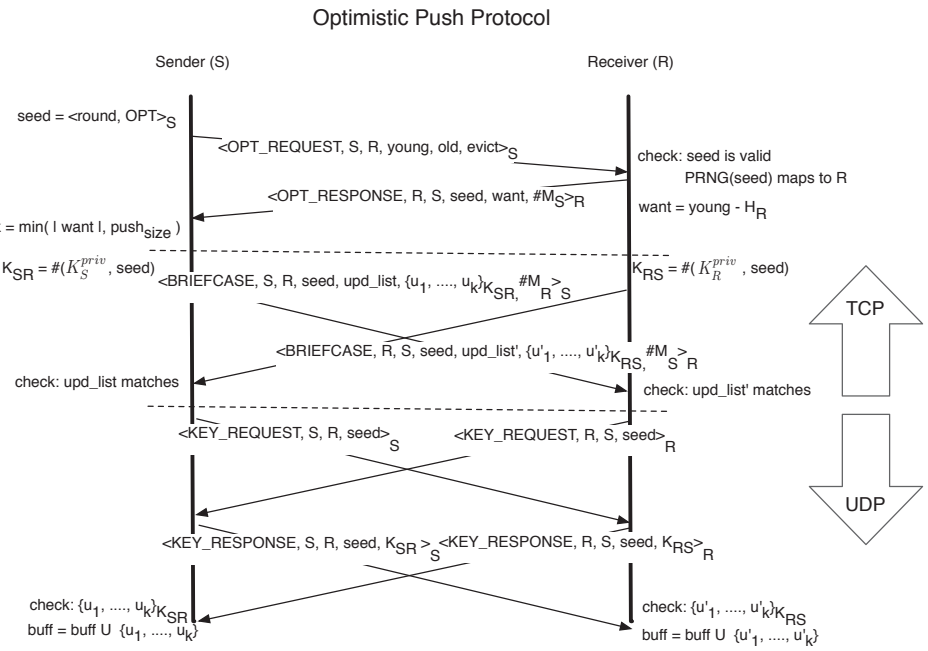

Figure 5.1: Optimistic Push Protocol for client S contacting client R.

list. The young list identifies the most recent updates that S possesses. The old list identifies updates that S still needs and that are about to expire. When S initiates an optimistic push, S hopes to obtain updates from its old list in exchange for uploading updates from the young list. This hope is precisely what makes optimistic pushes *optimistic*. $S's$ partner, R, has the option to upload fewer stream updates from the old list than it requests from the young list. It is this flexibility to obtain stream updates while uploading (possibly) fewer stream updates in return that allows a peer who may have fallen behind to quickly catch up with the rest of the system.

More concretely, after R receives the young and old lists from S, R replies with a *want* list that contains update identifiers from the young list that R is missing. S and R then exchange briefcases. $S's$ briefcase contains the updates identified by the want list along with an appropriate plaintext description of the update ids. In contrast, R includes in its briefcase as many updates from the old list as possible

without exceeding the number of updates R wants from the young list. If R has not reached the limit of how many updates it can include in the briefcase, it makes up the difference with *junk* updates. Furthermore, the plaintext description on $R's$ briefcase does not identify the exact contained updates, only how many items are inside, each of which can either be junk or from the *old list*.

We emphasize that junk updates are crafted to be larger than stream updates. If junk updates were smaller, Optimistic Push would encourage rational peers to deviate from Balanced Exchange because updates might be had for cheaper in Optimistic Push.

We regulate Optimistic Push with two parameters, `push_age` and `push_size`: the *young list* consists only of update ids that have been broadcast within the last `push_age` rounds and `push_size` is an upper limit on the number of updates that the receiver can place in its *want list*. Larger values of `push_size` help lagging peers catch up faster; however, they also increase the likelihood that such peers will waste bandwidth by sending junk.

The Optimistic Push Protocol follows nearly the same steps as the Balanced Exchange Protocol. Peers select partners in a verifiable and pseudo-random manner, exchange histories, swap encrypted updates, and divulge decryption keys. Like Balanced Exchange, this trading structure greatly limits the ways in which a rational peer could cheat. Unlike Balanced Exchange, the extra flexbility afforded by junk updates makes faithful participation less certain. For example, a rational peer may disingenuously claim to be missing an update so as to reduce the expected number of received junk updates. Worse, rational peers may choose to deviate from the Optimistic Push protocol by simply not participating, never initiating pushes but responding to them, or sending junk updates in lieu of useful updates.

We have been unable to prove whether a rational peer would obey or disobey the Optimistic Push protocol. The obstacle is the difficulty in accurately model-

ing what a rational peer expects in a setting involving hundreds of peers making pseudo-random choices to trade hundreds of updates. Traditionally, researchers have taken approaches to simplify the analysis by only considering how a single update spreads [10, 23]. In BAR Gossip though, how one update spreads is dependent on how other updates spread; updates are *traded*. At the end of the previous chapter, Figure 4.2 illustrates how these interdependencies affect Balanced Exchange. In the absence of a theoretical proof for obedience, we turn to more empirical methods. Later in this chapter, we include experimental evidence which suggests that despite the gains possible from cheating, realizing those gains is not easy.

5.1.2 Optimizations

We now describe four optimizations that increase the practicality of BAR Gossip. Our optimizations focus on limiting the bandwidth used by the protocol as so far explained.

1. We cap the number of balanced exchanges and optimistic pushes that a peer accepts in any round. As with all gossip protocols, random selection distributes load across participating peers. Occasionally, however, that randomness may overwhelm some peers that by chance are selected by many peers at once. We use the standard heuristic that each peer accepts requests up to some per round maximum and ignores further requests that round [10, 21].

2. We reign in bandwidth spikes by allowing each peer to limit the number of updates that are actually swapped in a balanced exchange. This heuristic is similar to the Round Retransmission Limit technique proposed in Bimodal Multicast [10]. A peer limits the number of updates traded in balanced exchanges by including a cap when reporting its history. The actual number of updates exchanged is therefore the minimum of each peer's reported cap and the most number of updates that can be traded in a balanced manner.

3. We limit the overhead of eviction notices by having the broadcaster embed each notice into a constant number of updates. More concretely, when the broadcaster learns of an eviction, it embeds a notice in one update every round for $r' > 0$ rounds. With high probability, every peer learns of an eviction within `deadline` $+r'$ rounds.

4. We reduce overhead further by letting peers elide old eviction notices from gossip requests. An eviction notice is old if the peer it refers to was evicted more than `deadline`$+r'$ rounds ago. Since every peer possesses each old eviction notice with high probability, peers can verify whether others select partners appropriately.

5.2 Evaluation

BAR Gossip is a robust p2p streaming protocol capable of providing stable and reliable throughput. We evaluate BAR Gossip through experiments and simulations, denoting figures derived from simulation data with '[sim]'. Our evaluation consists of four parts.

1. We examine how rational activity hurts traditional gossip protocols so as to understand BAR Gossip's impact of curbing rational deviations.

2. We explore ways that a rational peer may attempt to cheat in BAR Gossip and show BAR Gossip discourages such attempts.

3. We demonstrate BAR Gossip is robust to rational peers who may collude.

4. We show BAR Gossip tolerates up to 20% of peers being Byzantine.

Protocol Parameter	Simulation	Prototype
`ups_per_round` (updates)	100	98-101
`deadline` (rounds)	10	10
`push_size` (updates)	20	20
`push_age` (updates)	3	3
`junk_cost`	1.39	1.39
`seed_frac`	5%	5%
# Peers	500	45

Table 5.1: Parameter settings used in simulations and prototype experiments.

5.2.1 Methodology

Several parameters regulate the Balanced Exchange and Optimistic Push Protocols. The broadcaster multicasts `ups_per_round` updates per round and sends each update to a fraction `seed_frac` of random peers. Each update expires `deadline` rounds after it was multicast. In optimistic pushes, `push_age` denotes the maximum age of updates sent in the young list, while `push_size` is the maximum length of the want list. The ratio of junk update size to real update size is `junk_cost`> 1. Table 6.4.2 provides the values for these parameters for our simulation and prototype experiments.

For our prototype evaluations, we implement BAR Gossip in Python to stream an MPEG-4 video [65]. We recorded a 200 Kbps UDP video stream at 30 frames per second using Quicktime Broadcaster with one key frame every 60 frames. Quicktime Broadcaster generates UDP datagrams for the broadcast with an average size of 179 bytes ($\sigma = 62$), resulting in 116–131 datagrams per second.

Our broadcaster, auditor, and peers are a mix of 45 600 MHz and 850 MHz Emulab machines sharing a 100 Mbps Ethernet subnet, configured with a 100ms end-to-end latency and 1% probability of any packet being dropped. The broadcaster reads the recorded video from disk, encapsulates two to three Quicktime UDP datagrams into an update, pads every update to the same size (640 bytes),

and unicasts each update using UDP to three 3 random peers. Peers then exchange updates as in Figures 4.1 and 5.1. A peer delivers an update by extracting the contained datagrams and sending them to the local Quicktime player that displays the video content. We use MD5 to compute cryptographic hashes, 128 bit RSA keys with full domain hashing [8] to create unique signatures and the Mersenne Twister algorithm [51] to generate pseudo-random numbers. Each peer accepts at most 6 requests for balanced exchanges and optimistic pushes. Additionally, each peer limits to 100 the number of updates it uploads in balanced exchanges for any given round, parceling 50 to the first balanced exchange in which it participates, 25 to the next, and so on.

In the following sections we measure the reliability (expressed as the percentage of updates received by the deadline), jitter (measured as the percentage of rounds in which any update missed its deadline), and bandwidth characteristics of BAR Gossip. Unless otherwise noted, measurements in simulations are averaged over 100 rounds and using the prototype are averaged over 180 rounds across 15 trials.

5.2.2 Traditional Gossip

We now compare BAR Gossip against two traditional gossip protocols, push-pull and pull-only. The push-pull protocol is very similar to Balanced Exchange: a peer randomly selects a partner in every round, exchanges histories, and then swaps updates. The key difference is that peers swaps *all* updates that either peer possesses but for which the other peer does not. The peer who initiates the gossip exchange pushes updates to its partners and pulls updates from its partner. As its name implies, a pull-only gossip protocol sends updates in only one direction, towards the peer who initiates the exchange.

While pushing and pulling may disseminate data faster than just pulling,

the pull-only protocol is more efficient. A peer using the pull-only protocol receives updates from exactly one source in every round, compared to multiple sources in the push-pull gossip protocol. The pull-only version thereby wastes less bandwidth by never having peers receive redundant updates. Several researchers have taken advantage of the efficiency inherent to pull-only gossip techniques in building p2p live streaming systems [33, 60].

We now demonstrate that these gossip protocols, although robust in the settings for which they were designed, are ill-suited for a BAR environment. In particular, we examine each protocol's reliability and bandwidth usage in the presence of rational peers who choose to not upload any data to their partners. Note that this evaluation is conservative, as there is nothing that prevents a rational peer from initiating many more exchanges than normal to obtain even more updates for free. Furthermore, a single malicious peer could severely impair either a push-pull or pull-only protocol by monopolizing all trading opportunities. In the following simulations, we do not incorporate optimizations into BAR Gossip to reduce bandwidth that are present in the prototype.

Figures 5.2 and 5.3 plot the reliability seen and upload bandwidth used, respectively, by an altruistic peer as the proportion of rational peers in the system increases. While BAR Gossip's lines remain relatively constant in both graphs, push-pull's and pull-only's lines degrade noticeably.

Figure 5.2 shows that the push-pull protocol achieves comparable reliability to BAR Gossip when up to approximately 50% of the peers in the system are rational. Afterwards, push-pull's reliability drops dramatically. The pull-only protocol behaves similarly, maintaining near perfect reliability until approximately 30% of the peers are rational. However, the robustness seen in push-pull and push-only does not come for free.

Both push-pull and pull-only tolerate rational activity by shifting the burden

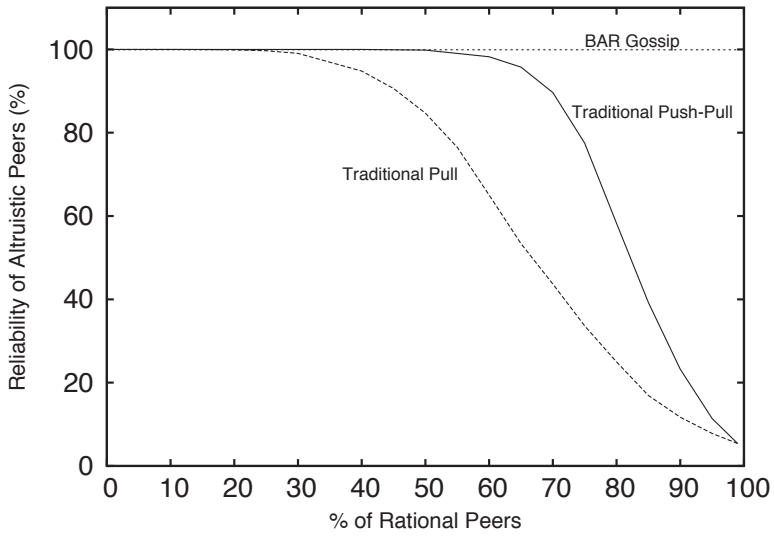

Figure 5.2: [sim] Reliability experienced by an altruistic peer using a traditional push-pull gossip protocol, a pull-only gossip protocol, and BAR Gossip.

to altruistic peers. Figure 5.3 shows the average upload bandwidth of altruistic peers rising as the fraction of rational peers also increases. The decline in bandwidth occurs when altruistic peers obtain fewer and fewer updates to spread since rational peers are neglecting to spread any updates.

Note that the rise in bandwidth experienced by altruistic peers in traditional gossip represents dangerous negative reinforcement: as more rational peers choose to cheat, the remaining altruistic peers are punished with increased bandwidth load, encouraging them also to defect until reliability collapses. BAR Gossip exhibits robustness to rational behavior with steady reliability and bandwidth measurements.

Although BAR Gossip clearly outperforms both push-pull and push-only when rational behavior is widespread, it may be worthwhile to consider when traditional gossip techniques may be preferred over BAR Gossip. Figures 5.2 and 5.3

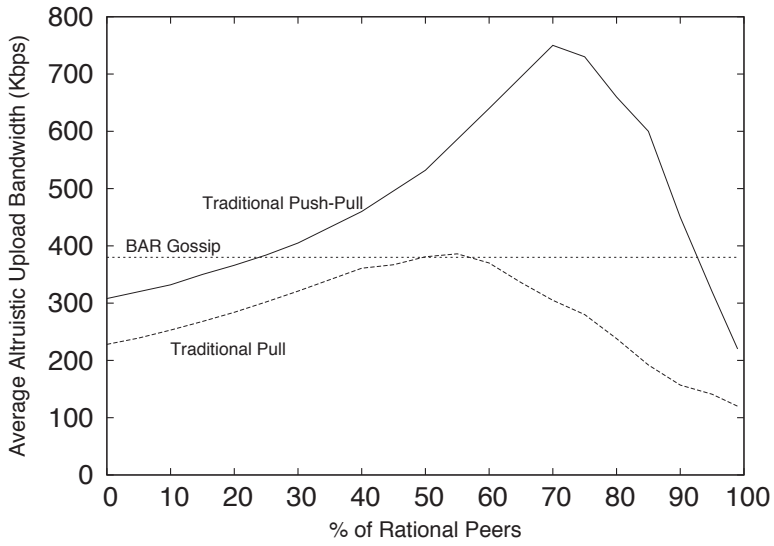

Figure 5.3: [sim] Send bandwidth used by an altruistic peer using traditional gossip versus BAR Gossip.

show that when rational peers comprise approximately 30% or more of the system, BAR Gossip is as reliable as push-pull and more reliable than pull-only. BAR Gossip consumes less bandwidth than push-pull at this point, as well. When rational peers make up less than 30% of the total peer population, a pull-only protocol is a better option than both BAR Gossip and push-only, as pull-only provides comparable reliability while wasting less bandwidth.

5.2.3 Unilateral Rational Deviation

We now examine deviant strategies that a rational peer might pursue. In our experiments, a rational peer pursues these strategies while the remaining peers obey the protocol. This approach is the experimental analog to the standard Nash equilibrium proof technique.

Strategy	Accepts Optimistic Push	Initiates Optimistic Push	Returns
Proactive/Data	Yes	Yes	Data
Proactive/Junk	Yes	Yes	Junk
Proactive/Decline	No	Yes	None
Passive/Data	Yes	No	Data
Passive/Junk	Yes	No	Junk
Passive/Decline	No	No	None

Table 5.2: Six strategies a rational peer may follow with regards to the Optimistic Push Protocol.

In this analysis, we make the following simplifying assumption. A rational peer's primary concern is to improve the delivered stream's quality by maximizing reliability and minimizing jitter; minimizing consumed bandwidth is a subordinate goal. We now consider the choices available to a rational peer with respect to Optimistic Push.

Table 5.2 lists the five strategies we consider that a rational peer may pursue to deviate from the Optimistic Push Protocol. *Proactive* strategies dictate that a rational peer initiates optimistic pushes as specified by the Optimistic Push Protocol. In contrast, *passive* strategies specify to never initiate optimistic pushes. *Data, junk*, and *decline* strategies prescribe that rational peers responding to an optimistic push send useful updates (when possible), send as much junk as allowed, or decline the exchange, respectively. Note that following the Optimistic Push Protocol corresponds to the Proactive/Data strategy.

Figure 5.4 plots the average probability the rational peer is missing each update as a function of time since that update was multicast by the broadcaster. We show how effective each strategy is in acquiring updates before they expire—lower lines corresponding to more effective strategies. Table 5.3 provides the corresponding jitter for each strategy.

Taken together, Figure 5.4 and Table 5.3 imply that rational peers should

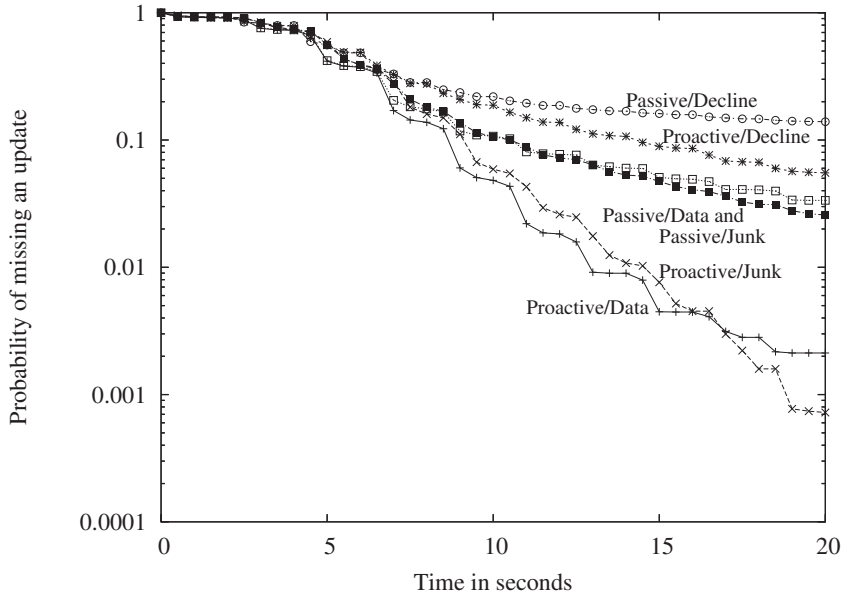

Figure 5.4: Average probability of the rational peer missing each update for different strategies.

Strategy	Average Jitter	Standard Deviation
Proactive/Data	0.48%	1.16%
Proactive/Junk	0.32%	0.78%
Proactive/Decline	11.59%	6.22%
Passive/Data	18.10%	6.08%
Passive/Junk	14.76%	9.44%
Passive/Decline	47.94%	7.52%

Table 5.3: Jitter that the rational peer experiences while pursuing different strategies.

follow either proactive/data or proactive/junk strategies. This is perhaps unsurprising, given that proactive strategies perform additional exchanges likely to result in more deliverable updates than passive strategies.

Figure 5.5 breaks the tie between the proactive/data and proactive/junk

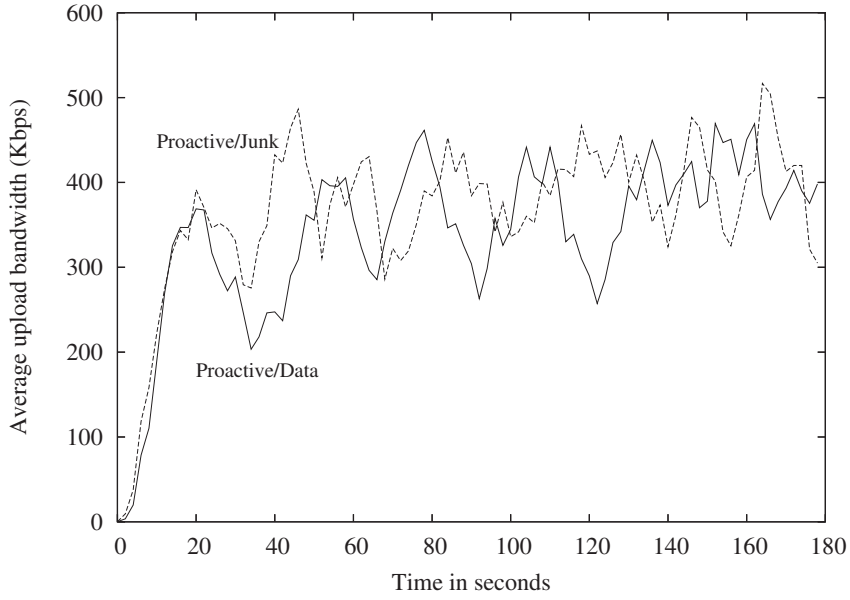

Figure 5.5: Rational peer's consumed bandwidth for different strategies.

strategies. The proactive/data strategy uses approximately 300 Kbps of upload bandwidth compared to proactive/junk's 317 Kbps. This difference is not an accident: we have *designed* BAR Gossip with `junk_cost`> 1 so that rational peers prefer filling their briefcases with valuable updates, rather than junk, whenever possible.

From these experiments, we conclude that a rational peer, when surrounded by other peers that follow BAR Gossip, has no obvious incentive for deviation—in fact, quite the contrary. While our experiments clearly fall short of proving that BAR Gossip as a whole (Balanced Exchange plus Optimistic Push) constitutes a Nash equilibrium, it does suggest that a Nash equilibrium is likely to be found at or near the strategy that corresponds to BAR Gossip. For instance, while we are unable to prove that there are no beneficial hybrid strategies that, depending on the environment, switch between two or more of the six strategies we have considered, it appears that the benefit of a proactive strategy derives from consistently participat-

ing in more exchanges, making it unlikely that switching occasionally to a passive strategy would provide a net gain. As for switching among proactive strategies, it yields no change in benefit while changing bandwidth costs, also providing little room for improvement.

5.2.4 Rational Collusion

Although a rational peer may find it difficult to benefit from cheating on its own, multiple rational peers can coordinate their actions to increase collective utility. We explore the effect of such collusion through simulations assessing the impact such a group may have on peers obeying the protocol.

We assume that colluding and non-colluding rational peers share a utility function. We also assume that colluding peers run a private protocol to disseminate updates among themselves. This protocol may be an alternative BAR protocol or it may be a non-BAR protocol bolstered by a high level of trust among colluding peers. We simulate a *perfect collusion* scenario in which every colluding peer immediately broadcasts new updates within the group at no cost. This source of updates reduces the incentive to fully participate in the BAR Gossip protocol. In particular, colluding peers only participate in balanced exchanges.

Figure 5.6 shows how the size of a perfect collusion group affects the quality of the stream seen by a peer following BAR Gossip. The intuition for the degraded performance is *i)* a non-colluding peer trades little when participating in a balanced exchange with a colluding peer and *ii)* colluding peers do not participate in optimistic pushes. In perfect collusion groups, colluding peers get most of their updates for free from other colluding peers, reducing their contributions to the rest of the system.

We find that when the collusion group size reaches 50% of the participants, other peers see an average convergence of 93% for an update, resulting in an unusable

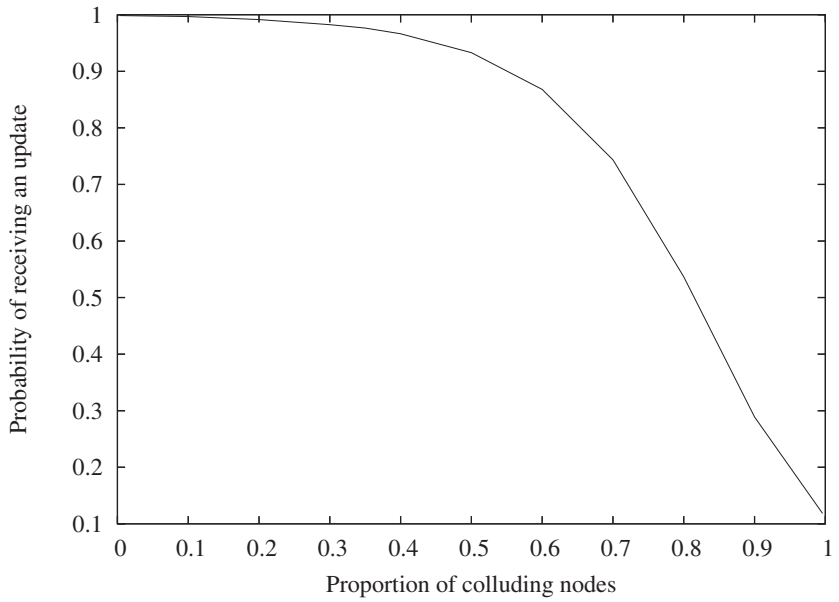

Figure 5.6: [sim] Effect of collusion on an altruistic peer's reliability when colluding peers are following the passive/decline strategy.

stream. Although near-perfect collusion among small groups seems plausible, it is unclear that collusion on a large scale is a significant threat. As the colluding group grows, so do the challenges of coordinating and trusting peers. Ironically, as a colluding group grows, it might require BAR Gossip to distribute updates internally as trust begins to break down among members.

5.2.5 Byzantine Impact

While we entice rational peers to behave correctly, we should also limit the impact of Byzantine actions on good users of the system. In these experiments, we focus our attention on Byzantine peers who exploit the messages and behaviors allowed by our protocol so as to harm the other peers in the system.

In BAR Gossip, Byzantine peers cannot subvert the system's safety prop-

erties. Because the broadcaster signs each update, Byzantine peers cannot undetectably tamper with the contents of any delivered update. Such peers can, however, impair progress by sending two kinds of messages: non-protocol messages and protocol messages. We regard generic DoS attacks based on non-protocol messages (e.g. bandwidth or connection flooding) as outside the scope of this work.

We design BAR Gossip to be robust against protocol-based attacks on liveness even if initiated by a significant number of Byzantine peers. First, BAR Gossip's partner selection protocol limits how many partners a peer can contact in a round—unlike traditional gossip, where a Byzantine node could potentially contact an unlimited number of nodes and involve them in useless exchanges. Second, Byzantine peers can inflict limited damage in the exchanges in which they participate. A Byzantine peer can remain silent during an exchange to slow the spread of updates, but fortunately, gossip protocols are naturally resilient to benign failures. One remaining concern is that a Byzantine peer could impact liveness by luring its partners into expensive message exchanges that ultimately fail in helping to disseminate updates.

We examine this kind of attack in the next set of experiemnts. During balanced exchanges and optimistic pushes, Byzantine peers report histories to maximize the number of updates that would be exchanged. For a balanced exchange, a Byzantine peer reports a history that is an exact complement of its partner's. For an optimistic push, a Byzantine peer announces a full young list and an empty old list if initiating, and requests the entire young list if receiving. Byzantine peers never enter the update or key exchange phases, so as not to generate a PoM, but still inducing its partner to devote significant bandwidth to the exchange without receiving any benefit. The presence of Byzantine peers can be viewed as an increase in the overhead associated with the environment as the costs associated with Byzantine peers depends upon the probability of entering an exchange with a Byzantine

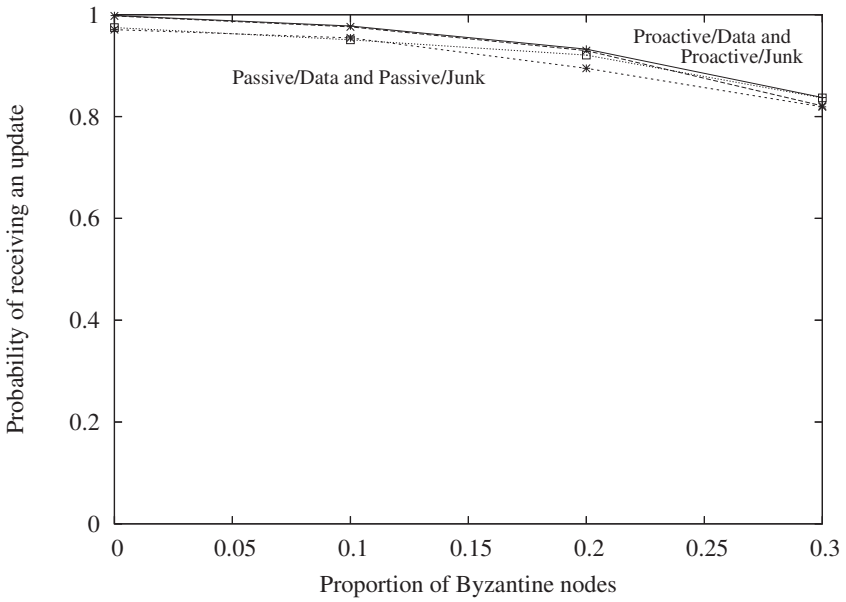

Figure 5.7: Rational peer's reliability for different strategies.

peer.

To mitigate the effects of this attack, non-Byzantine peers could refuse to participate in exchanges with peers who have historically been unreliable partners. We ignore such cat-and-mouse tactics and assess the impact of the above attack when non-Byzantine peers forget about previous unproductive exchanges.

Figures 5.7 and 5.8 show the reliability seen and bandwidth used, respectively, by a rational peer pursuing each strategy in the presence of different proportions of Byzantine peers. The remaining non-Byzantine peers are altruistic. The choice of strategies is similar to Section 5.2.3 where we considered unilateral deviation with no Byzantine peers. We elide proactive/decline and passive/decline strategies in which rational peers decline to participate in optimistic pushes, as these strategies performed significantly worse than the other strategies in the absence of Byzantine behavior. Passive and proactive strategies deliver unwatchable video

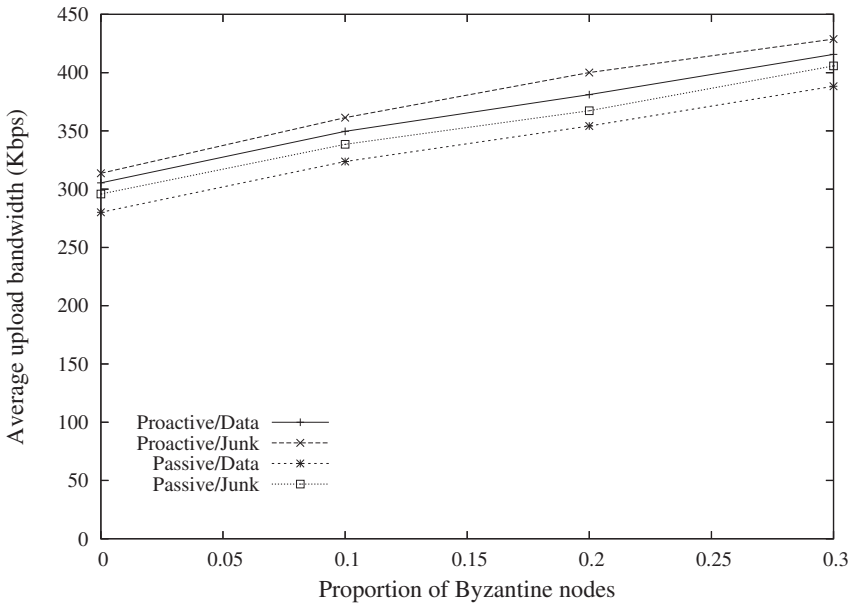

Figure 5.8: Rational peer's consumed bandwidth for different strategies.

streams when the proportion of Byzantine peers reaches 10% and 30%, respectively.

We conclude that among the strategies available, a rational peer should obey the protocol, i.e. pursue the proactive/data strategy, regardless of the presence of Byzantine peers. If all non-Byzantine peers are following the protocol, with a system comprised of 20% Byzantine peers, the bandwidth costs remain relatively constant while the convergence suffers by less than 7%.

5.3 Discussion

BAR Gossip is the first p2p live streaming system robust to Byzantine peers and rational peers. Its successes in a BAR setting, however, are reduced by its practical and theoretical limitations.

High overheads To stream a 200 Kbps video, peers upload approximately 300 Kbps on average to their neighbors. A significant fraction of this overhead stems from signatures on every message and receiving redundant stream updates. Furthermore, the Optimistic Push Protocol requires peers to waste network bandwidth in the form of junk data, a technique used in both BAR-B [3] and Equicast [38].

Static membership It is unclear how to design mechanisms to allow peers to join and leave the system without disrupting the properties of Nash equilibria. In a dynamic system, peers may deviate shortly before leaving if there are benefits to such actions.

Indefinite end time We assume that a streaming event ends at an indefinite time to eliminate end-game strategies in which peers may attempt to cheat at the end of a game because doing so is less risky than cheating at the beginning or middle. These end-game strategies pose a classic shortcoming of game theory—a limitation that is troubling given that many live events have well-known and published end times.

Short-sighted strategies Our proof that Balanced Exchange is a Nash equilibrium requires that rational peers only pursue strategies to maximize the number of useful updates traded in each exchange. We ignore more sophisticated strategies that may take into account the consequences of concurrent or future exchanges.

Partial Guarantees Our evaluation of Optimistic Push suggests that rational peers have little to gain from attempting to deviate unilaterally. However, those experiments fall short of *proving* that Optimistic Push is a Nash equilibrium. In essence, we sacrifice the purist approach of Balanced Exchange so as to achieve adequate performance in the form of BAR Gossip.

In addition to highlighting weaknesses of BAR Gossip, the above limitations reflect a broader frustration with formal approaches toward p2p system design. The resulting systems, BAR Gossip included, have been inefficient and have relied on strong assumptions. In the next chapter of this dissertation, we remove these limitations and show that rigor can also be practical.

Chapter 6

FlightPath

This chapter resolves the tension in p2p systems between designing practical incentives for cooperation and rigorously justifying that those incentives are enough. We propose *approximate equilibria* [15] as a new way to design p2p systems. Using these equilibria, we can design robust mechanisms to tolerate Byzantine peers. More importantly, approximate equilibria guide how we design systems to incentivize obedience while providing enough flexibility to implement practical solutions. We use this new approach to design FlightPath, a p2p live streaming system inspired by BAR Gossip, but fundamentally different because of how we take advantage of the flexibility afforded by not requiring a strict solution.

In FlightPath specifically, approximate equilibria let us use run-time adaptations to tame the randomness of our gossip-based protocol, making it suitable for low jitter media streaming while retaining the robustness and load balancing of traditional gossip. The key techniques enabled by this flexibility include allowing a bounded imbalance between peers, redirecting load away from busy peers, avoiding trades with unhelpful peers, and arithmetic coding of data to increase trading opportunities.

As a result of these dynamic adaptations, FlightPath is a highly efficient and

robust media streaming service that has several attractive properties:

High quality streaming: FlightPath provides good service to every peer, not just good average service. In experiments with over 500 peers, 98% of peers deliver every packet of an hour long video. No peer misses more than 6 seconds.

Broad deployability: FlightPath uses a novel block selection algorithm to cap the peak upload bandwidth so that the protocol is accessible to users behind cable or ADSL connections.

Rational-tolerant: FlightPath is a $\frac{1}{10}$-Nash equilibrium under a reasonable cost model, meaning that rational peers have provably little incentive to deviate from the protocol. We define an ϵ-Nash equilibrium in the next section.

Byzantine-tolerant: FlightPath provides good streaming quality despite 10% of peers acting maliciously to disrupt it.

Churn-resilient: FlightPath maintains good streaming quality while over 30% of the peer population may churn every minute. Further, it easily absorbs flash crowds and sudden peer departures.

Compared to BAR Gossip, the above properties represent *both* a qualitative and quantitative improvement. We reduce jitter by several orders of magnitude and decrease overhead by 50%. Additionally, we allow peers to join and leave the system without disrupting service.

Although approximate equilibria provide weaker guarantees than strict ones, they can be achieved without relying on the strong assumptions needed by the existing systems that implement strict Nash equilibria. BAR Gossip assumes that rational participants only pursue short-sighted strategies, ignoring more sophisticated ones that might pay off in the long term. Equicast [38] assumes that a user is hurt by an infinite amount if it loses any packet of a stream. FlightPath does away

with such assumptions, relying instead on the existence of a threshold below which few rational peers find it worthwhile to deviate.

We organize the rest of this chapter as follows. Section 6.1 defines ϵ-Nash equilibria and additonal assumptions we make. Section 6.2 describes FlightPath's basic trading protocol, discusses how to add flexibility to improve performance significantly, and explains how to handle churn. We evaluate our prototype in Section 6.3, which looks at FlightPath in a static setting, with churn, and under attack. In Section 6.4, we analyze the incentives a rational peer may have to cheat and show under what conditions FlightPath is a $\frac{1}{10}$-Nash equilibrium.

6.1 Assumptions

We analyze and evaluate FlightPath using ϵ-Nash equilibria.

Definition 3 *A protocol is an ex ante ϵ-Nash equilibrium if it defines a strategy profile such that each rational player expects to gain at most a factor of ϵ from deviating unilaterally. More formally, a strategy profile $\mathbf{s} = (s_0, \ldots, s_n)$ is an ex ante ϵ-Nash equilibrium if for every player i there exists no strategy s_i^* such that i expects utility $u_i(\mathbf{s}_{-i}, s_i^*) > (1+\epsilon)u_i(\mathbf{s})$ from following s_i^**[1].

Again, we elide "ex ante" for brevity. Within the framework of ϵ-Nash equilibria, we assume that rational peers deviate if and only if they expect to benefit by more than a factor of ϵ from deviating unilaterally. This assumption is reasonable as switching protocols incurs a non-trivial cost such as effort to develop a new algorithm, work to install new software, or risk that new software will be buggy or malicious. Under such circumstances, it may not be worth the trouble to develop

[1] As Chien et al. [15] note, an alternative notion of ϵ-Nash equilibria is based on ϵ being an additive component instead of a factor. They observe that both are equally natural, although treating ϵ as a factor is more in line with traditional approximation guarantees in computer science. Additionally, several works that treat ϵ as an additive component also normalize all utilities into the range [0,1], giving ϵ more of a relative role rather than a strict additive one.

or use an alternate protocol. In FlightPath, we assume that protocols that bound the gain from cheating to $\epsilon \leq \frac{1}{10}$ are sufficient to discourage rational deviations.

FlightPath is the first p2p system that is based on an approximate equilibrium. To our knowledge, FlightPath is the first work to explore how these equilibria can be used to trade off resilience to rational manipulation against practical concerns such as performance and overhead. Other works [15, 20], which have been mainly theoretical, have used approximate equilibria only when the strict versions have been difficult to find.

A peer's utility As in BAR Gossip, rational peers are interested in acquiring updates to reconstruct a data stream. A rational peer benefits from delivering a jitter-free stream and benefits less as jitter gets worse. Rational peers incur cost by uploading bytes. In contrast to BAR Gossip, we assume downloading costs are negligble. We again assume that eviction is a sufficient penalty to deter any peer from sending a message that the receiver could present as a PoM.

Although FlightPath is not tied to any specific utility function that combines these benefits and costs, we provide one here for concreteness. Note that crafting a utility function that accurately captures a peer's utility is a dark art at best. We therefore design a conservative function defining a peer $i's$ utility when strategy profile $\mathbf{s}$ is played as

$$u_i(\mathbf{s}) = (1 - j_i)\beta - w_i\kappa$$

where j_i is the average number of jitter events per minute that i experiences, w_i is the average upload bandwidth used by i in kilobits per second, β is the benefit received from a jitter free data stream, and κ is the cost for each kilobit per second of upload bandwidth consumed. We consider this function conservative because a peer's benefit decreases rapidly according to the number of jitter events experienced: a peer that is jittered once every minute obtains no benefit, while a peer that is

jittered once every ten minutes receives 90% of the benefit. At the end of this chapter, we show how the benefit to cost ratio ($\frac{(1-j_i)\beta}{w_i\kappa}$) affects the ϵ we can bound in an ϵ-Nash equilibrium.

6.2 Design

We discuss FlightPath's design in three iterations. In the first, we give an overview of a basic structure, similar to Balanced Exchange, that allows peers to trade updates with one another. We design trades to force rational peers to act faithfully in each trade until the last possible action, where deviating can save only negligible cost. This basic protocol allows few opportunities for a peer to game the system, but by the same token, it provides few options for dynamically adapting to phenomena like bad links, malicious peers, or overload. Therefore, in the second iteration, we describe how we add controlled amounts of choice to the basic trading protocol to improve its performance dramatically. In the third iteration, we show how to modify the protocol to deal with changing membership.

In contrast to our previous discussion of Balanced Exchange, readers may be surprised to see that in the last two iterations we do not argue step-by-step about possible ways to cheat and why a rational peer would not. This difference is due to the flexibility of approximate equilibria, which allows optimizations that improve a user's start-to-finish benefits and costs, while still limiting any possible gains from cheating. At the end of this chapter, we show that a rational peer expects little to gain from attempting to cheat.

6.2.1 Basic Protocol

Prior to a live event, peers contact the *tracker* to join a streaming session. After authenticating each peer, the tracker assigns unique random member ids to peers and posts a static membership list for the session.

In each round, the *source* sends two kinds of updates: stream updates and linear digests. A stream update contains the actual contents of the stream. The source tags each stream update with the round in which it is multicast and the order in which it should be delivered in a round. Each stream update can be uniquely identified by its round number and order in which it should be delivered, a pair of numbers we refer to as an update id. A linear digest [32] contains information that allows peers to authenticate received stream updates. The source associates one or more linear digests to each round and includes update ids and corresponding secure hashes of stream updates in those digests. The source signs linear digests so that their contents can be immediately authenticated. We use linear digests in place of digitally signing every stream update to reduce the computational load and bandwidth necessary to run FlightPath. The source sends each of the `ups_per_round` unique stream updates for a round to a small fraction `seed_frac` of random peers in the system. When the source multicasts stream updates to selected peers at the beginning of every round, it also sends them the appropriate linear digests.

In each round, peers initiate and accept trades from their neighbors. We provide Figure 6.1 for reference and base our design on Balanced Exchange. A trade consists of four phases:

Partner Selection: A peers selects a partner using a verifiable pseudo-random algorithm.

History Exchange: Partners exchange histories describing which updates they possess and which they still need. Partners use the histories to compute deterministically the exact updates they expect to receive and are obligated to send, under the constraint that partners exchange equal numbers of updates.

Update Exchange: Partners swap updates by encrypting them and sending the encrypted data in a briefcase message. Immediately afterwards, a peer sends

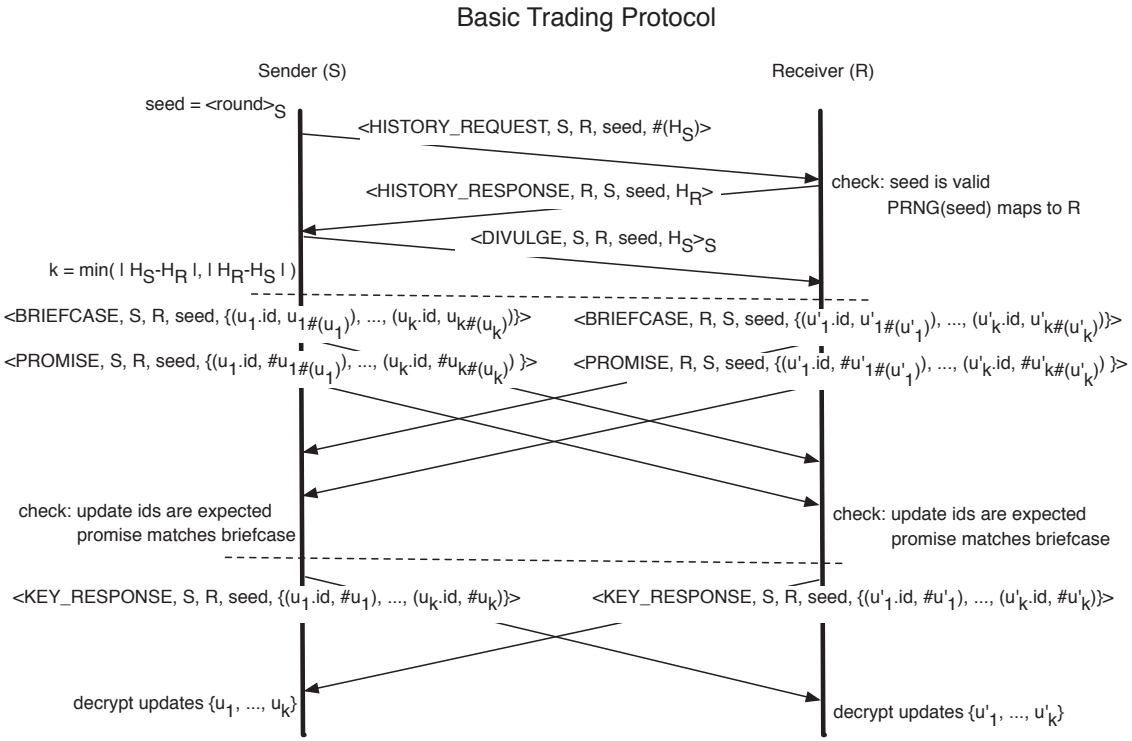

Figure 6.1: Illustration of a trade in the basic protocol.

a *promise* pledging that the contents of its briefcase are legitimate and not garbage data.

Key Exchange: Once a peer receives a briefcase and matching promise message from its trading partner, that peers sends the decryption keys necessary to unlock the briefcase it sent.

In contrast to BAR Gossip and BAR-B [3], FlightPath peers do *not* sign every protocol message. Promises are the only digitally signed message in a trade; peers authenticate other messages using message authentication codes [73]. As in balanced exchange, these phases guarantee that a rational peer has to upload the bulk of data in a trade to obtain any benefit from the trade. By deferring gratification and holding

peers accountable via promise messages, we limit how much a cheating strategy can gain over obeying the protocol. The main difference between trades in this protocol compared to balanced exchanges in BAR Gossip is the addition of the promise.

We structure promises so that for each briefcase there is exactly one matching promise. Further, if a briefcase contains garbage data, then the matching promise is a PoM. Briefcases and promises provide this property because of how we intertwine these two kinds of messages. For each update u that a peer is obligated to send, that peer includes the pair $\langle u.id, u_{\langle \#u \rangle} \rangle$ in the briefcase it sends, where $u_{\langle \#u \rangle}$ denotes update u encrypted with a hash of itself. For each entry in the briefcase, the matching promise message contains a pair $\langle u.id, \#(u_{\langle \#u \rangle}) \rangle$. Therefore, if a briefcase holds garbage data, then the matching promise message would serve as a PoM since that promise would contain at least one pair in which the hash for a self-encrypted update is wrong. Of course, a peer could upload garbage data in its briefcase but send a legitimate promise message to avoid sending a POM, but then the briefcase and promise would not match and that peer's partner would refuse to send the decryption keys.

6.2.2 Taming Gossip

Gossip protocols are well-known for their robustness [10, 21, 23, 29, 30, 74, 76] and are especially attractive in a BAR environment because their robustness helps tolerate Byzantine peers. However, while gossip's pair-wise interactions make crafting incentives easier than in a tree-based streaming system, it is reasonable to question whether that very randomness may make gossip inappropriate for streaming live data in which updates need to be propagated to all nodes by a hard deadline.

In this section, we explain how the flexibility of approximate equilibria allows us to tame gossip's randomness by dynamically adapting run-time decisions. For concreteness, we show in Figures 6.2 and 6.3 how poorly the basic protocol performs

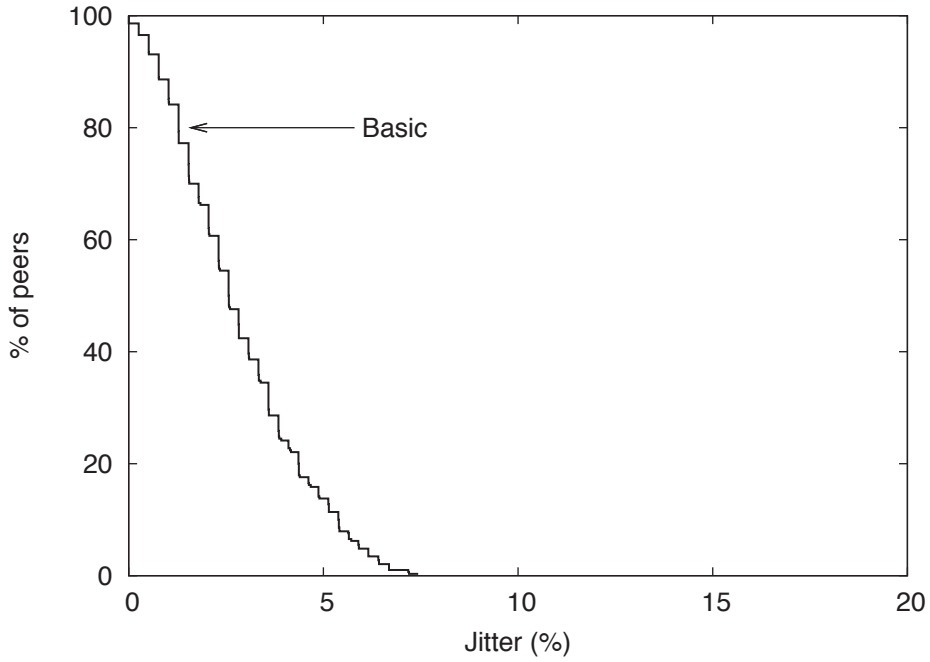

Figure 6.2: Reverse cumulative distribution of jitter.

when disseminating a 200 Kbps stream to 517 clients. In this experiment, the source generates `ups_per_round` = 50 unique stream updates per round and sends each one to a random `seed_frac` = 5% of the peers. Updates expire `deadline` = 10 rounds from the round in which they are sent. Figure 6.2 shows a reverse cumulative distribution of jitter experienced by peers. To read this graph, it may be useful to focus where the arrow is pointing. At that point, approximately 80% of peers are jittered at least 2% of the time. Figure 6.3 depicts cumulative distributions of peers' average and peak upload bandwidths. We observe that a peer uses 300 kbps on average, a modest overhead that many home broadband connections can handle. Peak bandwidths, however, far exceed the constraints that most cable or DSL connections can bear.

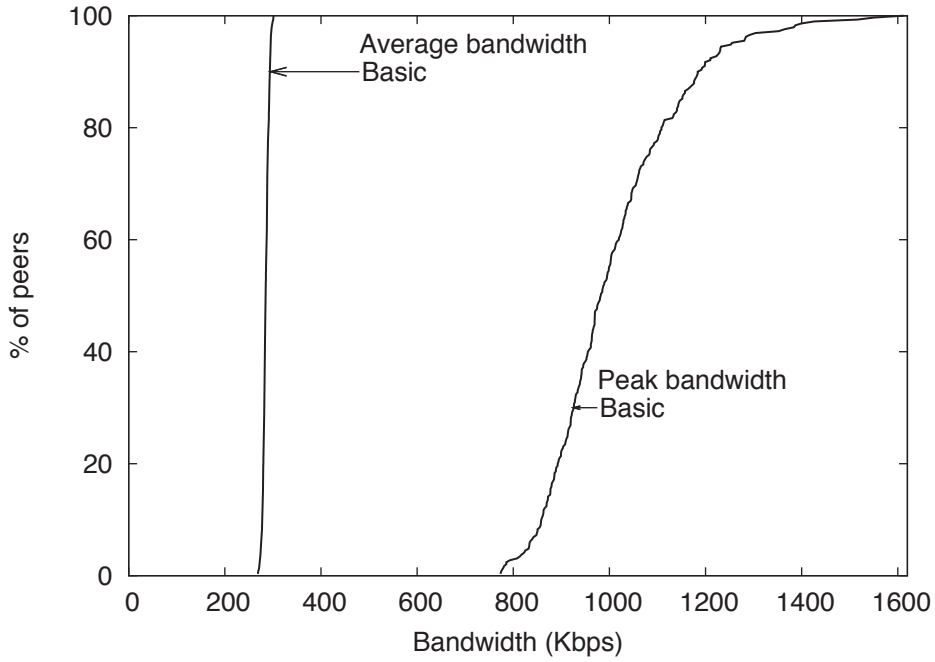

Figure 6.3: Cumulative distribution of average and peak bandwidths.

In the rest of this subsection, we explore how to modify the basic trading protocol both to reduce jitter and to lower consumed bandwidth. The first set of modifications aim at capping the peak bandwidth used by the protocol. As expected, by reining in gossip's largesse with bandwidth, these improvements make jitter worse. With these techniques in place though, we are free to explore ways to reduce jitter while constraining bandwidth to reasonable limits.

Reservations: One of the problems of using gossip to stream live data is the widely variable number of trading partners a peer may have in any given round. In particular, although the expected number of trades in which a peer participates in each round is two, the actual number varies widely, occasionally going past eight.

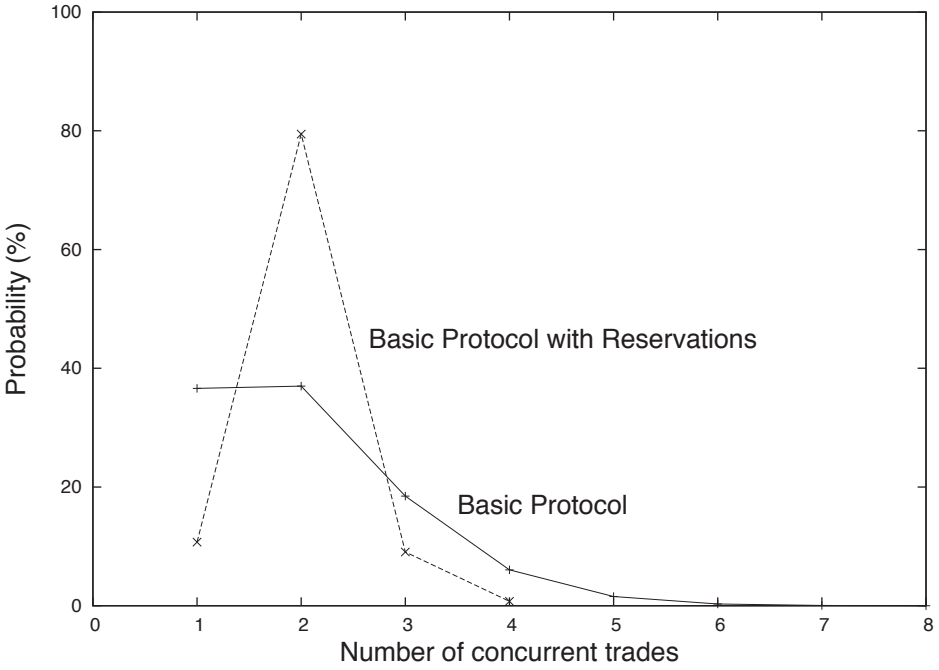

Figure 6.4: Probability distribution showing likelihood of a peer participating in x concurrent trades.

Such high numbers of concurrent trades are undesirable for two reasons. First, a peer can be overwhelmed and be unable to finish all of its concurrent trades within a round. Figure 6.3 illustrates this problem as a high peak bandwidth in the basic protocol, making it impractical in bandwidth-constrained environments. Second, a peer is likely to waste bandwidth by trading for duplicate updates when participating in many concurrent trades.

Rather than accept all incoming connections, FlightPath distributes the number of concurrent trades more evenly by providing a limited amount of flexibility in partner selection. The idea is simple. A peer S reserves a trade with a partner R before the round rnd in which that trade should happen. If R has already

accepted a reservation for S, then S looks for a different partner. Figure 6.4 illustrates that this straight-forward approach can significantly reduce the probability of a peer committing to more than two concurrent trades in a round. At the same time, reservations also reduce the probability that a peer is only involved in the one trade it initiates each round. The challenge in implementing reservations is how to give peers *verifiable* flexibility in their trading partners.

FlightPath provides each peer a small set of potential partners in each round. We craft this set carefully to address several requirements:

1. Peers need to select partners in a sufficiently random way to retain gossip's robustness.

2. Each peer needs enough choices to avoid overloaded or Byzantine peers.

3. A peer's partners should be relatively unchanged if the population does not change much.[2]

4. The selection algorithm ought to be resilient to attacks in which malicious peers attempt to position themselves so as to eclipse good peers [69].

Figure 6.5 illustrates how we provide flexibility in choosing a partner while meeting the above constraints. We force each peer to communicate with at least $\lceil \log n \rceil$ distinct neighbors by partitioning the membership list of n peers into $\lceil \log n \rceil$ bins and requiring a peer to choose a partner from a verifiable pseudorandomly chosen bin each round. Leitao et al. demonstrate that a set of gossip partners that grows logarithmically with system size can tolerate severe disruptions [40]. In round rnd, peer S seeds a pseudo-random generator with $\langle rnd \rangle_S$, and uses the generator to select a bin; note that any peer can verify any other peer's bin selection.

[2] Although we discuss dynamic membership in the next section, its demands constrain the partner selection algorithm we describe here.

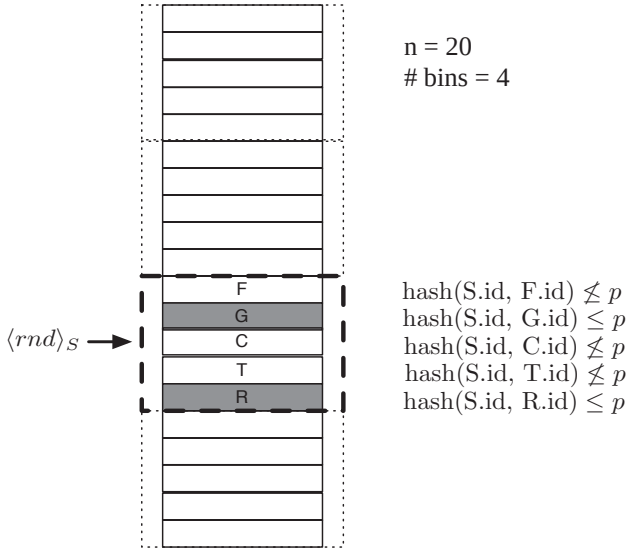

Figure 6.5: Illustration of partner selection algorithm using bins.

Within a bin, we further restrict the nodes with whom a peer can communicate by giving each peer a customized view of each bin's membership based upon a peer's id. We define S's view to be all peers R such that the hash of S's member id with R's member id is less than a parameter p. The tracker adjusts p so that almost every peer is expected to have at least one non-Byzantine partner in every bin. To achieve this condition, the tracker first shuffles the membership list so that every entry has equal likelihood to correspond to a Byzantine peer. The tracker then sets p to satisfy the following inequality:

$$[1 - [1 - p(1 - F_{byz})]^{\frac{n}{\lceil \log n \rceil}}]^{\lceil \log n \rceil} \geq 1 - \frac{1}{n} \tag{6.1}$$

Though perhaps initially initimidating, the above inequality can be intuitively understood. The expression on the left represents the probability that a peer has at least one non-Byzantine partner in every bin. We wish that probability to

be sufficiently high so that nearly all peers save one—represented by the expression on the right—has at least one non-Byzantine partner in every bin.

Consider two random peers S and R. The probability that R is not Byzantine and in S's view is $p(1 - F_{byz})$. Therefore, the probability that for a given bin, S has no partners or all the partners it does have are Byzantine is $[1 - p(1 - F_{byz})]^{\frac{n}{\lceil \log n \rceil}}$, capturing the probability that a given bin is *bad* for S. The probability that a given bin is *okay* for S is therefore $1 - [1 - p(1 - F_{byz})]^{\frac{n}{\lceil \log n \rceil}}$. Putting it together, the probability that all of S's bins are okay is given as the left expression in Inequality 6.1. Figure 6.6 gives an intuition for how this inequality affects a peer's choices as the system scales up and as the bound of Byzantine peers changes. This two level selection gives us a way to provide $\log n$ distinct random partners with high probability while limiting the variance a single level selection scheme may possess.

A peer S can use the choice provided by the combination of bins and views to reserve trades. A peer R that receives such a reservation verifies that S's view contains R and that $\langle rnd \rangle_S$ maps to the bin that contains R's entry in the membership list. If these checks pass, then R can either accept or reject the reservation.

As a general rule, peer R accepts a reservation only if it has not already accepted another reservation for the same round. Otherwise, S rejects the reservation, and S attempts a reservation with a different peer. Peer S can be exempt from this rule by setting a *plead* flag in its reservation, indicating that S has few options left. In this case, R accepts the reservation unless it has already committed to c trades in round r. We find that setting c to 4 is good in practice, as c should be small but greater than 2 and setting it to 3 did not perform as well as 4.

Splitting need: Reservations are effective in ensuring that peers are never involved in more than 4 concurrent trades. However, a peer that is involved in concurrent trades may still be overwhelmed with more data than it can handle during a round and may still receive too much duplicate data.

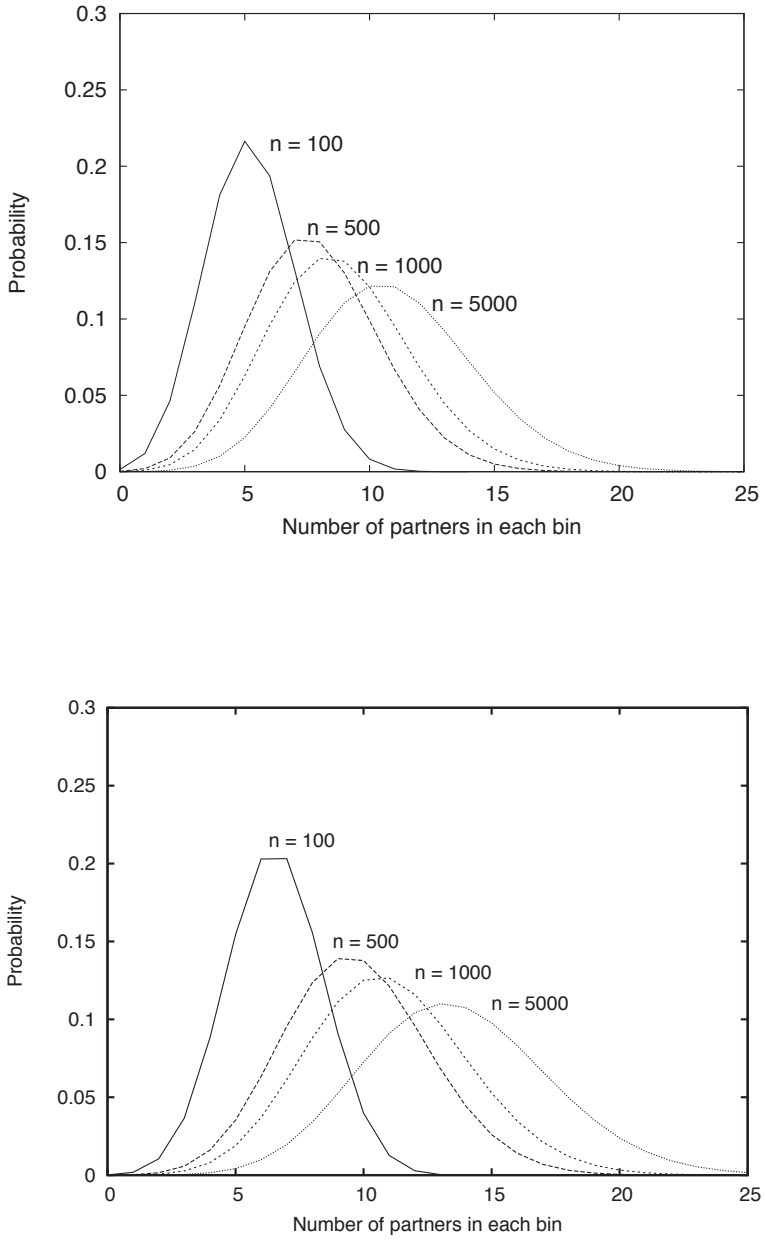

Figure 6.6: Distribution of view sizes in each bin for different membership list sizes. Graphs are calculated with $F_{byz} = 5\%$ (top) and 20% (bottom).

For example, consider a peer S involved in concurrent trades with peers R_0, R_1, R_2, and R_3. Peer S is missing eight updates for a given round. The basic protocol may overwhelm S and waste bandwidth by having peers R_0–R_3 each send those 8 updates to S. Something more intelligent is for c's need to be split evenly across its trading partners, limiting each partner to send at most two updates. Note that while this scheme may be less wasteful than before, c now risks not receiving the eight updates it needs since it is unlikely that its partners each independently select disjoint sets of two updates to exchange.

There seems to be a fine line between being conservative to avoid jitter but receiving redundant data or taking a risk to save resources. We sidestep this trade-off by using erasure coding [4, 50].

Erasure codes: Erasure coding has been used in prior works to improve content distribution [2, 16, 26, 39], but never to support live streaming in a setting with Byzantine participants. The source codes all of the stream data in a given round into $m >$ `ups_per_round` stream updates such that any `ups_per_round` blocks are necessary and sufficient to reconstruct the original data. A peer stops requesting blocks for a given round once it has a sufficient number. Erasure coding has three important benefits. First, it increases the diversity of updates among peers, making the barter system we have constructed more effective. Second, it improves the fault-tolerance of the overall system as it is more resilient to the loss of a few updates. Third, erasure coding reduces the probability that concurrent trades involve the same block.

To provide an example of this third benefit, consider a peer S involved in concurrent trades with R_0 and R_1. Suppose there are 50 updates per round erasure coded into 100 blocks. S possesses 44 of those blocks, R_0 has 40, and R_1 holds 49. Although S only needs 6 blocks to reconstruct the original updates, R_0 and R_1 probably have more than 6 blocks not in common with S. Combinig the splitting

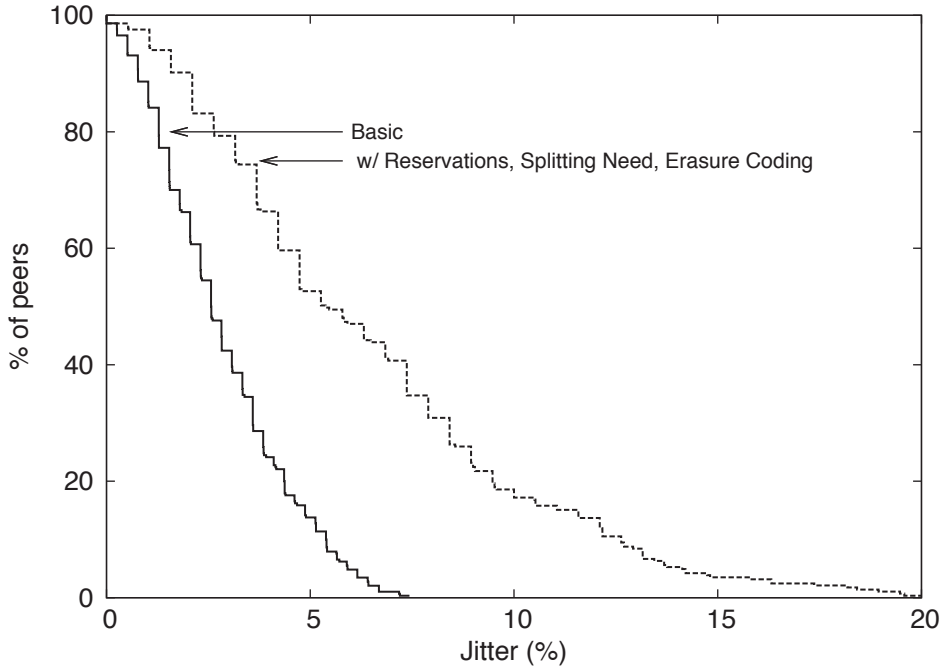

Figure 6.7: Reverse cumulative distribution of jitter in the basic trading protocol and with the reservations, splitting need, and erasure coding techniques incorporated.

need and erasure coding techniques, R_0 randomly selects 3 blocks which S does not possess and trades them to S. R_1 takes a similar action. So that peers can determine exactly which blocks will be traded, peers select blocks pseudo-randomly based upon the seed value used to initiate the trade.

In our experiments, we erasure code `ups_per_round` stream updates into $m = c \times$ `ups_per_round` blocks and modify the source to send each one to $\frac{\texttt{seed\_frac}}{c}$ of the peers. Although coding stream data into more and more blocks can reduce jitter, we choose not to increase the coding beyond $m = 2$`ups_per_round` because a greater diversity of blocks also has a side-effect. More coded blocks means that trades are more effective and peers can gather the updates they need in a shorter

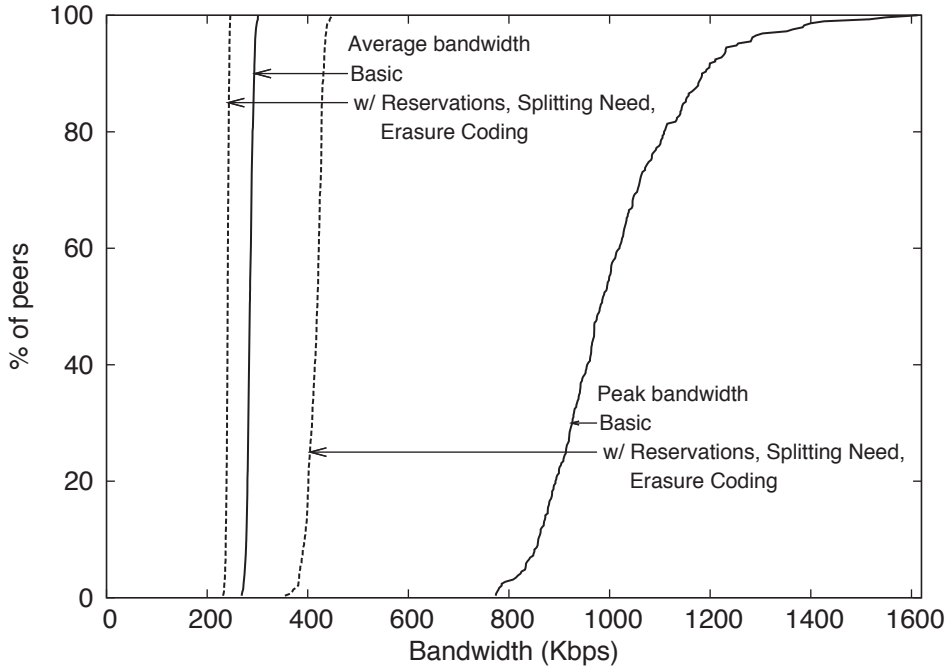

Figure 6.8: Cumulative distribution of average and peak upload bandwidths in the basic trading protocol and with the reservations splitting need, and erasure coding techniques incorporated.

amount of time. However, such efficiency hurts peers who may have fallen behind as they then have fewer opportunities to catch up with their neighbors before those neighbors acquire all the updates they need.

In Figures 6.7 and 6.8, the source generates 2`ups_per_round` $= 100$ blocks and sends each one to a random 2.5% of the peers. The reservations, splitting need, and erasure coding techniques reduce the protocol's peak bandwidth significantly, but at the cost of making jitter worse. We now describe three techniques that together nearly eliminate jitter without compromising the steps we have taken to keep the protocol from overwhelming any peer.

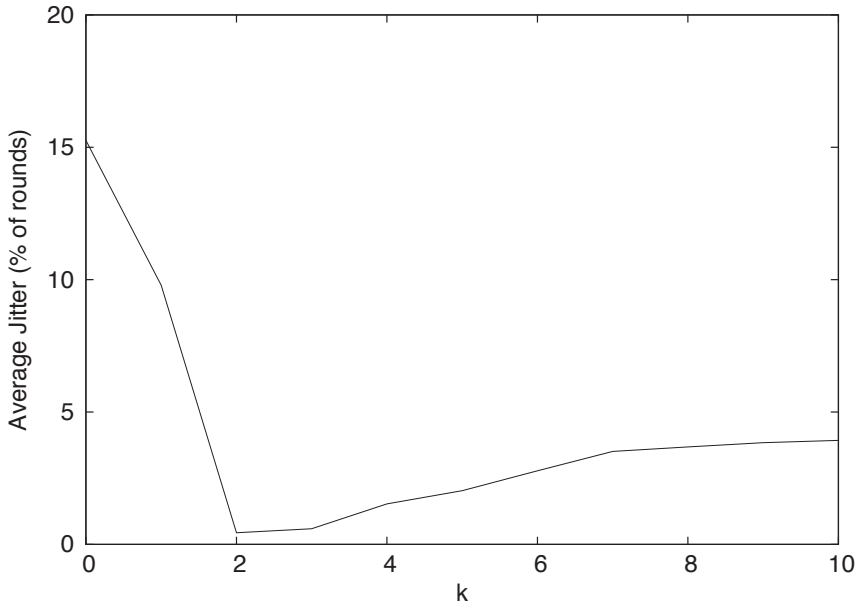

Figure 6.9: Average jitter as a function of the number k of older rounds from which a peer prefers to receive updates.

Tail inversion: As in many gossip protocols, the basic trading protocol biases recent updates over older ones to disseminate new data quickly. However, in a streaming setting, peers may sometimes value older updates over younger ones, for example when a set of older updates is about to expire and a peer seeks to avoid jitter. Indeed, we find that after using the techniques described so far, a peer is typically missing fewer than five updates for the round in which it experiences jitter.

The drawback in preferring to trade for updates of an old round is that the received updates may not be useful in future exchanges because many peers may already possess enough updates to reconstruct the data streamed in that round. Indeed, an oldest-first bias does not perform well in our experiments. Therefore,

FlightPath provides a peer with the flexibility to balance recent updates that it can leverage in future exchanges against older updates that it may be missing. Instead of requesting updates in most-recent-first order, a peer has the option to receive updates from the k oldest rounds first and then updates in most-recent-first order. Figure 6.9 depicts the jitter experienced by peers for values of k ranging from zero to the deadline (10). When k is zero, peers value updates in a most-recent-first order. When k is equal to the deadline, peers prefer receiving older update. Our experiments indicate that inverting the tail with $k = 2$ is very effective at reducing jitter. However, we acknowledge that this technique is not the result of deep insight—it simply works well and is the product of a low-level understanding of the FlightPath system. Better ways to prioritize updates may well exist.

Imbalance ratio: The basic protocol balances trades so that a peer receives no more than it contributes in any round. Such equity can make it difficult for a peer that has fallen behind to recover.

FlightPath uses an *imbalance ratio* `imb_ratio` to introduce flexibility into how much can be traded. Each peer tracks the number of updates sent to and received from its neighbors, ensuring that its credits and debits for each partner are within `imb_ratio` of each other. We find that the imbalance ratio's most dramatic effect is that it allows individual trades to be very imbalanced if peers have long-standing relationships.

When `imb_ratio` is set to 1, the trading protocol behaves like a traditional unbalanced gossip protocol, vulnerable to free-riding behavior [43]. When `imb_ratio` is set to 0, every trade is balanced, offering little for rational peers to exploit, but also allowing unlucky peers to suffer significant jitter. We would like to set `imb_ratio` to be as low as possible while maintaining low jitter. In Figure 6.10, we show the impact different imbalance ratios have on peers' jitter and find setting `imb_ratio` to 10% is an acceptable tradeoff between the competing concerns of reducing jitter

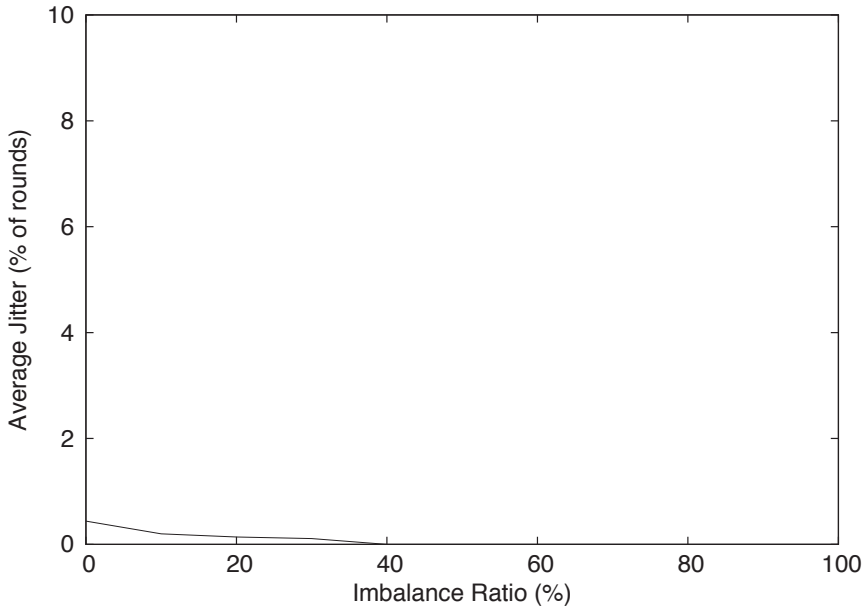

Figure 6.10: Average jitter as a function of the imbalance ratio. For readability, the scale of the y-axis is from 0 to 10%.

while limiting the incentives for rational peers to cheat. Figure 6.11 shows how the tail inversion and imbalance ratio techniques reduce jitter, while illustrating that both techniques have a small impact on the bandwidth consumed.

Trouble Detector: Our final improvement takes advantage of the partner flexibility afforded by the reservation mechanism. Each peer monitors its own performance by tracking how many updates it still needs for each round. If its performance falls below a threshold, then that peer proactively initiates more than one trade in a round to avoid jitter. Peers treat this option as a safety net, as increasing the average number of concurrent trades also increases the average cost to trade for each unique update.

We implement a simple detection module that informs a peer whether re-

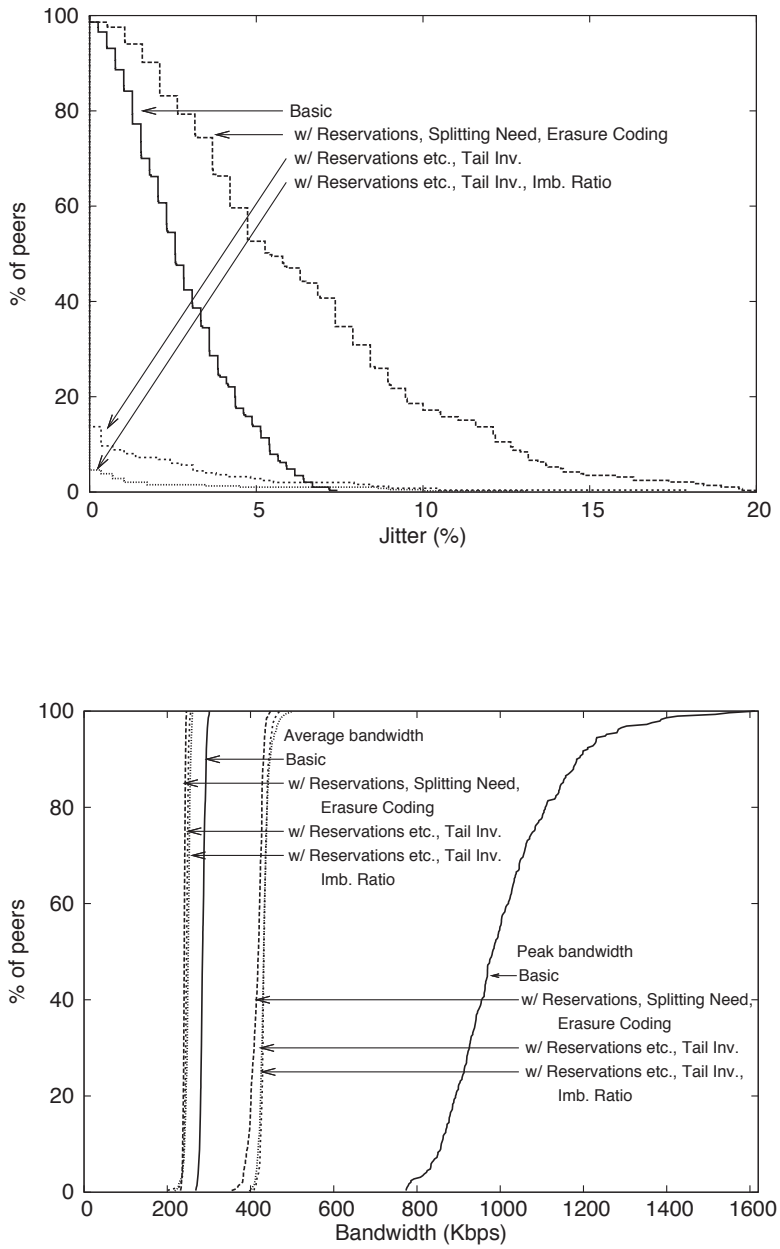

Figure 6.11: Reverse cumulative distribution showing impact of tail inversion and imbalance ratio techniques on reducing jitter. CDFs of average and peak bandwidths demonstrating tail inversion and imbalance ratio techniques impose modest overhead.

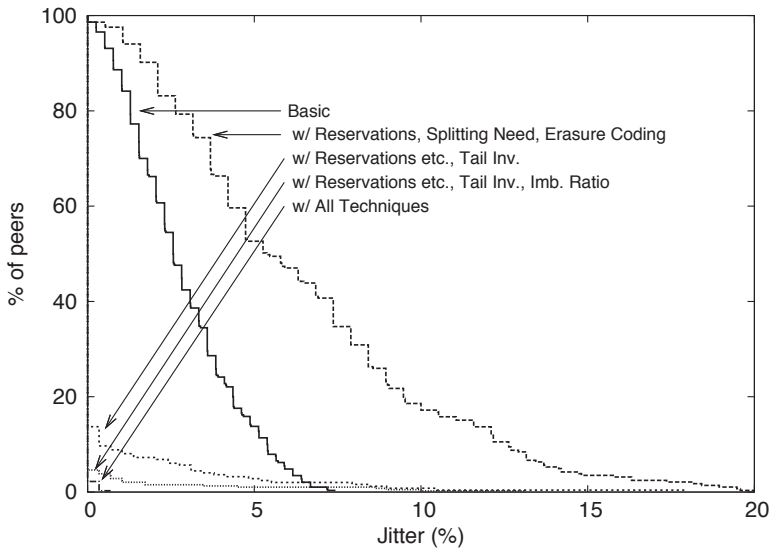

Figure 6.12: Reverse CDFs showing how peers can use the trouble detector to reduce jitter by proactively initiating more trades.

serving more trades may be advisable. We assume that after each round a peer expects to double the number of updates that have not yet expired up to the point of possessing `ups_per_round` updates for each round. In practice, we find that peers typically gather updates more quickly than just doubling them. If a peer c notices that it possesses fewer updates than the detection module advises, c schedules additional trades. Note that this is a local choice, based only on how many packets the peer has received for that round. Figure 6.13 demonstrates the effectiveness of adding the trouble detector module.

6.2.3 Dynamic Membership

We now explain how to augment the protocol to handle peers joining and leaving the system. In FlightPath, the main challenge is in allowing peers to join an existing streaming session. Gossip's robustness to benign failures lends FlightPath a natural

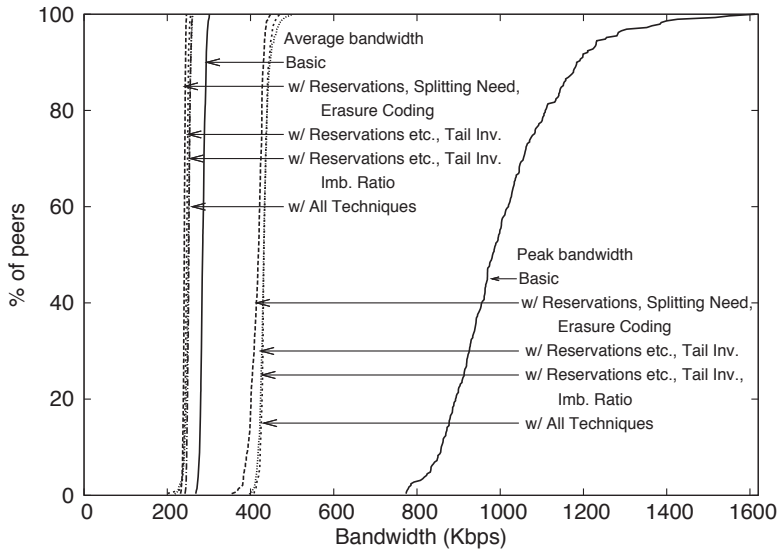

Figure 6.13: CDFs of average and peak bandwidths demonstrating modest overheads due to the trouble detection technique.

resilience to departures. However, the tracker still monitors peers to discover if any have left the system abruptly. Currently, we employ a simple pinging protocol, although we could use more sophisticated mechanisms as in Fireflies [35].

When a peer attempts to join a session, it expects to begin reliably receiving a stream without a long delay. As system designers, we have to balance that expectation against the resources available to get that peer up to speed. In particular, dealing with a flash crowd where the ratio of new peers to old ones is high presents a challenge. Moreover, in a BAR environment, we have to be careful in providing benefit to any peer who has not earned it. For example, if a single peer joins a system consisting of 50 peers, it may be desirable for all 50 to aid the new participant using balanced trades so that the new peer cannot free-ride off the system. However, consider the case when instead of a single peer joining, 200 or 400 join. It is unreasonable to expect the original 50 to support a population of 400 peers who

initially have nothing of value to contribute.

Below, we describe two mechanisms for allowing peers to join the system. The first allows the tracker to modify the membership list and to disseminate that list to all relevant peers. The second lets a new peer *immediately* begin trading so that it does not have to wait in silence until all peers have been notified of its presence.

Epochs: A FlightPath tracker periodically updates the membership list to reflect joins and leaves. The tracker defines a new membership list at the beginning of each epoch, where the first epoch contains the first `epoch_len` rounds, the second epoch contains the next `epoch_len` rounds and so on. If a peer joins in epoch e, the tracker places that peer into the membership list that will be used in epoch $e + 2$.

At the boundary between epochs e and $e + 1$, the tracker shuffles the membership list for epoch $e + 2$ and notifies the source of the shuffled list. Shuffling prevents Byzantine peers from attempting to position themselves at specific indices of the membership list, so as to take over a bin. Recall that we construct each peer's membership view to be independent of these indices so as not to end long-standing relationships prematurely.

After the tracker notifies the source of the next epoch's membership list, the source divides that list into pieces and places each piece into a third kind of update: *a partial membership list*. The source signs these lists and distributes them to peers as it would a stream update. Peers can trade partial membership lists just like they trade linear digests and stream updates. The only difference is that partial membership lists are given priority over all other updates in a trade and only expire when the epoch corresponding to that list ends. Once a peer obtains every partial membership list for an epoch, that peer can reconstruct the original membership list and use it to select trading partners.

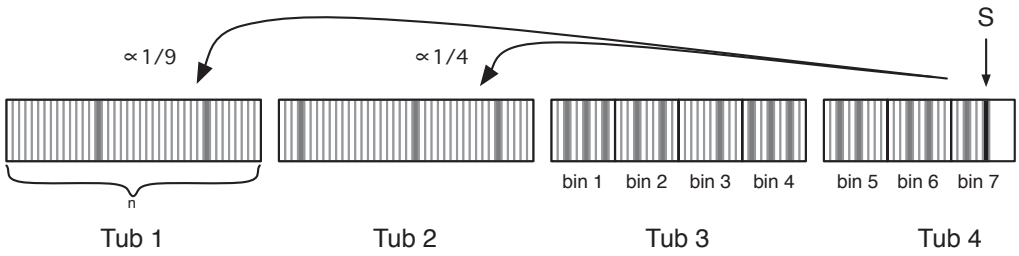

Figure 6.14: Illustration of the tub protocol from peer S's perspective. Shaded entries represent peers that S can contact for a trade when appropriate. Note that S only uses bins for its own tub and the immediately preceding one.

Tub Algorithm: As described, a new peer would have to wait at least one epoch before it appears in the membership list and can begin to trade. FlightPath augments the static partner selection algorithm that uses bins with an online algorithm that allows new peers to begin trading immediately without overwhelming the existing peers in the system. This algorithm also allows every peer to verify partner selections without global knowledge of how many peers joined nor of when they did so. Intuitively, our algorithm organizes all peers into *tubs* such that the first tub contains the peers in the current epoch's membership list and subsequent tubs contain peers who have recently joined. A peer selects partners from its own tub and also from any tub preceding its own. However, the probability that a peer from tub t selects from a tub $t' < t$ decreases geometrically with $t - t'$. This arrangement ensures that the load on a peer from all subsequent tubs is bound by a constant regardless of how many peers join. Figure 6.14 illustrates our algorithm.

For clarity, we describe our online algorithm assuming all peers have a global list that enumerates every peer in the system. Later, we show that this knowledge is unnecessary. The first n indices in this global list correspond to the n indices of the current epoch's membership list. The rest of the global list is sorted according to the order in which peers joined. We divide the global list into *tubs* where the first tub corresponds to the first n indices of the global list, the second tub to the next

n indices, and so forth.

A peer S's membership view depends on its position in the global list. If S is in the first tub, its view and how it selects partners is unchanged from the static case (Section 6.2.2). If S is in a tub $t > 1$, S's view obeys three constraints:

1. Only peers that precede S in the list can be in $S's$ view..
2. If R is in tub t or $t-1$, then R is in $S's$ view iff the hash of concatenating S's member id with R's member id is less than p (see inequality 6.1).
3. If R is in a tub $t' < t-1$, then R is in S's view iff the hash of S's member id concantenated with R's member id is less than a parameter p'.

We use a parameter p' different from our previous bins-based parameter p to address two competing concerns. We want new peers to not overwhelm older ones and so peers select from a tub with decreasing probability the farther away that tub is. However, because new peers select less frequently from tubs that are not adjacent, it may be harder for a new peer to build a relationship with older ones and take advantage of the imbalance ratio. We can resolve the tension between wanting less frequent contact between some peers yet still allowing them to quickly build a relationship by giving a new peer the *same* small set of possible partners every time it chooses a tub. The tracker adjusts p' according to inequality (6.2) so that almost every peer is expected to have at least one non-Byzantine partner in every tub.

$$[1 - p'(1 - F_{byz})]^n \leq \frac{1}{n} \tag{6.2}$$

A new peer S in tub $t_S > 1$ selects a trading partner for round r using two verifiable pseudo-random numbers, $rand_1$ and $rand_2$. First, S uses $rand_1$ to select a tub, exponentially weighting the selection towards its own tub. If S selects a tub $t < t_S - 1$, then S can trade with any peer in tub t that is also in S's view. If

S selects either its own tub or the one immediately preceding its tub, then S uses $rand_2$ to make the bin selection. S maps $rand_2$ to a bin starting from the first bin in tub $t-1$ and ending with S's own bin. From the selected bin, S can trade with any peer in its view.

If every peer knew the global list, then it would be straight-forward to select and verify trading partners. Fortunately, this global knowledge is unnecessary: to select trading partners, a newly joined peer only needs to know the peers in its own view, the epoch in which those peers joined the system, and the indices of those peers in the global list. When a peer S joins the system, S obtains such information directly from the tracker.

To verify that a peer S selects a partner R appropriately, R needs to know S's index in the global membership list. The tracker encodes such information in a *join token* that it gives to S when S joins the system. The join token specifies S's index in the global list for the two epochs until S is part of an epoch's membership list. S includes its join token in its reservation message to R.

6.3 Evaluation

We now show that FlightPath is a robust p2p live streaming protocol. Through experiments on over 500 peers, we demonstrate that FlightPath:

- Reduces jitter by several orders of magnitude compared to BAR Gossip
- Caps peak bandwidth usage to within the constraints of a cable or ADSL connection
- Maintains low jitter and efficiently uses bandwidth despite flash crowds
- Recovers quickly from sudden peer departures
- Continues to deliver a steady stream despite churn

- Tolerates up to 10% of peers acting maliciously

6.3.1 Methodology

We use FlightPath to disseminate a 200 Kbps data stream to several hundred peers distributed across Utah's Emulab and UT Austin's public Linux machines. In most experiments, we use 517 peers, but drop to 443 peers in the churn and Byzantine experiments as the availability of Emulab machines declined. We run each experiment 3 times. When we present cumulative distributions, we combine points from all three experiments. We include standard deviation when doing so keeps figures readable.

In our experiments, rounds last 2 seconds and epochs last 40 rounds. In each round, the source sends 100 Reed-Solomon coded stream updates and 2 linear digests. 50 stream updates are necessary and sufficient to reconstruct the original data. Stream updates expire 10 rounds after they are sent. The source sends each stream update to a random 2.5% of peers. Stream updates are 1072 bytes, linear digests are 1153 bytes, and partial membership lists are 1650 bytes.

We implement FlightPath in Python using MD5 for secure hashes and RSA-full domain hashing with 512 bit keys for digital signatures. Peers exchange public certificates and agree on secret keys for MACs a few seconds before communicating with one another for the first time. Peers also set the budget for how many updates they are willing to upload in a round to $\mu = 100$, which is split evenly across concurrent trades.

6.3.2 Experiments

Steady State Operation: In the first experiment, we run FlightPath on 517 peers to assess its performance under a relatively well-behaved and static environment. Figure 6.15 shows that the average jitter of FlightPath is orders of magnitude

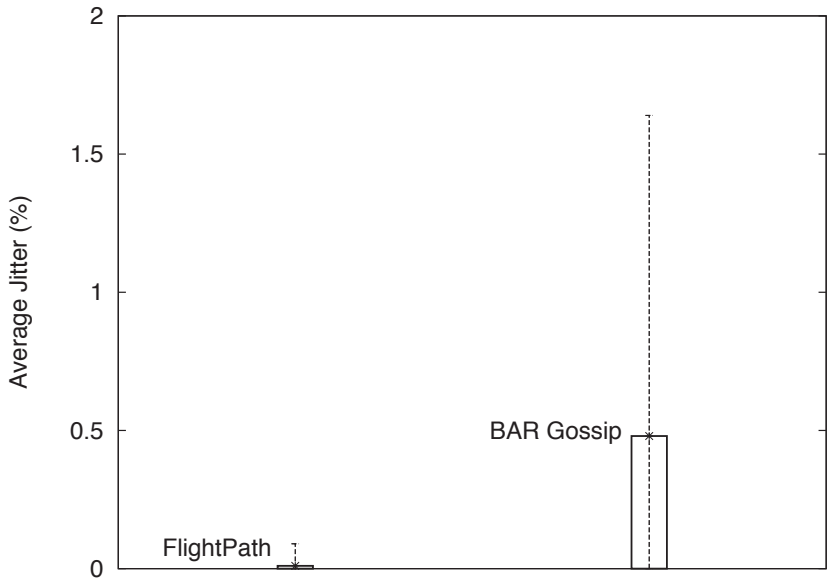

Figure 6.15: Average jitter in FlightPath and BAR Gossip peers. ($n = 517$)

lower than BAR Gossip. Of the three experiments we ran for one hour, the worst jitter was in an experiment in which 1 peer missed 6 seconds of video, 5 peers missed 4 seconds, and 3 peers missed 2 seconds. All jitter events occurred during the first minute. This effect can be explained by the scarcity of items to be traded initially. As the stream continues, peers possess more updates to barter amongst one another. Figure 6.16 confirms that peers use approximately 250 Kbps on average and also depicts cumulative distributions tracing the peak bandwidth of each peer along with curves for the 99 and 95 percentile bandwidth curves. As in Section 6.2.2, the combination of reservations, splitting a peer's need and erasure coding is effective in capping peak bandwidth.

Joins: We now examine how well FlightPath handles joins into the system. In particular, we evaluate how well the tub algorithm (described in Section 6.2.3), handles large populations of peers who seek to join a streaming session all at once.

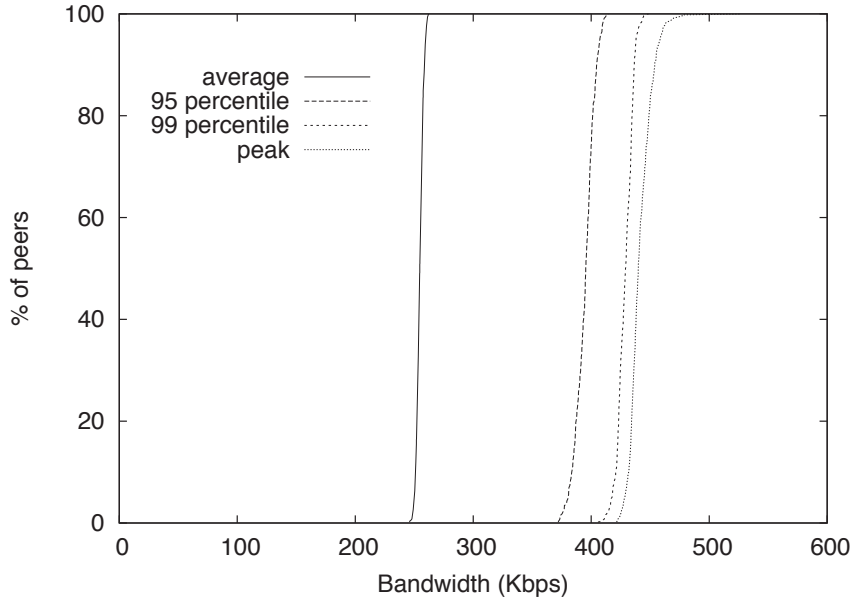

Figure 6.16: Distributions of peers' average, 95 percentile, 99 percentile, and peak bandwidths. ($n = 517$)

In this experiment, we start a session with 50 peers. When the second epoch begins at round 41, varying numbers of peers simultaneously attempt to join the system. As Figure 6.17 illustrates, the average bandwidth of the original peers rises noticeably immediately after the first epoch and settles to a higher level than before. When the fourth epoch begins and new peers are integrated into the membership list, average bandwidth of the original 50 drops back to its previous levels. As shown, FlightPath peers are relatively unaffected by joining events. None of the original 50 peers experienced a jitter event during any of these experiments. Also note that the peak bandwidth across all three runs of each experiment was 482.5 Kbps.

Figure 6.17 shows that the tub algorithm is effective in ensuring that newer peers do not overwhelm older ones. Note that this resilience is easy to achieve at the expense of the newer peers. For example, one could imagine FlightPath without the tub algorithm, requiring new peers to wait over an epoch before participating in

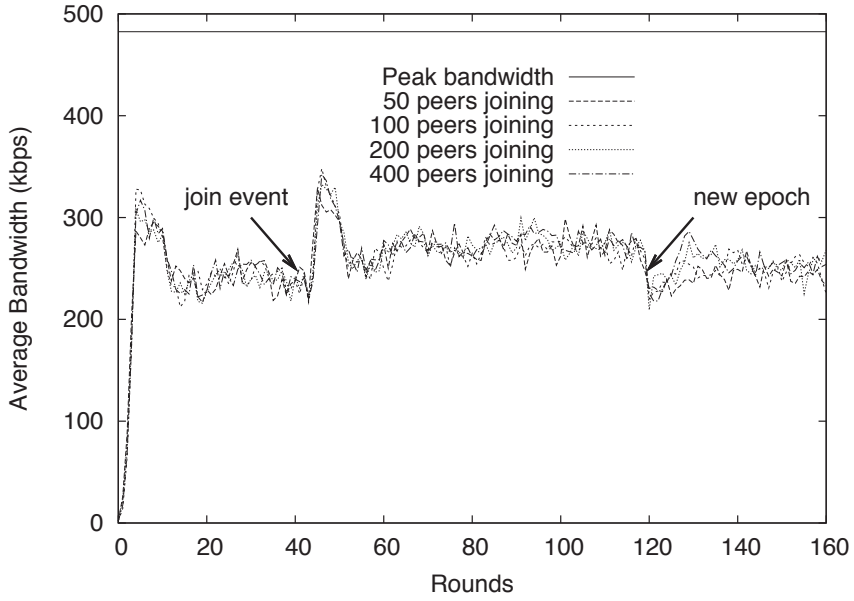

Figure 6.17: Bandwidth of peers already in the system with different sized flash crowds. ($n = 50$)

the system. Such an approach would be very resilient to flash crowds since it would require those crowds to sit idly until the system could accommodate them.

The tub algorithm's contribution is its resilience to flash crowds *while* letting new peers deliver the stream quickly and reliably. Figure 6.18 depicts the number of rounds a peer may have to wait before it begins to deliver a stream reliably. We define the round in which a peer reliably begins to deliver a stream as the first round in which a peer experiences no jitter for three rounds. Interestingly, we see that if more peers join, the average delay decreases. This effect can be explained by our tub algorithm. The peers in the last tub are contacted the least. In the experiment in which only 50 peers join, all of the newly joined peers are in the last tub. The last tub in the experiment with 400 peers joining has a similar problem, but the last tub is masked by the success of the preceding 7 tubs.

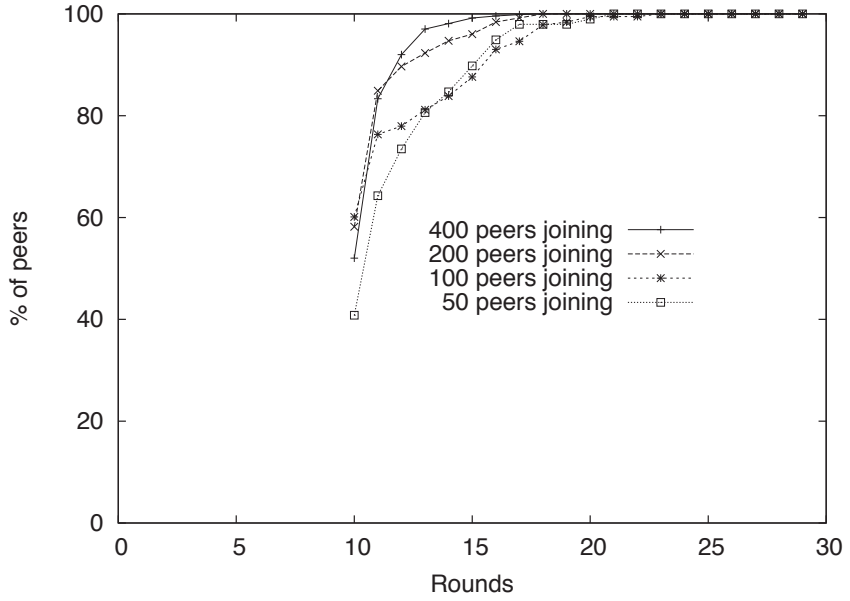

Figure 6.18: CDF of join delays for different size joining crowds. ($n = 50$)

Departures: Figure 6.19 shows FlightPath's resilience to large fractions of a population suddenly departing. Departing peers exit abruptly without notifying the tracker or completing reserved trades. The figure shows the percentage of peers jittered after a massive departure event of 70% and 75% of random peers. We chose these fractions because smaller fractions had little observable effect with respect to jitter. The figure shows that there exists a threshold between 70% and 75% in which FlightPath cannot tolerate any more departures.

FlightPath's resilience to such massive departures is a consequence of a few traits. First, peers discover very quickly whether potential partners have left or not via the reservation system. Second, peers have choice in their partner selection, so they can avoid recently departed peers. Finally, each peer's trouble detector helps in reacting quickly to avoid jitter. Figure 6.20 shows the effect of the trouble predictor. Average bandwidth of remaining peers drops dramatically after the leave event, but then spikes sharply to make up for missed trading opportunities.

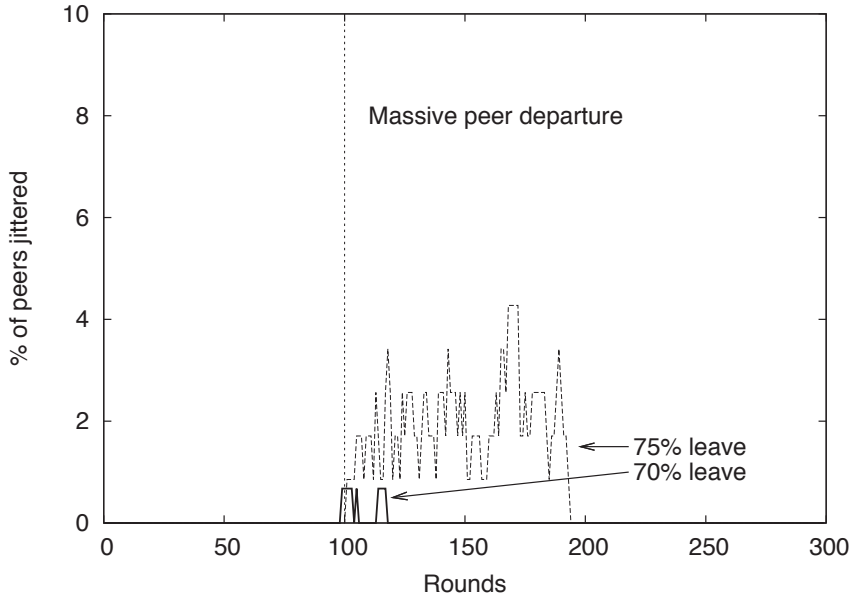

Figure 6.19: Jitter during massive departure. ($n = 517$)

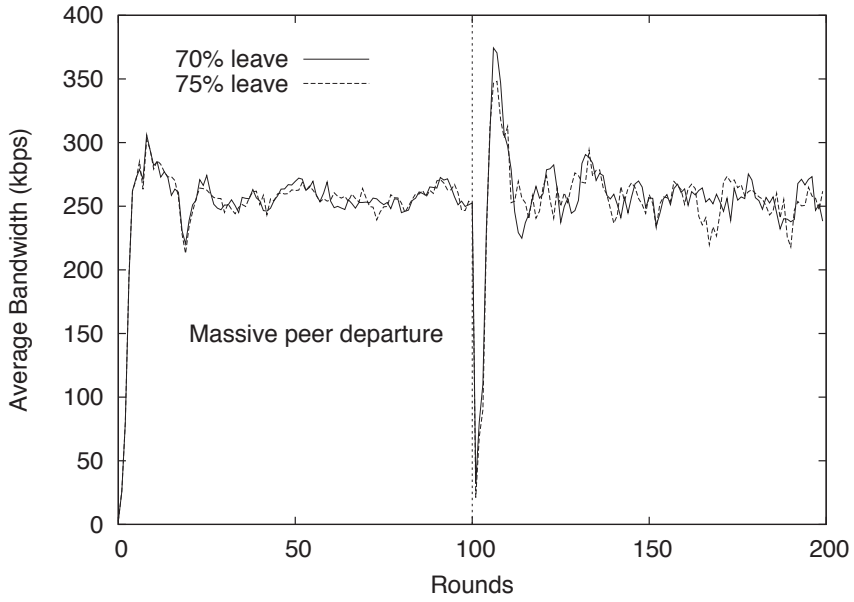

Figure 6.20: Average bandwidth after a massive departure.($n = 517$)

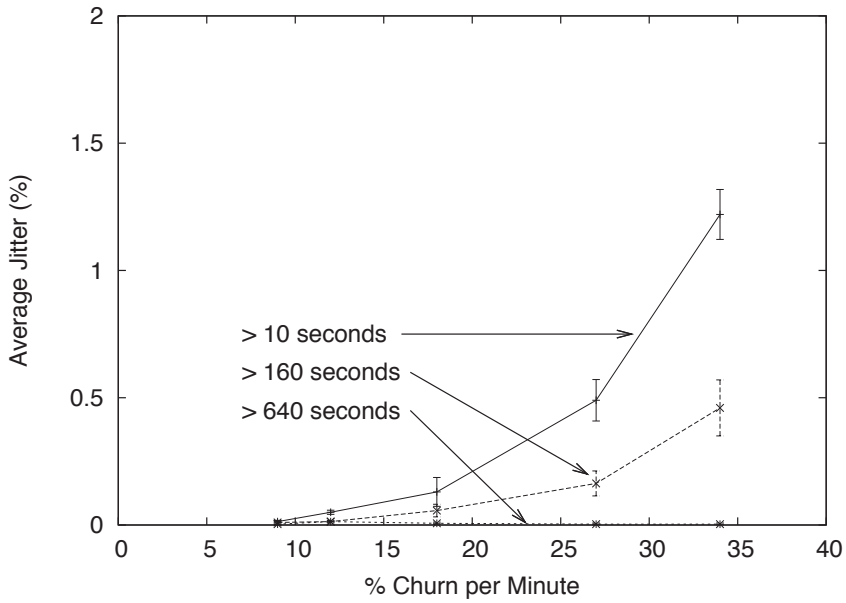

Figure 6.21: Average jitter as churn increases.($n = 443$)

Churn: We now evaluate how FlightPath performs under varying amounts of churn. In our experiments, peers join and then leave after an exponentially random amount of time. We intend this behavior to model users who join a stream and shortly thereafter switch to a different stream. Those who have been in the system longer are more likely to stay. Because short-lived participants are proportionally more affected by their start-up transients, our presentation segregates peers by the amount of time they remain in the system.

Figure 6.21 shows average jitter as we increase churn. The average jitter of peers who join the system for at least 10 seconds steadily increases with churn. Peers who stay in the system for at least 640 seconds experience very little jitter even when 37% of peers churn every minute. In these experiments, all jitter events occur in the first two minutes after joining a streaming session. Afterwards, the chance of being jittered falls to nearly 0.

Figure 6.22 shows that churn does manifest as increasing join delays for new

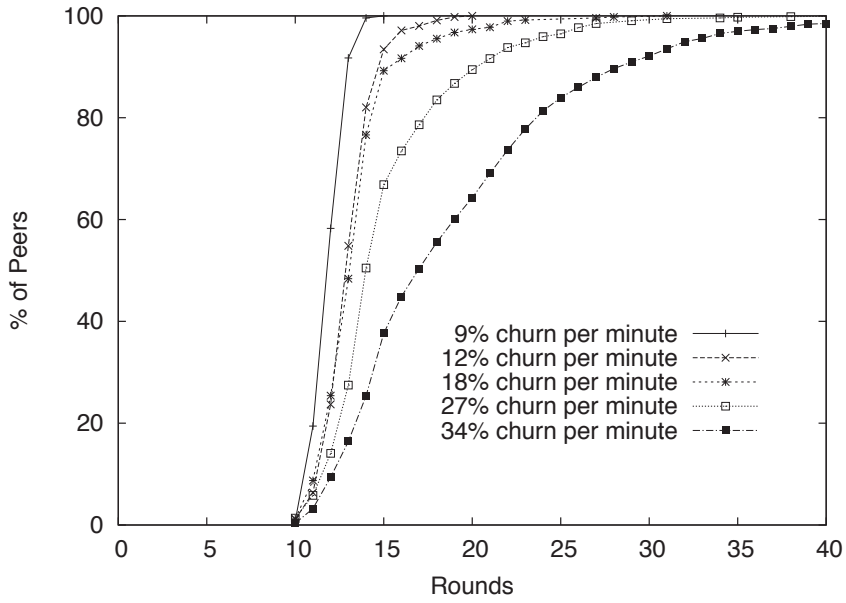

Figure 6.22: Join delay under churn.($n = 443$)

peers. We see that the time needed to join a session is unacceptable under high amounts of churn. This quality points to a weakness of FlightPath and suggests a need for a bootstrapping mechanism for new peers. However, care needs to be exercised in not allowing peers to game the system by abusing the bootstrapping mechanism to obtain updates without uploading.

Malicious attack: In this experiment, we evaluate FlightPath's ability to deliver a stream reliably in the presence of Byzantine peers. While any peer whose utility function is unknown is strictly speaking Byzantine in our model, we are especially interested in understanding how FlightPath behaves under attack, when Byzantine peers behave maliciously.

Although malicious peers cannot make a non-Byzantine peer deliver an inauthentic update, they can harm the system by hurting performance. A malicious peer could hamper the dissemination of updates or increase the overhead for other

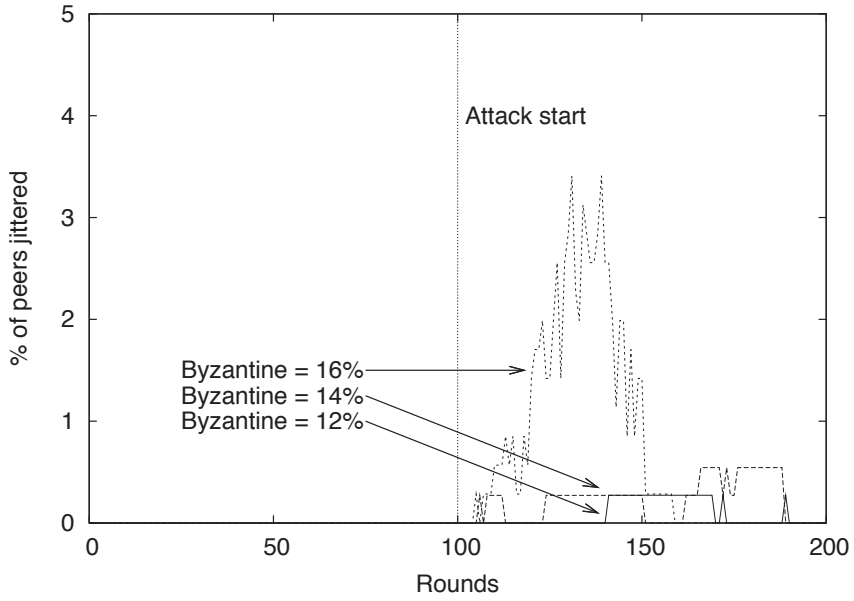

Figure 6.23: Jitter with malicious peers. ($n = 443$)

peers. For example, malicious peers could simultaneously halt all communication to disrupt others. However, we have already seen that FlightPath is very robust to such benign failures. More deviously, malicious peers could launch a coordinated effort to monopolize as many trading opportunities as allowed without making those trades useful to partners. We explore this concerted attack next.

In the following experiment, malicious peers act normally for the first 100 rounds of the protocol. Starting in round 100, they initiate as many trades as they can and respond positively to all trade reservations, seeking to monopolize as many trades in the system as possible. The malicious peers participate in the history exchange phase of a trade but in no subsequent phase. In a history exchange, a malicious peer reports that it has all the updates that are less than 3 rounds old and is missing all the other updates. This strategy commits a large amount of its partner's bandwidth to the exchange, while committing little of the malicious peer's. Ultimately, non-Byzantine peers find trades with Byzantine ones useless.

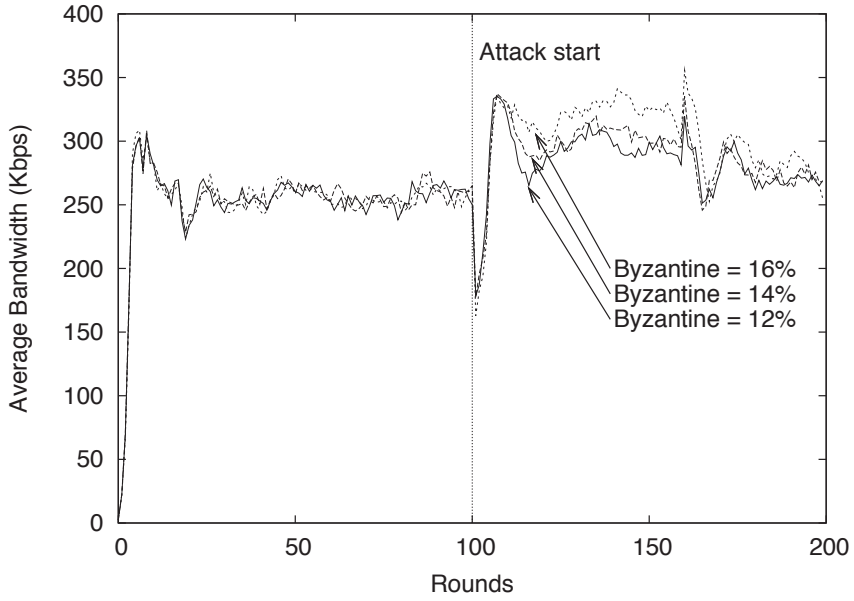

Figure 6.24: Bandwidth with malicious peers. ($n = 443$)

Figure 6.23 shows the percentage of peers jittered when 12%, 14%, and 16% of peers behave in this malicious way. We elide the experiment in which 10% of peers are Byzantine because no peer suffered jitter in those experiments. Figure 6.24, which depicts the average bandwidth of non-Byzantine peers, is similar to the one in which peers abruptly leave the system. The subtle difference is that the average bandwidth used remains higher with more Byzantine peers.

Wide Area Network: Finally, we evaluate how FlightPath performs under wide area network conditions. In this experiment, we use 300 clients on a local area network but delay all packets between clients according to measured Internet latencies. We assign each client a random identity from the 1700+ hosts listed in the King data set of Internet latencies [28]. We use the data set to delay every packet according to its source and destination.

As in the case without added delays, all jitter events occurred in the first

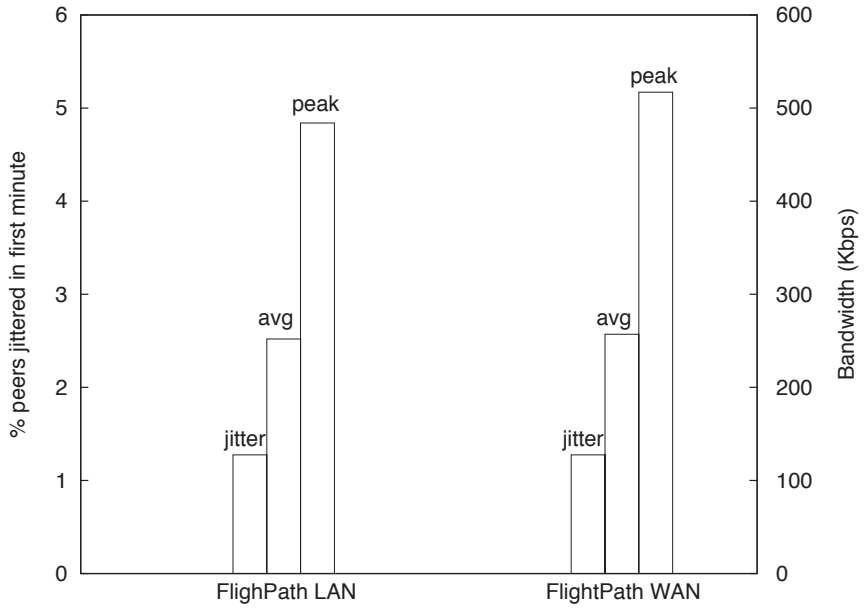

Figure 6.25: Bandwidth under WAN conditions. ($n = 300$)

minute of the experiments. Figure 6.25 depicts the average percentage of peers jittered in the first minute, the average upload bandwidth, and the peak upload bandwidth for our experiments with the added delays and without. Aside from a slight increase (almost 10 Kbps) in average upload bandwidth, peak upload bandwidth rose by approximately 40 Kbps. These increases are the result of some exchanges not completing by the end of a round, requiring peers involved to make up for the loss in subsequent rounds.

6.4 Equilibria Analysis

In contrast to previous rigorous approaches to dissuade rational deviation, FlightPath does not ensure that every step of the protocol is in every peer's best interest. Indeed, it is easy to imagine circumstances in which a peer might benefit from deviating, for example, by setting the plead flag early to increase the likelihood that

a selected peer will accept its invitation. Instead, FlightPath ensures an ϵ-Nash equilibrium in which no peer can significantly improve its overall utility regardless of how it makes these individual choices.

The high level argument is simple. A peer can only increase its utility by obtaining more benefit (receiving less jitter) or reducing cost (uploading fewer bytes). Since we engineered FlightPath to provide very low jitter in a wide range of environments, a peer has very little ability to obtain more benefit. With respect to decreasing costs, we structure trades so that they are relatively balanced.

We now develop this argument more formally to bound the added utility that can be gained by a peer who deviates. We analyze FlightPath in the steady state case and ignore transient start-up effects or end game scenarios, which would matter little in the overall utility of watching something as long as a movie.

6.4.1 Defining ϵ

We begin by revisiting the utility function $u_i(\mathbf{s}) = (1 - j_i)\beta - w_i\kappa$. Recall that j_i is the average number of jitter events per minute that i experiences, β is the benefit from watching a jitter-free stream, w_i is $i's$ average upload bandwidth in kilobits per second, and κ is the cost per Kbps. Let $\mathbf{s}$ be the strategy profile corresponding to when all peers obey the FlightPath protocol and let us consider a peer i. We desire that no matter how clever i may be, i expects to benefit little from following any strategy $s_i^* \neq s_i$. More formally,

$$\forall s_i^* \in S_i : u_i(\mathbf{s}_{-i}, s_i^*) \leq (1 + \epsilon)u_i(s)$$

We can therefore bound ϵ as follows:

$$\epsilon \geq \frac{u_i^*(\mathbf{s}_{-i}, s_i^*) - u_i(\mathbf{s})}{u_i(\mathbf{s})} = \frac{((1 - j_i^*)\beta - w_i^*\kappa) - ((1 - j_i)\beta - w_i\kappa)}{(1 - j_i)\beta - w_i\kappa} \tag{6.3}$$

We simplify equation 6.3 by assuming that a cheating peer experiences no jitter, i.e., $j_i^* = 0$.

$$\epsilon \leq \frac{j_i\beta - (w_i^* - w_i)\kappa}{(1 - j_i)\beta - w_i\kappa} \tag{6.4}$$

We then divide the numerator and denominator by the expected cost to understand how epsilon changes as a function of the benefit to cost ratio $c = \frac{(1-j_i)\beta}{w_i\kappa}$, which we assume to be greater than 1.

$$\epsilon \leq \frac{\frac{cj_i}{1-j_i} + \frac{w_i - w_i^*}{w_i}}{c - 1} \tag{6.5}$$

Inequality 6.5 captures ϵ as a function of the benefit-to-cost ratio c, the expected number of jitter events per minute j_i, and the proportional savings in cost $\frac{w_i^* - w_i}{w_i}$. We can gain an intuitive understanding of this bound on ϵ by considering three cases. First, if obeying the FlightPath protocol provides no jitter and no cost can be saved by cheating, then FlightPath would be a Nash equilibrium for all benefit-to-cost ratios greater than 1. Second, if cost matters little compared to benefit, $c \gg 1$, then ϵ is essentially bound by $\frac{j_i}{1-j_i}$, indicating that as expected jitter increases so does the gap in utilities between obeying and cheating. Third, if obedience provides no jitter, the important term is the proportional savings in cost achieved by cheating, which matters less and less as c increases.

We present our analysis in two stages. In the first, we provide a lower bound on the bandwidth w_i^* required by i when following s_i^*. In the second stage, we develop a conservative estimate for the jitter a peer expects to see and the upload bandwidth a peer expects to use.

6.4.2 A lower bound for w_i^*

In FlightPath, a peer who cheats still expects to pay costs. We purposefully structure trades so that a cheating peer has to pay most of the cost in an exchange before obtaining anything useful from the interaction. We take advantage of this fact when we derive the minimum amount of upload bandwidth a peer needs to use so to obtain all the updates it needs.

Recall that each peer needs `ups_per_round` stream updates for every round to avoid jitter events and that the source distributes linear digests so that peers can authenticate stream updates. To simplify the analysis, we assume that peers obtain all the linear digests they need directly from the source. However, peers may have to trade for partial membership lists to continue participating in the system. In a system of size n, a peer needs $\lceil \frac{n}{\texttt{entries\_per\_partial}} \rceil$ partial membership lists every `epoch_len` rounds, where `entries_per_partial` is the number of peer entries in each partial membership list; in our prototype, we include at most 30 entries in each partial membership list. In a given epoch, a peer expects to have to gather `missing_per_epoch` updates via trades, where $\texttt{missing\_per\_epoch} = (1 - \texttt{seed\_frac})(\texttt{ups\_per\_round} \times \texttt{epoch\_len} + \lceil \frac{n}{\texttt{entries\_per\_partial}} \rceil$.

Assuming that i is lucky or clever enough to upload no more updates than it has to in all trades, i still uploads at least $\texttt{min\_upload} = \lceil \frac{\texttt{missing\_per\_epoch}}{1+\texttt{imb\_ratio}} \rceil$ updates in every epoch. Since i expects no peer to upload more than `budget` updates in a single trade, i therefore expects to participate in at least $\lceil \frac{\texttt{min\_upload}}{\texttt{budget}} \rceil$ trades in each epoch.

We combine the above expressions with empirical numbers for the minimum cost of each trade and how that cost grows as the number of updates sent increases. Taking into account reservation messages, history exchange messages, briefcases, promises, and keys, each trade costs at least 698 bytes of upload bandwidth with each update uploaded costing an additional 1136 bytes. Using the parameter values

Parameter	Description	Value
ups_per_round	num stream updates per round needed	50
num_coded_blocks	num stream updates per round	100
entries_per_partial	peer entries per partial membership list	30
seed_frac	fraction of updates received from source	5%
budget	max num of updates sent in a round	100
imb_ratio	imbalance ratio	10%
epoch_len	epoch length in rounds	40
r_len	round length in seconds	2
n	system size	500

Table 6.1: Summary of the analysis parameters.

listed in Table 6.4.2, a peer who cheats on average participates in at least 18 trades every epoch and uploads 1741 updates, meaning that $w_i^* \geq 199$ Kbps.

6.4.3 An estimate for j_i and w_i

We develop a bound for ϵ by conservatively estimating expected jitter j_i and the expected upload bandwidth w_i. It is difficult to establish a tight bound on both these values analytically because system dynamics, system size, and randomness quickly make a pure mathematical characterization intractable. We therefore take a conservative approach.

We assume the expected jitter number of jitter events per minute, j_i, is 0.01, indicating that on average a peer expects a jitter event once every ten minutes. The observed jitter in our prototype is orders of magnitude less than what we assume here.

We further assume that peers acquire the updates they need by participating in trades every round. On average, a peer needs to gather $\lceil \frac{\texttt{missing\_per\_epoch}}{\texttt{epoch\_len}} \rceil$ updates in each round. Ideally, a peer would obtain these updates in one trade, avoiding the fixed cost for additional trades and not risking having to receive redundant updates. In this analysis, we assume instead that a peer acquires the updates it

needs in each round through 4 concurrent trades (the maximum number of allowed concurrent trades). Furthermore, the updates it needs are for the same round and evenly divided across these concurrent trades to increase the likelihood that updates acquired at the same time overlap.

Given the parameters listed in Table 6.4.2, $\lceil \frac{\texttt{missing\_per\_epoch}}{\texttt{epoch\_len}} \rceil = 48$ and a peer expects that it needs to receive 60 updates spread evenly across 4 concurrent trades in order to acquire 48 unique updates for each round. Assuming that trades are relatively balanced for peers who obey the protocol, a peer expects to upload at approximately $w_i = 291$ Kbps.

Figure 6.26 uses the derived bounds to show epsilon as a function of the benefit-to-cost ratio. For $\epsilon = \frac{1}{10}$, solving for c in Inequality 6.5 indicates that FlightPath is a $\frac{1}{10}$-Nash equilibrium as long as the user values the stream at least 4.63 times as much as the bits uploaded to participate in the system. We also provide a curve for ϵ using empirical values for expected jitter and average bandwidth instead of analytical ones. In practice, it appears as though FlightPath remains a $\frac{1}{10}$-Nash equilibrium for $c \geq 3.07$.

6.5 Discussion

Recall that our shift from exact equilibria to their approximate counterparts is the result of a broad frustration with the former approach. In BAR Gossip, our mechanisms to meet an exact solution concept incur high overheads while providing no way to deal with dynamic membership. Moreover, our inability to show Optimistic Push is a Nash equilibrium even under a constrained strategy space is disappointing.

We address many of the shortcomings of exact approaches by designing systems to be approximate equilibria. The practical advantages are clear; we can engineer practical solutions that are flexible enough to handle many adverse situations, such as churn and Byzantine peers. In addition to those practical benefits,

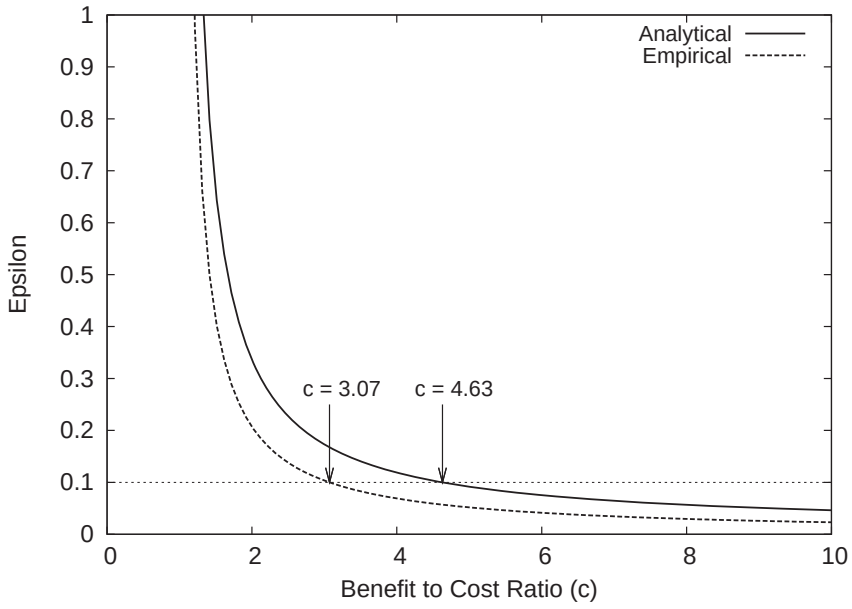

Figure 6.26: ϵ as a function of the benefit to cost ratio.

approximate equilibria also have more subtle theoretical ones. The analysis we provide in the previous section allows for rational peers who may pursue strategies that consider concurrent and future trades. Note that it remains difficult and perhaps intractable to determine the best way for a peer to cheat. However, demonstrating that a system, such as FlightPath, is an approximate equilibrium does not require finding the best strategy to cheat, but rather bounding the gains possible from any cheating to a small factor.

Approximate equilibria free us from having to micro-manage our protocols so we no longer have to ensure that obeying each step is in every rational peer's best interest. That freedom affects how we design systems and how we describe them, a change that is apparent when comparing the chapters on Balanced Exchange and FlightPath. In the former, we interpose lemmas and proofs with prose that describes how a balanced exchange works and why rational peers would obey. In this chapter, we describe FlightPath's basic trading protocol and propose several modifications,

focusing on systems issues such as performance and bandwidth and leaving a game theoretic analysis until the last pages of this chapter. Our contributions in FlightPath are a good start to designing practical approaches to curb rational deviations in p2p systems. In the next and final chapter, we discuss some problems that remain in combining mechanism design with system design.

Chapter 7

Conclusion

This dissertation explores how to design p2p systems to tolerate both Byzantine peers and rational peers. As we design mechanisms to cope with these peers in a large-scale system, two themes emerge: obedience and choice. In BAR Gossip, we focus on obedience, creating the first p2p live streaming system that tolerates Byzantine and rational peers. However, our dictatorial approach leads to several practical and theoretical limitations. Those shortcomings reflect a broader frustration with existing works that focus on designing systems to be exact equilibria.

We next temper obedience with controlled amounts of choice, and use approximate equilibria to drive the design of FlightPath. This approach lets us reduce jitter by several orders of magnitude, use bandwidth more efficiently, handle churn, and adapt to attacks. By switching to approximate equilibria, we can retain the rigor of a formal approach while providing enough flexibility to engineer practical solutions. BAR Gossip and FlightPath represent good starts in building robust p2p systems. Yet, many problems remain. We present some open questions below and put forth initial answers that may lead to interesting avenues for future work.

Why do none of the presented mechanisms allow peers to be altruistic? Many p2p systems rely on the existence and participation of users who contribute more than is required. Taking advantage of user altruism can increase the robustness and performance of a system and is a desirable goal. Yet, we need to be careful how we hand out such generosity so that it is not (grossly) abused by rational peers seeking to contribute less.

How can we add allow altruistic actions without compromising the built-in incentives? A potential way to introduce altruism while avoiding previous pitfalls is to have altruistic actions benefit every participant evenly. In a multicast system such as FlightPath, altruistic peers could relay younger updates to small subsets of random peers, thereby helping the initial spread of data in the system. Such a mechanism leaves no room for abuse, as peers cannot control when they receive these updates. However, it also leaves no avenue for peers to ask for help when they need it most.

How can we allow peers with few resources to be a part of a system? Dealing with resource-poor nodes requires us to answer a basic question: do we want to support nodes who cannot support themselves? If the answer is no, then the resulting system population would be a self-selecting group capable of meeting their own needs. If, however, the answer is yes, we need to decide where to obtain extra resources necessary to support resource-poor users. It is desirable that the additional resources comes from the other peers in the system.

How can we incentivize users who have a surplus of resources to contribute those resources for the good of others? A possible approach is to impose a progressive tax as Chu et al. [17] propose. They, however, leave how to enforce such a tax scheme open. It may be possible to separate a p2p service into

multiple layers as proposed by Gorinsky et al. [27]. In this scheme, peers have an incentive to participate in as many layers as possible. However, a peer can only gain access to a layer by participating sufficiently at all lower layers. We can tax peers who can gain access to higher layers more heavily than other peers, thereby forcing those with more to help those with less.

Can the techniques used in BAR Gossip and FlightPath be extended to more general content distribution networks? BAR Gossip and FlightPath create ways for users to barter pieces of information with one another. Because both systems focus exclusively on streaming live data in which peers are interested in the same data at the same time, bartering is a simple and effective mechanism to create incentives for peers to disseminate information. However, in settings such as content distribution networks or video-on-demand applications, peers may not be interested in the same data at the same time; direct swaps may not be possible and a currency or credit-based system may be better suited to encourage cooperation.

What are potential problems in using credits to build a robust p2p system? Two common critiques of credit-based systems are that they can offer loopholes which participants can abuse or are overly complicated. Many systems offer a small amount of credit to new users to bootstrap them into the system. Without appropriate safeguards against Sybil attacks though, these systems can make it easy to free-ride. Additionally, a constant influx of credit due to churn can lead to complex, ad hoc techniques to manage inflation and deflation.

This dissertation set out with a simple goal: build a p2p live streaming system that tolerates Byzantine peers and rational peers. Disappointed with systems that justify their mechanisms informally, we adopt a principled approach based on game theory and traditional fault-tolerance. In contrast to what the state-of-the-art suggests,

this work demonstrates that we do not have to sacrifice rigor to engineer Byzantine and rational-tolerant systems that perform well and operate efficiently. Key to our success is a careful balance between enforcing obedience and providing choice enabled by using approximate equilibria.

Bibliography

[1] Eytan Adar and Bernardo A. Huberman. Free riding on Gnutella. *First Monday*, 5(10):2–13, October 2000.

[2] R. Ahlswede, Ning Cai, S.-Y.R. Li, and R.W. Yeung. Network information flow. *Information Theory, IEEE Transactions on*, 46(4):1204–1216, Jul 2000.

[3] A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J-P. Martin, and C. Porth. BAR fault tolerance for cooperative services. In *Proceedings of the 20th Symposium on Operating Systems Principles*, pages 45–58, October 2005.

[4] N. Alon, J. Edmonds, and M. Luby. Linear time erasure codes with nearly optimal recovery. In *Proceedings of the 36th Symposium on Foundations of Computer Science*, page 512, Washington, DC, USA, 1995. IEEE Computer Society.

[5] Azureus. http://azureus.sourceforge.net.

[6] Mahesh Balakrishnan, Ken Birman, Amar Phanishayee, and Stefan Pleish. Ricochet: Lateral error correction for time-critical multicast. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation*, Cambridge, MA, USA, 2007. IEEE Computer Society.

[7] Mahesh Balakrishnan, Stefan Pleisch, and Ken Birman. Slingshot: Time-

critical multicast for clustered applications. In *Proceedings of the 4th Symposium on Network Computing Applications*, pages 205–214, 2005.

[8] Mihir Bellare and Phillip Rogaway. Random oracles are practical: a paradigm for designing efficient protocols. In *Proceedings of the 1st Conference on Computer and Communications*, pages 62–73, New York, NY, USA, 1993. ACM Press.

[9] Mihir Bellare and Phillip Rogaway. The exact security of digital signatures — how to sign with RSA and Rabin. *Lecture Notes in Computer Science*, 1070:399–416, 1996.

[10] Kenneth P. Birman, Mark Hayden, Oznur Oskasap, Zhen Xiao, Mihai Budiu, and Yaron Minsky. Bimodal multicast. *ACM Transactions on Computer Systems*, 17(2):41–88, May 1999.

[11] Edward Bortnikov, Maxim Gurevich, Idit Keidar, Gabriel Kliot, and Alexander Shraer. Brahms: byzantine resilient random membership sampling. In *Proceedings of the 27th Symposium on Principles of Distributed Computing*, pages 145–154, New York, NY, USA, 2008. ACM.

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

[13] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. SplitStream: high-bandwidth multicast in cooperative environments. In *Proceedings of the 19th Symposium on Operating Systems Principles*, pages 298–313. ACM Press, 2003.

[14] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In

Proceedings of the 3rd Symposium on Operating Systems Design and Implementation, pages 173–186, February 1999.

[15] Steve Chien and Alistair Sinclair. Convergence to approximate nash equilibria in congestion games. In *Proceedings of the 18th ACM-SIAM Symposium on Discrete Algorithms*, pages 169–178, Philadelphia, PA, USA, 2007. SIAM.

[16] Philip A. Chou, Yunnan Wu, and Kamal Jain. Practical network coding. In *Proceedings of the 41st Allerton Conference on Communication, Control, and Computing*, October 2003.

[17] Yang-hua Chu, John Chuang, and Hui Zhang. A case for taxation in peer-to-peer streaming broadcast. In *Proceedings of the ACM SIGCOMM Workshop on Practice and Theory of Incentives in Networked Systems*, pages 205–212, New York, NY, USA, 2004. ACM.

[18] Bram Cohen. Incentives build robustness in BitTorrent. In *Proceedings of the 1st Workshop on the Economics of Peer-to-Peer Systems*, June 2003.

[19] Landon P. Cox and Brian D. Noble. Samsara: honor among thieves in peer-to-peer storage. In *Proceedings of the 19th Symposium on Operating Systems Principles*, pages 120–132, 2003.

[20] Constantinos Daskalakis, Aranyak Mehta, and Christos Papadimitriou. A note on approximate nash equilibria. In *Proceedings of the 2nd International Workshop on Internet and Network Economics*, 2006.

[21] A. Demers, D. Greene, C. Houser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the 11th Symposium on Operating Systems Principles*, August 1987.

[22] John R. Douceur. The Sybil attack. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems*, pages 251–260. Springer-Verlag, 2002.

[23] P. Eugster, S. Handurukande, R. Guerraoui, A. Kermarrec, and P. Kouznetsov. Lightweight probabilistic broadcast. In *Proceedings of the 31st International Conference on Dependable Systems and Networks*, pages 254–269, July 2001.

[24] Ayalvadi J. Ganesh, Anne-Marie Kermarrec, and Laurent Massouli. Peer-to-peer membership management for gossip-based protocols. *IEEE Transactions on Computing*, 52(2):139–149, 2003.

[25] Benot Garbinato and Ian Rickebusch. Impossibility results on fair exchange. Technical Report DOP-20051122, Université de Lausanne, Distributed Object Programming Lab.

[26] C. Gkantsidis and P.R. Rodriguez. Network coding for large scale content distribution. *Proceedings of the 24th IEEE Conference on Computer Communications*, 4:2235–2245 vol. 4, March 2005.

[27] S. Gorinsky, K.K. Ramakrishnan, and H. Vin. Addressing heterogeneity and scalability in layered multicast congestion control. Technical Report TR2000-31, Department of Computer Sciences, University of Texas at Austin, November 2000.

[28] Krishna P. Gummadi, Stefan Saroiu, and Steven D. Gribble. King: estimating latency between arbitrary internet end hosts. *SIGCOMM Computer Communications Review*, 32(3):11–11, 2002.

[29] I. Gupta, K. Birman, and R. van Renesse. Fighting fire with fire: using randomized gossip to combat stochastic scalability limits. *Journal of Quality and Reliability Engineering International*, 18(3):165–184, 2002.

[30] Indranil Gupta, Anne-Marie Kermarrec, and Ayalvadi J. Ganesh. Efficient and adaptive epidemic-style protocols for reliable and scalable multicast. *IEEE Transactions on Parallel and Distributed Systems*, 17(7):593–605, 2006.

[31] Ahsan Habib and John Chuang. Incentive mechanism for peer-to-peer media streaming. In *12th IEEE International Workshop on Quality of Service.*, 2004.

[32] Maya Haridasan and Robbert van Renesse. Defense against intrusion in a live streaming multicast system. In *Proceedings of the 6th International Conference on Peer-to-Peer Computing*, pages 185–192, Washington, DC, USA, 2006. IEEE Computer Society.

[33] Maya Haridasan and Robbert van Renesse. Securestream: An intrusion-tolerant protocol for live-streaming dissemination. *Computer Communications*, 31(3):563–575, 2008.

[34] John Jannotti, David K. Gifford, Kirk L. Johnson, M. Frans Kaashoek, and Jr. James W. O'Toole. Overcast: reliable multicasting with on overlay network. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, pages 14–14, Berkeley, CA, USA, 2000. USENIX Association.

[35] Håvard Johansen, André Allavena, and Robbert van Renesse. Fireflies: scalable support for intrusion-tolerant network overlays. *SIGOPS Operating Systems Review*, 40(4):3–13, 2006.

[36] Kazaa. http://www.kazaa.com.

[37] Kazaa Lite. http://en.wikipedia.org/wiki/Kazaa_Lite.

[38] Idit Keidar, Roie Melamed, and Ariel Orda. Equicast: Scalable multicast with selfish users. In *Proceedings of the 25th Symposium on Principles of Distributed Computing*, 2006.

[39] Dejan Kostić, Adolfo Rodriguez, Jeannie Albrecht, and Amin Vahdat. Bullet: high bandwidth data dissemination using an overlay mesh. In *Proceedings of the 19th Symposium on Operating Systems Principles*, pages 282–297, New York, NY, USA, 2003. ACM.

[40] Joao Leitao, Jose Pereira, and Luis Rodrigues. Hyparview: A membership protocol for reliable gossip-based broadcast. In *Proceedings of the 37th International Conference on Dependable Systems and Networks*, pages 419–429, Washington, DC, USA, 2007. IEEE Computer Society.

[41] Dave Levin, Katrina LaCurts, Neil Spring, and Bobby Bhattacharjee. Bittorrent is an auction: analyzing and improving bittorrent's incentives. *SIGCOMM Computer Communications Review*, 38(4):243–254, 2008.

[42] Dave Levin, Rob Sherwood, and Bobby Bhattacharjee. Fair file swarming with FOX. In *Proceedings of the 5th International Workshop on Peer-to-Peer Systems*, Feb 2006.

[43] Harry C. Li, Allen Clement, Edmund Wong, Jeff Napper, Indrajit Roy, Lorenzo Alvisi, and Michael Dahlin. BAR Gossip. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 191–204, November 2006.

[44] LimeWire. http://www.limewire.com/.

[45] Meng-Jang Lin and Keith Marzullo. Directional gossip: Gossip in a wide area network. In *Proceedings of the 2nd European Dependable Computing Conference*, pages 364–379, 1999.

[46] W.S. Lin, H.V. Zhao, and K.J.R. Liu. A game theoretic framework for incentive-based peer-to-peer live-streaming social networks. In *ICASSP '08: International Conference on Acoustic, Speech, and Signal Processing*. IEEE, 2008.

[47] Zhengye Liu, Yanming Shen, Shivendra S. Panwar, Keith W. Ross, and Yao Wang. Using layered video to provide incentives in p2p live streaming. In *Proceedings of the 2007 Workshop on Peer-to-Peer Streaming and IP-TV*, pages 311–316, New York, NY, USA, 2007. ACM.

[48] Livestation. http://www.livestation.com.

[49] Thomas Locher, Patrick Moor, Stefan Schmid, and Roger Wattenhofer. Free Riding in BitTorrent is Cheap. In *5th Workshop on Hot Topics in Networks (HotNets), Irvine, California, USA*, November 2006.

[50] Michael G. Luby, Michael Mitzenmacher, M. Amin Shokrollahi, Daniel A. Spielman, and Volker Stemann. Practical loss-resilient codes. In *Proceedings of the 29th Symposium on Theory of Computing*, pages 150–159. ACM Press, 1997.

[51] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, 1998.

[52] Roie Melamed and Idit Keidar. Araneola: A scalable reliable multicast system for dynamic environments. In *Proceedings of the 3rd Symposium on Network Computing Applications*, pages 5–14, Washington, DC, USA, 2004. IEEE Computer Society.

[53] Silvio Micali, Salil Vadhan, and Michael Rabin. Verifiable random functions. In *Proceedings of the 40th Symposium on Foundations of Computer Science*, page 120, Washington, DC, USA, 1999. IEEE Computer Society.

[54] Animesh Nandi, Tsuen-Wan "Johnny" Ngan, Atul Singh, Peter Druschel, and Dan S. Wallach. Scrivener: Providing incentives in cooperative content distribution systems. In *Proceedings of the 6th International Conference on Middleware*, Grenoble, France, November 2005.

[55] Napster. http://www.napster.com.

[56] John Nash. Non-cooperative games. *The Annals of Mathematics*, 54:286–295, September 1951.

[57] Tsuen-Wan Johnny Ngan, Dan S. Wallach, and Peter Druschel. Incentives-compatible peer-to-peer multicast. In *Proceedings of the 2nd Workshop on the Economics of Peer-to-Peer Systems*, June 2004.

[58] Derek C. Oppen and Yogen K. Dalal. The clearinghouse: a decentralized agent for locating named objects in a distributed environment. *ACM Transactions on Information Systems*, 1(3):230–253, 1983.

[59] Henning Pagnia and Felix C. Gärtner. On the impossibility of fair exchange without a trusted third party. Technical Report TUD-BS-1999-02, Darmstadt University of Technology, Department of Computer Science, Darmstadt, Germany, March 1999.

[60] V. Pai, K. Kumar, K. Tamilmani, V. Sambamurthy, and A. Mohr. Chainsaw: Eliminating trees from overlay multicast. In *Proceedings of the 4th International Workshop on Peer-to-Peer Systems*, February 2005.

[61] Kunwoo Park, Sangheon Pack, and Taekyoung Kwon. Climber: An incentive-based resilient peer-to-peer system for live streaming services. February 2008.

[62] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of the 16th Symposium on Operating Systems Principles*, 1997.

[63] Fabio Pianese, Joaquin Keller, and Ernst W. Biersack. Pulse, a flexible p2p live streaming system. In *9th IEEE Global Internet Workshop*, 2006.

[64] Michael Piatek, Tomas Isdal, Thomas Anderson, Arvind Krishnamurthy, and Arun Venkataramani. Do incentives build robustness in bittorrent? In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation*, 2007.

[65] Atul Puri and Alexandros Eleftheriadis. MPEG-4: an object-based multimedia coding standard supporting mobile applications. *Mobile Networks and Applications*, 3(1):5–32, 1998.

[66] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218:329–350, 2001.

[67] F. B. Schneider. Implementing fault–tolerant services using the state machine approach: A tutorial. *Computing Surveys*, 22(3):299–319, September 1990.

[68] Jeffrey Shneidman, David C. Parkes, and Laurent Massoulie. Faithfulness in internet algorithms. In *Proceedings of the ACM SIGCOMM Workshop on Practice and Theory of Incentives in Networked Systems*, pages 220–227, Portland, USA, 2004.

[69] Atul Singh, Miguel Castro, Peter Druschel, and Antony Rowstron. Defending against eclipse attacks on overlay networks. In *Proceedings of the 11th SIGOPS European workshop: beyond the PC*, page 21, New York, NY, USA, 2004. ACM Press.

[70] Michael Sirivianos, Jong Han Park, Rex Chen, and Xiaowei Yang. Free-riding in bittorrent networks with the large view exploit. February 2007.

[71] Michael Sirivianos, Jong Han Park, Xiaowei Yang, and Stanislaw Jarecki. Dandelion: cooperative content distribution with robust incentives. In *ATC'07: 2007 USENIX Annual Technical Conference on Proceedings of the USENIX*

Annual Technical Conference, pages 1–14, Berkeley, CA, USA, 2007. USENIX Association.

[72] Skype. http://www.skype.com.

[73] Gene Tsudik. Message authentication with one-way hash functions. In *INFOCOM*, pages 2055–2059, May 1992.

[74] Robbert van Renesse, Kenneth P. Birman, and Werner Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems*, 21(2):164–206, 2003.

[75] Vivek Vishnumurthy, Sangeeth Chandrakumar, and Emin Gun Sirer. KARMA: A secure economic framework for p2p resource sharing, June 2003.

[76] Werner Vogels, Robbert van Renesse, and Ken Birman. The power of epidemics: robust communication for large-scale distributed systems. *SIGCOMM Computer Communications Review*, 33(1):131–135, 2003.

[77] Spyros Voulgaris, Spyros Voulgaris, Daniela Gavidia, Daniela Gavidia, Maarten Van Steen, and Maarten Van Steen. Cyclon: Inexpensive membership management for unstructured p2p overlays. *Journal of Network and Systems Management*, 13:2005, 2005.

[78] Haifeng Yu, Michael Kaminsky, Phillip B. Gibbons, and Abraham Flaxman. Sybilguard: Defending against sybil attacks via social networks. In *SIGCOMM Computer Communications Review*, September 2006.

Vita

Harry Li was born in 1980 in Manhattan, New York. His father and mother, Tak-po and Sui-yee, moved the entire family out to Nesconset on Long Island when he was about 4. Although he remembers very little from that time, he is grateful that he has had countless more opportunities in his life than either of his parents have had in theirs. Harry is also grateful to have Raymond, his older brother, who has looked out for him these twenty-eight years.

After eighteen socially awkward years of life, Harry attended Brown University in 1998. Not knowing what academic field to pursue in college, he ended up having a very relaxing first semester. However, after watching the movie Hackers while on winter break, Harry was inspired and decided to devote his education to computer science. That devotion did not preclude an abundance of free time sophomore year, leading to his joining of Brown's fencing team, perhaps the best whim of his life. Despite being an embarassment to the sport, he stayed on the team and ultimately met Meg Robinson in September of 2004. He promptly asked her out. She quite quickly said no. They fell in love the following summer. During that summer, Harry had the pleasure of working with Shriram Krishnamurthi and Kathi Fisler. Their guidance led to Harry's first taste of research and ultimately to his desire to pursue a Ph.D.

In 2002, Harry graduated from Brown with honors and joined the University of Texas at Austin. His interest in theory and childlike desire to make things led him

to distributed systems. During his graduate school career, his advisors—Lorenzo Alvisi and Mike Dahlin—made him better than he thought he could ever be. He hopes to make them proud later in his career.

In June of 2006, Harry asked Meg to marry him. She quite quickly said yes and soon joined him in Austin. That winter saw the addition of a third member to their family, Tinkerbell. In 2008, Harry and Meg became husband and wife and moved to California for Meg to study educational policy at Stanford. Although their new life is wonderful in so many ways, they miss their Texan friends dearly.

Permanent Address: 785 Live Oak Avenue Apt. C
Menlo Park, CA 94025

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay, James A. Bednar, and Ayman El-Khashab.