# FlightPath: Obedience vs. Choice in Cooperative Services

Harry C. Li[1], Allen Clement[1], Mirco Marchetti[2], Manos Kapritsos[1], Luke Robison[1], Lorenzo Alvisi[1], and Mike Dahlin[1]
[1]The University of Texas at Austin, [2]University of Modena and Reggio Emilia,
{harry, aclement, mirco, manos, lrobison, lorenzo, dahlin}@cs.utexas.edu

**Abstract:** We present FlightPath, a novel peer-to-peer streaming application that provides a highly reliable data stream to a dynamic set of peers. We demonstrate that FlightPath reduces jitter compared to previous works by several orders of magnitude. Furthermore, FlightPath uses a number of run-time adaptations to maintain low jitter despite 10% of the population behaving maliciously and the remaining peers acting selfishly. At the core of FlightPath's success are *approximate equilibria*. These equilibria allow us to design incentives to limit selfish behavior rigorously, yet they provide sufficient flexibility to build practical systems. We show how to use an ε-Nash equilibrium, instead of a strict Nash, to engineer a live streaming system that uses bandwidth efficiently, absorbs flash crowds, adapts to sudden peer departures, handles churn, and tolerates malicious activity.

## 1 Introduction

We develop a novel approach to designing cooperative services. In a cooperative service, peers controlled by different entities work together to achieve a common goal, such as sharing files [23, 12] or streaming media [20, 25, 27]. Such a decentralized approach has several advantages over a traditional client-server one because peer-to-peer (p2p) systems can be highly robust, scalable, and adaptive. However, a p2p system may not see these benefits if it does not tolerate Byzantine peers that may disrupt the service or selfish peers that may use the service without contributing their fair share [3].

We propose *approximate equilibria* [11] as a rigorous and practical way to design cooperative services. Using these equilibria, we can design flexible mechanisms to tolerate Byzantine peers. More importantly, approximate equilibria guide how we design systems to incentivize selfish (or *rational*) peers to obey the protocols.

Recent deployed systems [12, 23] and research prototypes [1, 3, 25, 27, 33] build incentives into their protocols because they recognize the need to curb rational deviations. These works fall into two broad categories.

The first set includes works that use incentives informally to argue that rational peers will obey a protocol. This approach provides system designers the freedom to engineer efficient and practical solutions. BitTorrent [12] and KaZaA [23] are examples of this approach. However, informally arguing correctness leaves systems open to subtle exploits in adversarial environments. For example, Sirivianos et al. [38] demonstrate BitTorrent's vulnerability to large view attacks, and KaZaA Lite [24] shows how to manipulate KaZaA's participation score.

The second set of works emphasizes rigor by using game theory to design a protocol's incentives and punishments so that obeying the protocol is each rational peer's best strategy. This approach focuses on crafting a system to be a Nash equilibrium [34]. The advantage of this more formal technique is that the resulting system is provably resilient to rational manipulation. The disadvantage is that strict equilibrium solutions restrict the freedom to design practical solutions, yielding systems with several unattractive qualities. For example, BAR-Backup [3], BAR Gossip [27], and Equicast [25] do not allow dynamic membership, require nodes to waste network bandwidth by sending garbage data to balance bandwidth consumption, and provide little flexibility to adapt to changing system conditions.

The existing choices—practical but informal or rigorous but impractical—are discouraging, but approximate equilibria offer an alternative. These equilibria let us give a limited degree of choice to peers, departing from the common technique of eliminating choice to make a cooperative service a strict equilibrium.

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 provide more opportunities for fruitful trades.

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 our experiments with over 500 peers, 98% of peers deliver every packet of an hour long video. 100% of peers miss less 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.

**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 massive peer departures.

Compared to our previous work [27], the above properties represent *both* a qualitative and quantitative improvement. We reduce jitter by several orders of magnitude and decrease the overhead of our protocol by 50% compared to BAR Gossip. Additionally, we allow peers to join and leave the system without disrupting service.

Although approximate equilibria, in principle, provide weaker guarantees than strict ones, existing works achieve strict Nash equilibria by making strong assumptions. In BAR Gossip, for example, we assume that rational participants only pursue short-sighted strategies, ignoring more sophisticated ones that might pay off in the long term. In Equicast, Keidar et al. assume that users are hurt by an infinite amount if they do not receive every 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 the paper as follows. Section 2 defines the live streaming problem and the model in which we are working. Section 3 describes FlightPath's basic trading protocol and discusses how to add flexibility to improve performance significantly and handle churn. We evaluate our prototype in Section 4 which looks at FlightPath without churn, with churn, and under attack. In Section 5, we analyze the incentives a rational peer may have to cheat. Finally, Section 6 highlights related work and Section 7 concludes this paper.

## 2 Problem & Model

We explore approximate equilibria in the context of streaming a live event over the Internet. A *tracker* maintains the current set of peers that subscribe to the live event. A *source* divides time into rounds that are $\Gamma$ seconds long. In each round, the source generates $\sigma$ unique stream packets that expire after $\delta$ rounds. The source multicasts each packet to a small fraction $f$ of peers. All peers 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 than $\sigma$ stream updates in a round, we consider that round *jittered* and our goal is to minimize such rounds. We assume that the source and tracker nodes run as specified and do not fail, although we could relax this assumption using standard techniques for fault-tolerance [9, 15, 39]. Peers, however, may fail.

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. For example, Adar and Huberman point out that over 70 percent of Gnutella users share no files [2]. Thus, we desire to create protocols that continue to function even if all participants are rational and willing to deviate for a large enough 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 bounded by $F_{byz} < 1$. Altruistic peers obey the given protocol but may crash unexpectedly as can rational peers.

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 [16, 43].

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 [6]. We denote a message $m$ signed by peer $i$ as $\langle m \rangle_i$.

Finally, we hold peers accountable for the messages they send. We define a proof of misbehavior (POM) [3] 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 POM, although one could use a POM as a basis for additional sanctions [37].

## 2.1 Equilibrium Model

We analyze and evaluate FlightPath using ε-Nash equilibria [11]. In such an equilibrium, rational players deviate if and only if they expect to benefit by more than a factor of ε. This assumption is reasonable if switching protocols incurs a non-trivial cost such as effort to develop a new protocol, effort 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 or use an alternate protocol. In FlightPath, we assume that protocols that bound the gain from cheating to $\varepsilon \leq \frac{1}{10}$ are sufficient to discourage rational deviations.

FlightPath is the first peer-to-peer system that is based on an approximate equilibrium. Other works [11, 13] have used approximate equilibria only when the strict versions have been computationally hard to calculate. To our knowledge, FlightPath is the first work to explore how these equilibria can be used to trade off resilience to rational manipulation against performance.

**A peer's utility:** We assume that a rational peer benefits from watching or listening to a jitter-free stream, and that that benefit decreases monotonically as jitter increases. We also assume that a peer's cost increases proportionally with the amount of upload bandwidth consumed. Although FlightPath is not tied to any specific utility function that combines these benefits and costs, we provide one here for concreteness: $u = (1-j)\beta - w\kappa$, where $j$ is the average number of jitter events per minute, $w$ is the average bandwidth used in kilobits per second, $\beta$ is the benefit received from a jitter free data stream, and $\kappa$ is the cost of uploading at a rate of 1 kbps. In Section 5, we show how $\beta$ and $\kappa$ affect the $\varepsilon$ we can bound in an ε-Nash equilibrium.

## 3 FlightPath Design

We discuss FlightPath's design in three iterations. In the first, we give an overview of a basic trading protocol that allows peers to exchange updates. This protocol is inspired by our earlier work in BAR Gossip [27], where we structure trades using the principle of delayed gratification. We structure the basic trading protocol 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, overload, or missing updates. 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.

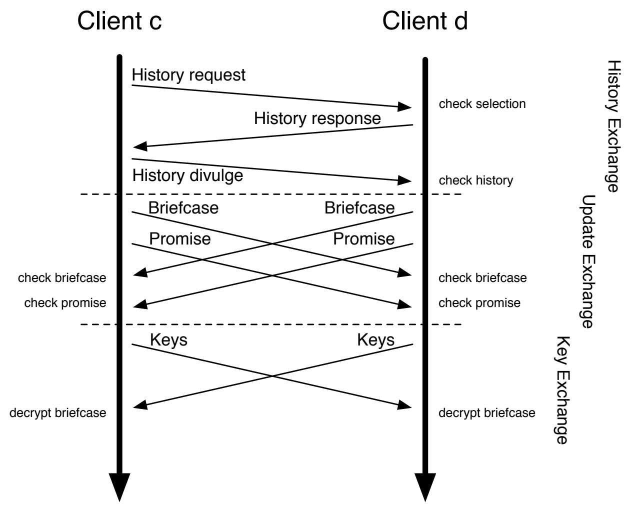

Figure 1: Illustration of a trade in the basic protocol.

Readers familiar with related works on rational peers may be surprised to see that in the last two iterations we do not argue step-by-step about incentives. This difference is a direct consequence of the freedom that approximate equilibria provide to system designers, allowing optimizations that may not *always* be in a rational peer's best interest, but for which cheating has marginal impact on a streaming session's start-to-finish benefits and costs. In Section 5, we demonstrate that FlightPath is a $\frac{1}{10}$-Nash equilibrium under reasonable assumptions.

## 3.1 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. A *linear digest* [20] contains information that allows peers to check the authenticity of received stream updates. Linear digests are signed by the source and contain secure hashes of stream updates. 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 $\sigma$ unique stream updates for a round to a small fraction $f$ of random peers in the system. When the source multicasts stream updates to selected peers at the beginning of every round, the source also sends them the appropriate linear digests.

In each round, peers initiate and accept trades from their neighbors. As in BAR Gossip, a trade consists of four phases: partner selection, history exchange, update exchange, and key exchange. First, a peer selects a partner using a *verifiable pseudo-random algorithm* [27]. Second, 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. Third, partners swap updates by encrypting them and sending the encrypted data in a briefcase message. Immediately afterwards, a peer sends a *promise* stating the identity of each encrypted update in the briefcase. Promises are the only digitally signed message in a trade; peers authenticate other messages using message authentication codes (MACs). Fourth, once a peer receives a briefcase and a matching promise message from its trading partner, that peer sends the decryption keys necessary to unlock the briefcase it sent.

These phases are very similar to exchanges in BAR Gossip and they provide a similar guarantee: 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 [27]. The main difference between a trade in this protocol compared to balanced exchanges in BAR Gossip is the addition of the promise.

We structure promises so that if a briefcase contains garbage data, then the matching promise is a proof of misbehavior (POM). Briefcases and promises provide this property because of how we construct the contents of these messages. A briefcase contains a list of pairs $\langle updateID, bitstring \rangle$, where $bitstring$ is the result of encrypting the matching update with its own secure hash. Note that this bit string is a deterministic function of the update. A promise contains a list of pairs $\langle updateID, hash \rangle$, where $hash$ is the secure hash of the corresponding $bitstring$ in the briefcase. If a briefcase holds garbage data, that lie will be reflected in one of the bit strings, which will then be reflected in the hash of the corresponding promise message and serve as a POM.

## 3.2 Taming Gossip

Gossip protocols are well-known for their robustness [7, 14]. Gossip protocols are especially attractive in a BAR environment because their robustness helps with tolerating Byzantine peers. However, while gossip's pairwise interactions make crafting incentives easier than in a tree-based streaming system, it is reasonable to question whether the very randomness that makes gossip robust may not make it inappropriate for streaming live data in which updates have to be propagated to all of the system by a hard deadline.

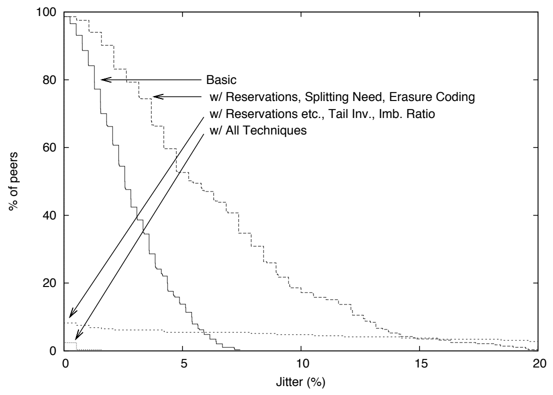

Figure 2: Reverse cumulative distribution of jitter of the basic trading protocol with several improvements.

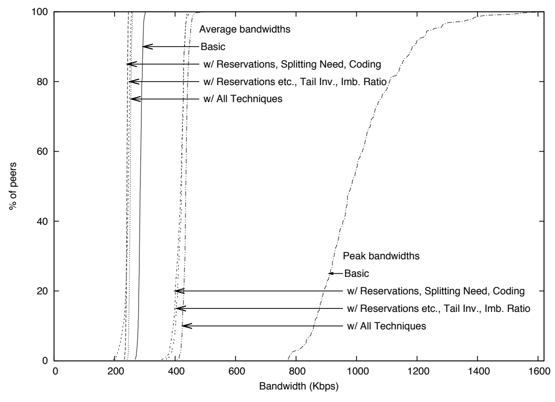

Figure 3: Cumulative distribution of average and peak bandwidths of the basic trading protocol with several improvements.

In this section, we explain how the flexibility of approximate equilibria allow us to tame gossip's randomness by dynamically adapting run-time decisions. For concreteness, we show in Figure 2 how poorly the basic protocol performs when disseminating a 200 Kbps stream to 517 clients. In this experiment, the source generates $\sigma$= 50 unique stream updates per round and sends each one to a random $f$= 5% of the peers. As the figure shows, the first three of the modifications we are about to discuss—reservations, splitting need, and erasure coding—help in capping the peak bandwidth used by the protocol but, by reining in gossip's largesse with bandwidth, make jitter worse. The next three—tail inversion, imbalance ratio, and a trouble detector—reduce

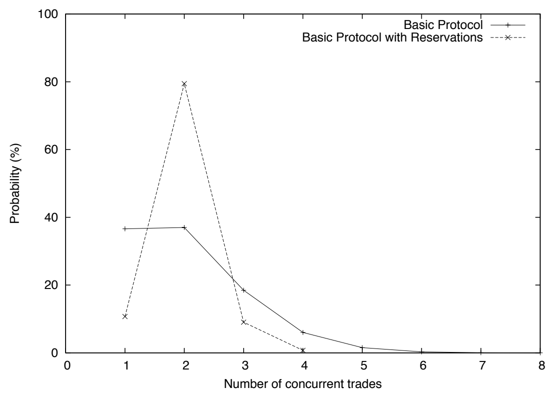

Figure 4: Effect of reservations on the probability of concurrent trades.

jitter by several orders of magnitude.

**Reservations:** One of the problems of using random 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 2, the actual number varies widely, sometimes going past 8 as Figure 4 illustrates. 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 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 receive several duplicate updates when participating in many concurrent trades, wasting bandwidth that could be better spent on other updates a peer is missing. We also see this effect in Figure 3 where the basic protocol's average bandwidth is almost 1.5 times the stream's bandwidth. The following improvements help FlightPath control the inherent variance in gossip protocols.

Rather than accept however many incoming connections occur in the basic protocol, we distribute the number of concurrent trades more evenly by providing a limited amount of flexibility in partner selection. The idea is simple. A peer $c$ reserves a trade with a partner $d$ before the round $r$ in which that trade should happen. If $d$ has already accepted a reservation for $r$, then $c$ looks for a different partner. Figure 4 shows that reservations significantly impact reduce the probability of a peer committing to more than 2 concurrent trades in a round. At the same time, it also reduces the probability that a peer is only involved in the trade it initiates. The technical difficulty 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 three requirements: peers need to select partners in a sufficiently random way to retain gossip's robustness, each peer needs enough choice to avoid overloaded or Byzantine peers, and these sets should be relatively stable as the population of peers change over time. (We defer discussing dynamic membership to Section 3.3, but its demands constrain the peer selection algorithm we describe here.)

We force each peer to communicate with at least $\lfloor \log n \rfloor$ distinct neighbors by partitioning the membership list of $n$ peers into $\lfloor \log n \rfloor$ bins, and we require a peer to choose a partner from a verifiable pseudorandomly chosen bin each round. In particular, in round $r$, peer $c$ uses $\langle r \rangle_c$ (the round number encrypted by the peer's private key) to select a bin via a deterministic function; note that any peer can verify any other peer's calculation using the peer's public key.

Within a bin, we further restrict the nodes with whom a peer can communicate by giving each peer a *view* of membership. We make views stable by basing them on peers' ids. The view construction has two parts. We define $c$'s view to be all peers $e$ such that the hash of $c$'s member id with $e$'s member id is less than some $p$. The expected view size of each peer should be large enough to ensure that *with high probability* every peer has at least one non-Byzantine peer in its view for every bin. The tracker ensures this condition by setting $p$ according to the inequality below. Figure 5 gives an intuition for how this inequality affects a peer's choices as the system scales up.

$$p \geq \frac{1 - \sqrt[\frac{n}{\lceil \log n \rceil}]{1 - \sqrt[\lceil \log n \rceil]{1 - \frac{1}{n}}}}{1 - P_{byz}} \qquad (1)$$

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

A peer $d$ generally accepts a reservation only if $d$ has not already committed to another reservation for the same round. If $d$ has, then $d$ rejects the reservation and $c$ attempts to reserve a trading opportunity with a different peer. There is an exception that allows $d$ to accept more than one reservation: if $c$ sets a *plead* flag in its reservation, indicating that $c$ has few options left, then $d$ accepts the reservation unless $d$ has already committed to 4 trades in round $r$. In our protocol, a peer only pleads

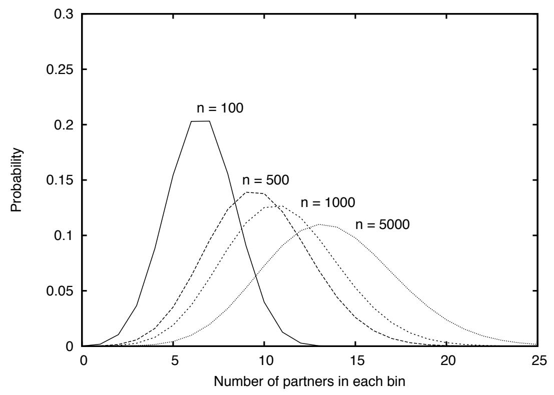

Figure 5: Distribution of view sizes in each bin for different membership list sizes. Graphs are calculated with $F_{byz} = 20\%$.

for a reservation if it has fewer than two remaining potential partners for round *r*.

**Splitting need:** Reservations are effective in ensuring that peers are never involved in more than 4 concurrent trades. However, reserving trades only addresses part of the problem. 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 duplicate data.

For example, consider a peer $c$ involved in concurrent trades with peers $d_0, d_1, d_2$, and $d_3$. Peer $c$ is missing 8 updates for a given round. The basic protocol may overwhelm $c$ and waste bandwidth by having peers $d_0$–$d_3$ each send those 8 updates to $c$. Something more intelligent is for $c$'s need to be split evenly across its trading partners, limiting each trading partner to send at most 2 updates. Note, however, that while this scheme may be less wasteful than before, $c$ now risks not receiving the 8 updates it needs since it is unlikely that its trading partners each independently select disjoint sets of 2 updates to exchange.

It appears as though $c$ has to walk a fine line between being conservative and receiving many duplicate updates to avoid jitter or taking a risk to save resources. We sidestep this trade-off by using erasure coding [4, 28].

**Erasure codes:** The source codes all of the stream data in a given round into $m > \sigma$ stream updates such that any $\sigma$ blocks are necessary and sufficient to reconstruct the original streamed data for that round. Peers attempt to gather any $\sigma$ blocks for a round, which is necessary and sufficient to reconstruct the original for that round. Erasure coding makes it much less likely that concurrent trades involve the same block.

In our experiments, we erasure code $\sigma$ stream updates into $m = 2\sigma$ blocks and modify the source to send each one to $\frac{f}{2}$ of the peers. In Figures 2 and 3, the source generates $2\sigma = 100$ blocks and sends each one to a random 2.5% of the peers.

As shown, the techniques proposed so far make jitter *worse*. However, our modifications significantly reduce the protocol's peak bandwidth. We now describe three ways that together reduce jitter to nearly 0 while using the improvements we have introduced so far to keep our protocol from overwhelming any peer.

**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 desires to avoid jitter.

The drawback in preferring to trade for older data is that the received updates may not be useful in future exchanges because many peers already possess enough of those older updates. Indeed, we have confirmed that using an oldest-first bias in our prototype performs very poorly. We therefore provide the flexibility for a peer 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 two most recent rounds first and then updates in oldest-first order. Out of several orderings that we tried, our experiments indicate that this one has the largest impact on reducing jitter.

**Imbalance ratio:** In order to provide incentives for peers to contribute their fair share, the basic protocol balances trades so that a peer receives no more than it contributes in any round. Such balanced trading can cause an unlucky peer to fall behind.

FlightPath uses an *imbalance ratio* $\alpha$ 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 $\alpha$ 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 $\alpha$ is set to 1, the trading protocol behaves like a traditional unbalanced gossip protocol, vulnerable to free-riding behavior as we show in previous work [27]. When $\alpha$ 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 find that setting $\alpha$ to 10% allows sufficient flexibility for low jitter.

**Trouble Detector:** Our final improvement takes advantage of the flexibility in partners that our reservation mechanism offers. 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 can initiate proactively 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 number of bytes uploaded to trade for each unique update.

We implement a simple detection module that informs a peer whether reserving 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 $\sigma$ 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.

Figure 2 demonstrates the effectiveness of tail inversion, the imbalance ratio, and the trouble detector.

## 3.3 Flexibility for Churn

We now explain how to augment the protocol to handle churn. 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 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 [22].

When a peer attempts to join a session, it expects to begin reliably watching or listening to 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 flash crowd where ratio of new peers to old ones is high presents 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 1 peer joining, many peers join, say 50, 200, or 400. 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 the tracker's list takes effect.

**Epochs:** A FlightPath tracker periodically assigns new membership lists 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 $\Delta$ rounds, the second epoch contains the next $\Delta$ 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. 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 chunks and places each chunk 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.

**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 uses an online algorithm that allows new peers to begin trading immediately without overwhelming the existing peers in the system. Intuitively, our algorithm organizes all peers into *tubs* such that the first tub contains the oldest peers and subsequent tubs contain younger and younger peers. A peer selects partners from its own tub and also from any tub older than itself. However, the probability that a peer from tub $t$ selects from a tub $t' < t$ decreases exponentially with $t - t'$. This arrangement ensures that the load on a peer from all newer tubs is bound by a constant regardless of how many peers join. Figure 6 illustrates our algorithm.

For clarity, we describe our online algorithm assuming all peers have knowledge of a *global list* consisting of the current membership list and all joins since the beginning of the previous epoch. Later, we show that this knowledge is unnecessary. The first $x$ indices in this global list correspond to $n$ indices of the membership list

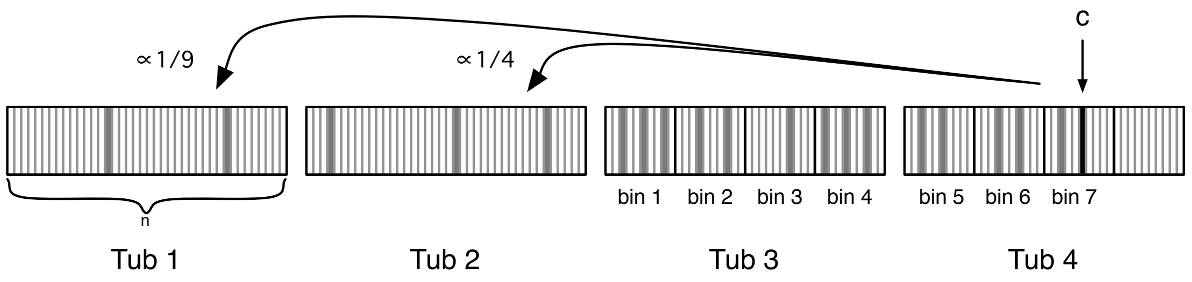

Figure 6: Illustration of the tub protocol from peer *c*'s perspective. Shaded entries represent peers that *c* can contact for a trade when appropriate. Note that *c* only uses bins for its own tub and the immediately preceding one.

for the current epoch. We sort the rest of the global list according to the order in which peers joined. We then divide the global list into *tubs* where the first tub corresponds to the first $x$ indices of the global list, the second tub to be the next $x$ indices, and so forth.

A peer $c$'s membership view depends on its position in the global list. If $c$ is in the first tub, its view is the same as in the static case. If $c$ is in a tub $t > 1$, $c$'s view obeys three constraints:

1. Peer $d$ is in $c$'s view only if $d$ precedes $c$ in the list.
2. If $d$ is in tub $t$ or $t-1$, then $d$ is in $c's$ view iff the hash of concatenating $c$'s member id with $d$'s member id is less than $p$.
3. If $d$ is in a tub $t' < t-1$, then $d$ is in $c$'s view iff the hash of concatenating $c$'s member and $d$'s member id is less than a parameter $p'$.

The tracker sets $p'$ for each epoch as

$$p' \geq \frac{1 - \sqrt[n]{\frac{1}{n}}}{1 - F_{byz}}$$

which guarantees with high probability that new peers have at least one non-Byzantine peer in every tub.

A new peer $c$ in tub $t_c > 1$ selects a trading partner for round $r$ using two verifiable pseudo-random numbers, $rand_1$ and $rand_2$. First, $c$ uses $rand_1$ to select a tub, exponentially weighting the selection towards its own tub. If $c$ selects a tub $t < t_c - 1$, then $c$ can trade with any peer in tub $t$ that is also in $c$'s view. If $c$ selects either its own tub or the one immediately preceding its tub, then $c$ uses $rand_2$ to make the final selection. $c$ maps $rand_2$ to a bin starting from the bin in tub $t-1$ and ending with $c$'s own bin. From the selected bin, $c$ can trade with any peer in its view.

If every peer always 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 $c$ joins the system, $c$ obtains such a list directly from the tracker.

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

## 4 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 being Byzantine

### 4.1 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.

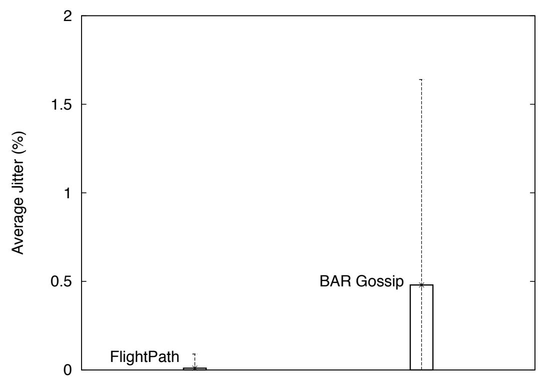

Figure 7: Average jitter in FlightPath and BAR Gossip peers. ($n$ = 517)

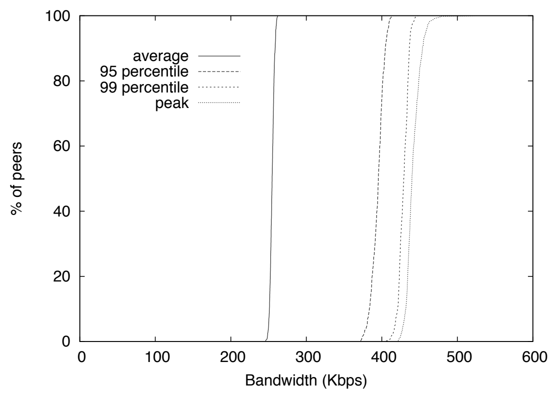

Figure 8: Distributions of peers' average, 95 percentile, 99 percentile, and peak bandwidths. ($n$ = 517)

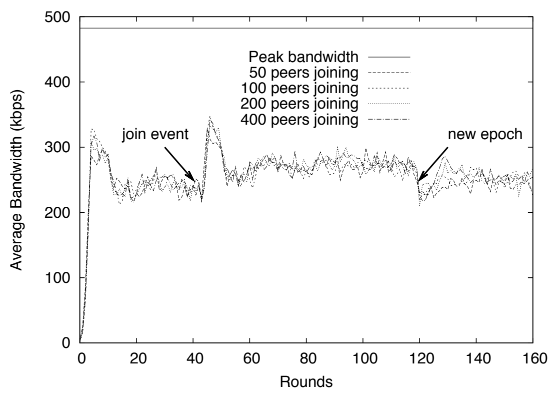

Figure 9: Bandwidth of peers already in the system with different sized flash crowds. ($n$ = 50)

In our experiments, each round lasts 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. The source sends each stream update to a random 2.5% of peers. Each stream update is 1072 bytes long, while linear digests are 1153 bytes long.

We implement FlightPath in Python using MD5 for secure hashes and RSA-FDH with 512 bit keys for digital signatures. Peers exchange public certificates and agree on secret keys for MACs a few seconds before reserving trades with one another. Peers also set the budget for how many updates they are willing to upload in a round to $\mu = 100$.

**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 7 shows that the average jitter of FlightPath is orders of magnitude 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. Figure 8 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 3.2, the combination of reservations, splitting a peers need and erasure coding is effective in capping peak bandwidth.

**Joins:** We now evaluate how well FlightPath handles joins into the system. In particular, we stress the tub algorithm, described in Section 3.3, to handle large populations of peers who seek to join a streaming session all at once. In this experiment, we start a session with 50 peers. In round 40, we have some number of additional peers simultaneously attempt to join the system. As Figure 9 illustrates, the average bandwidth of the original peers noticeably spikes immediately after round 40 and settles to a higher level than before. In round 120, when new peers are integrated into the membership list, average bandwidth of the original 50 drops back to its previous levels. We recognize that it may be difficult for the reader to distinguish lines on this graph, which is the point. 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 10 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

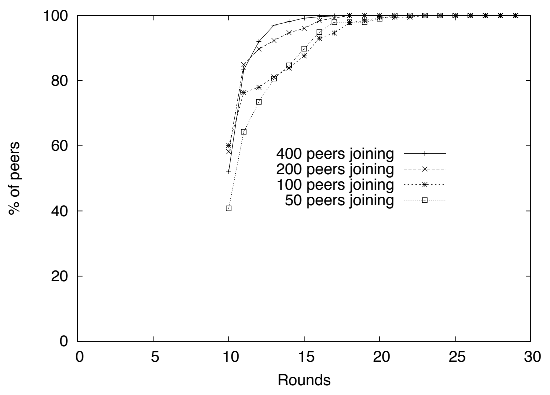

Figure 10: CDF of join delays for different size joining crowds. ($n = 50$)

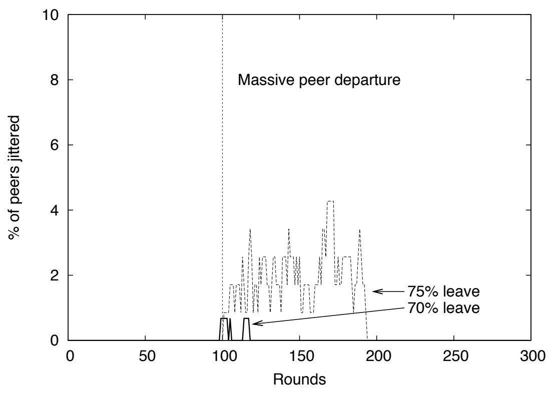

Figure 11: Jitter during massive departure. ($n = 517$)

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, performance improves. This effect can be explained by our tub algorithm. Those peers in the last tub are contacted the least. In the experiment in which only 50 peers join, they are in the last tub and take the longest to join. The last tub in the experiment with 400 peers joining has a similar problem, but the difficulty of peers in the last tub is masked by the success of the previous 350 peers.

**Departures:** Figure 11 shows FlightPath's resilience to large fractions of a peer population suddenly departing. In this experiment, 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

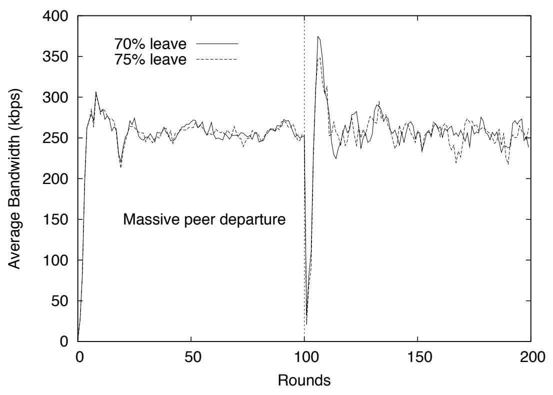

Figure 12: Average bandwidth after a massive departure.($n = 517$)

70% and 75% in which FlightPath cannot tolerate any more departures.

FlightPath's resilience to such large number of peers leaving the system is a consequence of a few traits. First, peers use TCP connections to conduct trades, so peers discover very quickly whether potential partners have left or not. 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. In Figure 12, we can see the effect of the trouble predictor where average bandwidth of remaining peers drops dramatically after the leave event, but then spikes sharply to make up for missed trading opportunities.

**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. Because peers who remain in the system a short time are proportionally more affected by their start-up transients, our presentation segregates peers by the amount of time they remain in the system. Figure 13 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. Interestingly, peers who stay in the system for at least 640 seconds experience very little jitter even when 37% of peers churn every minute. Further experiments (not included) show that there is a non-negligible probability of being jittered during the first two minutes after joining a streaming session. Afterwards, the chance of being jittered falls to nearly zero.

Figure 14 shows that churn does manifest as increasing join delays for new peers. We see that the time needed to join a session is unacceptable under high

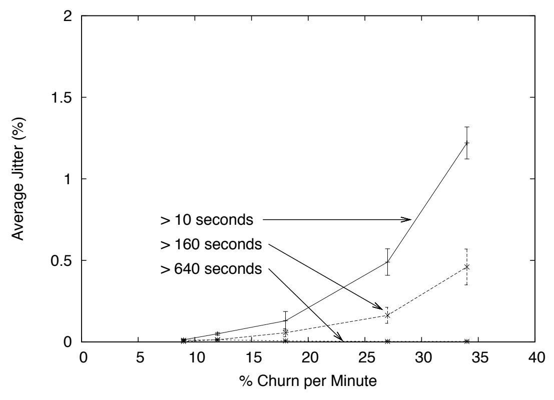

Figure 13: Average jitter as churn increases.($n$ = 443)

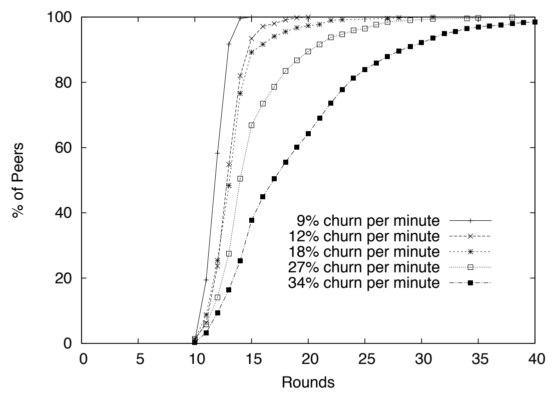

Figure 14: Join delay under churn.($n$ = 443)

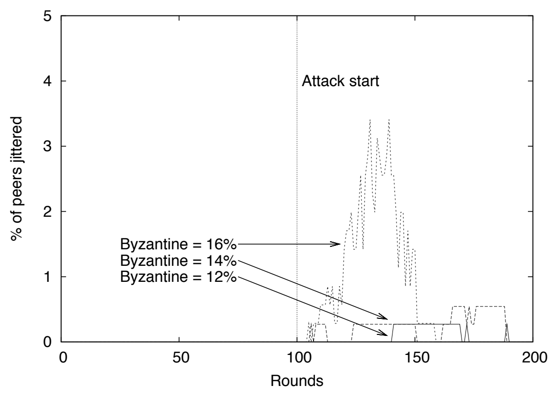

Figure 15: Jitter with malicious peers. ($n$ = 443)

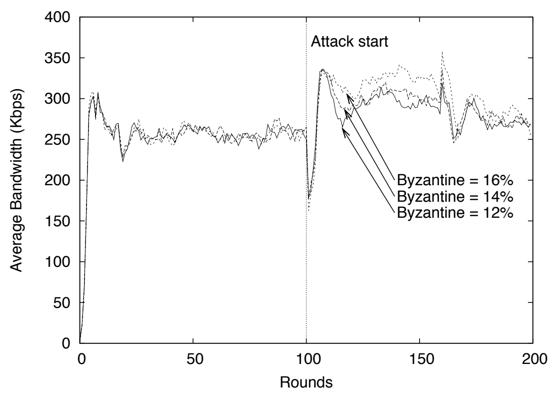

Figure 16: Bandwidth with malicious peers. ($n$ = 443)

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 the final experiment, we evaluate FlightPath's ability to deliver a stream reliably under attack. Byzantine peers may be malicious, possess unknown utility functions, or malfunction. Note that Byzantine peers cannot make a non-Byzantine peer deliver an inauthentic update. Byzantine peers can, however, harm the system by degrading performance. In particular, we present a strategy that malicious peers could employ to lower the utility of all non-Byzantine peers.

Malicious peers act normally for the first 100 rounds of the protocol. However, 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. These peers participate in the history exchange phase of a trade but in no subsequent phase. In a history exchange, a Byzantine 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 partners bandwidth to the exchange. Ultimately, however, non-Byzantine peers find trades with Byzantine ones useless. This strategy was able to induce the most amount of jitter out of several malicious attacks we tried.

Figure 15 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 16, 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.

## 5 Equilibria Analysis

In contrast to previous protocols that use equilibria to dissuade deviation by rational participants, FlightPath does not artificially ensure that every choice a peer makes is the best possible one. Indeed, it is easy to imagine circumstances under which, say, a peer might benefit by setting the *plead* flag early to increase the likelihood that a selected peer will accept its invitation. Instead, FlightPath ensures an $\varepsilon$-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 if it decreases its cost (by uploading less) or if it increases its benefit (by reducing jitter). We construct trades in FlightPath so that a peer has to pay at least $\lceil \frac{1}{1+\alpha} \rceil$ of the cost of uploading $x$ updates in order to receive $x$ updates, where $\alpha$ is the imbalance ratio. Since we also craft the FlightPath protocol so that it provides very low jitter ($\leq 0.01\%$) in a wide range of environments, a peer has very little ability to increase its benefit significantly.

We now develop this argument more formally to bound how close FlightPath's prescribed strategy is to an optimal one. 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.

We begin by revisiting the utility function $u = (1 - j)\beta - w\kappa$. Recall that $j$ is the average number of jitter events per minute, $\beta$ is the benefit from watching a jitter-free stream, $w$ is the average upload bandwidth used in Kbps, and $\kappa$ is the cost per Kbps. If we let the expected utility of an optimal cheating strategy be $u_o = (1 - j_o)\beta - w_o\kappa$ and the expected utility of obeying the protocol be $u_e = (1 - j_e)\beta - w_e\kappa$, then we can express $\varepsilon$ as follows:

$$\varepsilon = \frac{u_o - u_e}{u_e} = \frac{(j_e - j_o)\beta - (w_o - w_e)\kappa}{(1 - j_e)\beta - w_e\kappa} \tag{2}$$

We can simplify the above equation with the following assumptions: *i)* the benefit of running FlightPath exceeds the cost, *ii)* the optimal cheating strategy receives no jitter, and *iii)* the optimal cheating strategy uses a fraction $b < 1$ of the bandwidth of running the protocol. These assumptions let us express $\varepsilon$ as a function of the benefit-to-cost ratio $c$, the expected number of jitter events per minute $j_e$, and the proportional savings in cost $1 - b$.

$$\varepsilon = \frac{\frac{c j_e}{1-j_e} + (1-b)}{c-1} \tag{3}$$

| Parameter | Description |
|---|---|
| $\delta$ | num rounds a peer remains in system |
| $\sigma$ | num stream updates per round needed |
| $m$ | num stream updates per round |
| $f$ | fraction of updates received from source |
| $\mu$ | max num of updates sent in a trade |
| $\alpha$ | imbalance ratio |

Table 1: Summary of the analysis parameters.

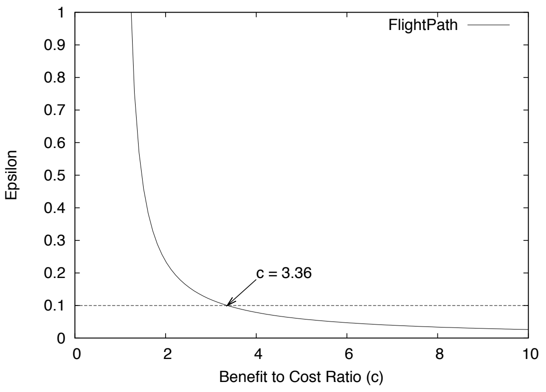

Figure 17: $\varepsilon$ as a function of the benefit to cost ratio.

As the jitter expected is an empirical phenomenon, we use our evaluation to determine $j_e = 0.01\%$. We then establish a lower bound on $b$ using parameters specific to our system, listed in Table 1.

In the steady state, a peer $p$ following a hypothetical optimal strategy participates on average in at least one trade every $t = \lfloor \frac{\mu}{\sigma - m(1-f)} \rfloor$ rounds. Furthermore, the average number of updates that it needs in each trade is $needed = t(\sigma - m(1-f))$. Assuming that $p$ is lucky or clever enough to upload no more updates than it has to in all trades, then $p$ still uploads at least $min\_up = \lceil \frac{needed}{1+\alpha} \rceil$ updates on average every two rounds.

Let $\gamma$ be the fixed cost in kilobits of a trade and let $\rho$ be the increase in cost of a trade for each update $p$ uploads. Then the average cost that $p$ has to pay for each trade is $\gamma + min\_up \times \rho$. Given the message encodings in our prototype, the fixed cost of a trade is 305 bytes and the increase for each uploaded update is 1104 bytes. These values correspond to $\gamma = 2.44$ and $\rho = 8.832$. Our goal is to ensure that the net utility of the optimal strategy is not significantly more than for FlightPath's strategy. For $\varepsilon = \frac{1}{10}$, solving for $c$ in Equation 3 indicates that FlightPath is a $\frac{1}{10}$-Nash equilibrium as long as the user values the stream at least 3.36 times as much as the bits uploaded to participate in the system. Figure 17 illustrates the $\varepsilon$

value FlightPath provides for each benefit-to-cost ratio.

## 6 Related Work

This work builds on a broad set of approaches for content dissemination and Byzantine or rational-tolerant protocols.

Clearly, BAR Gossip [27] is the work most closely related to FlightPath. We explain how it is similar and different from FlightPath throughout this paper, and in particular Section 3.

Several tree-based overlays [10, 21] are devised to disseminate streaming data is another tree-based overlay multicast protocol. Ngan et al. [35] suggest that periodically restructuring Splitstream [10] trees can guard against free-riders by periodically changing the parent-child relationships among peers, a communication pattern that begins to resemble gossip. Chunkyspread [41] uses a multi-tree based approach to multicast. Chunkyspread builds random trees using low overhead bloom filters and allows peers to make local decisions to tune the graph for better performance.

In Araneola [32], Melamed and Keidar construct random overlay graphs to multicast data. They show that Araneola's overlay structure achieves mathematical properties important for low-latency, load balancing, and resilience to benign failures.

Demers et al. introduced gossip protocols to manage consistency in Xerox's Clearinghouse servers [14]. Years later, Birman et al. [7] used gossip to build a probabilistic multicast—a middle ground between existing reliable multicast and best effort multicast protocols. Since then, many have explored ways to improve gossip's throughput and robustness [8, 17, 18, 19, 26, 42].

None of the above works consider Byzantine peers who can harm the system by spreading false messages. One can guard against such attacks by using techniques that avoid digital signatures [29, 30, 31], but signatures can dramatically simplify protocols and are used in many practical gossip implementations [8, 20, 27, 40].

Haridasan and van Renesse [20] build a Byzantine fault-tolerant live streaming system over the Fireflies system. Their system, SecureStream, introduces *linear digests* to efficiently authenticate stream packets. As in CoolStreaming [44] and Chainsaw [36], SecureStream also uses a pull-based gossip protocol to reduce the number of redundant sends.

Badishi et al. [5] show in DRUM how gossip protocols can resist Denial-of-Service (DoS) attacks by resource bounding public ports and port hopping. We could integrate DRUM's techniques into FlightPath.

To our knowledge, Equicast [25] is the first work to address formally rational behavior in multicast protocols. Equicast organizes peers into a random graph over which it disseminates content. The authors prove Equicast is an equilibrium, but assume that rational peers lack the expertise to modify the protocol beyond tuning a single parameter $H$ that represents the cooperation level.

BAR-Backup [3] is a p2p backup system for Byzantine and rational peers. Peers implement a replicated state machine that moderates interactions between peers to ensure that peers behave appropriately.

## 7 Conclusion

We present approximate equilibria as a new way to design cooperative services. We show that approximate equilibria allow us to provably limit how much selfish participants can gain by deviating from a protocol. At the same time, these equilibria provide enough freedom to engineer practical solutions that are flexible enough to handle many adverse situations, such as churn and Byzantine peers.

We use $\varepsilon$-Nash equilibria, an example of an approximate equilibrium, to design FlightPath, a novel p2p live streaming system. FlightPath improves on the existing state-of-the-art both qualitatively and quantitatively, reducing jitter by several orders of magnitude, using bandwidth efficiently, handling churn, and adapting to attacks. More broadly, FlightPath demonstrates that we do not have to sacrifice rigor to engineer Byzantine and rational-tolerant systems that perform well and operate efficiently.

## References

[1] I. Abraham, D. Dolev, R. Gonen, and J. Halpern. Distributed computing meets game theory: robust mechanisms for rational secret sharing and multiparty computation. In *Proc. 25th PODC*, pages 53–62, July 2006.

[2] E. Adar and B. A. Huberman. Free riding on Gnutella. *First Monday*, 5(10):2–13, Oct. 2000.

[3] A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J.-P. Martin, and C. Porth. BAR fault tolerance for cooperative services. In *Proc. 20th SOSP*, pages 45–58, Oct. 2005.

[4] N. Alon, J. Edmonds, and M. Luby. Linear time erasure codes with nearly optimal recovery. In *FOCS '95*, page 512, Washington, DC, USA, 1995. IEEE Computer Society.

[5] G. Badishi, I. Keidar, and A. Sasson. Exposing and eliminating vulnerabilities to denial of service attacks in secure gossip-based multicast. In *Proc. DSN-2004*, page 223, Washington, DC, USA, 2004. IEEE Computer Society.

[6] M. Bellare and P. Rogaway. Random oracles are practical: a paradigm for designing efficient protocols. In *Proc. 1st CCC*, pages 62–73, New York, NY, USA, 1993. ACM Press.

[7] K. P. Birman, M. Hayden, O. Oskasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM TOCS*, 17(2):41–88, May 1999.

[8] K. P. Birman, R. van Renesse, and W. Vogels. Spinglass: Secure and scalable communications tools for mission-critical computing. In *DARPA DISCEX-2001*, 2001.

[9] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. *ACM TOCS*, 14(1):80–107, 1996.

[10] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: high-bandwidth multicast in cooperative environments. In *Proc. 19th SOSP*, pages 298–313. ACM Press, 2003.

[11] S. Chien and A. Sinclair. Convergence to approximate nash equilibria in congestion games. In *SODA '07*, pages 169–178, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics.

[12] B. Cohen. Incentives build robustness in BitTorrent. In *P2PECON '03*, June 2003.

[13] C. Daskalakis, A. Mehta, and C. Papadimitriou. A note on approximate nash equilibria. In *WINE '06*, 2006.

[14] 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 *Proc. 11th SOSP*, Aug. 1987.

[15] D. Dolev and D. Malki. The transis approach to high availability cluster communication. *Commun. ACM*, 39(4):64–70, 1996.

[16] J. R. Douceur. The Sybil attack. In *Proc. 1st IPTPS*, pages 251–260. Springer-Verlag, 2002.

[17] P. Eugster, S. Handurukande, R. Guerraoui, A. Kermarrec, and P. Kouznetsov. Lightweight probabilistic broadcast. In *DSN '01*, pages 254–269, July 2001.

[18] 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.

[19] I. Gupta, A.-M. Kermarrec, and A. J. Ganesh. Efficient and adaptive epidemic-style protocols for reliable and scalable multicast. *IEEE TPDS*, 17(7):593–605, 2006.

[20] M. Haridasan and R. van Renesse. Defense against intrusion in a live streaming multicast system. In *Proceedings of P2P '06*, pages 185–192, Washington, DC, USA, 2006. IEEE Computer Society.

[21] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. James W. O'Toole. Overcast: reliable multicasting with on overlay network. In *Proceedings of OSDI '00*, pages 14–14, Berkeley, CA, USA, 2000. USENIX Association.

[22] H. Johansen, A. Allavena, and R. van Renesse. Fireflies: scalable support for intrusion-tolerant network overlays. In *EuroSys '06*, pages 3–13, New York, NY, USA, 2006. ACM Press.

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

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

[25] I. Keidar, R. Melamed, and A. Orda. Equicast: Scalable multicast with selfish users. In *PODC '06*, 2006.

[26] J. Leitao, J. Pereira, and L. Rodrigues. Hyparview: A membership protocol for reliable gossip-based broadcast. In *DSN '07*, pages 419–429, Washington, DC, USA, 2007. IEEE Computer Society.

[27] H. C. Li, A. Clement, E. Wong, J. Napper, I. Roy, L. Alvisi, and M. Dahlin. Bar Gossip. In *Proceedings of OSDI '06*, pages 191–204, Nov. 2006.

[28] M. G. Luby, M. Mitzenmacher, M. A. Shokrollahi, D. A. Spielman, and V. Stemann. Practical loss-resilient codes. In *STOC '97*, pages 150–159. ACM Press, 1997.

[29] D. Malkhi, Y. Mansour, and M. K. Reiter. Diffusion without false rumors: on propagating updates in a byzantine environment. *TCS*, 299(1-3):289–306, 2003.

[30] D. Malkhi, E. Pavlov, and Y. Sella. Optimal unconditional information diffusion. In *Proc. 15th DISC*, pages 63–77, London, UK, 2001. Springer-Verlag.

[31] D. Malkhi, M. Reiter, O. Rodeh, and Y. Sella. Efficient update diffusion in Byzantine environments. In *Proc. 20th SRDS*, 2001.

[32] R. Melamed and I. Keidar. Araneola: A scalable reliable multicast system for dynamic environments. In *Proc. of NCA '04*, pages 5–14, Washington, DC, USA, 2004. IEEE Computer Society.

[33] T. Moscibroda, S. Schmid, and R. Wattenhofer. When selfish meets evil: Byzantine players in a virus inoculation game. In *Proc. 25th PODC*, pages 35–44, July 2006.

[34] J. Nash. Non-cooperative games. *The Annals of Mathematics*, 54:286–295, Sept 1951.

[35] T.-W. J. Ngan, A. Nandi, A. Singh, D. Wallach, and P. Druschel. On designing incentives-compatible peer-to-peer systems.

[36] V. Pai, K. Kumar, K. Tamilmani, V. Sambamurthy, and A. Mohr. Chainsaw: Eliminating trees from overlay multicast. In *IPTPS '05*, February 2005.

[37] H. E. Ramadan. Abort, retry, litigate: dependable systems and contract law. In *HOTDEP'06*, pages 13–13, Berkeley, CA, USA, 2006. USENIX Association.

[38] M. Sirivianos, J. H. Park, R. Chen, and X. Yang. Free-riding in bittorrent networks with the large view exploit. In *IPTPS '07*, February 2007.

[39] R. van Renesse, K. P. Birman, and S. Maffeis. Horus: A flexible group communication system. *Comm. ACM*, 39(4):76–83, 1996.

[40] R. van Renesse, H. Johansen, and A. Allavena. Fireflies: Scalable support for intrusion-tolerant overlay networks. In *EuroSys '06*, 2006.

[41] J. Venkataraman, P. Francis, and J. Calandrino. Chunkyspread: Multi-tree unstructured peer-to-peer multicast. In *IPTPS '06*, February 2006.

[42] W. Vogels, R. van Renesse, and K. Birman. The power of epidemics: robust communication for large-scale distributed systems. *SIGCOMM Comp. Comm. Rev.*, 33(1):131–135, 2003.

[43] H. Yu, M. Kaminsky, P. B. Gibbons, and A. Flaxman. Sybilguard: Defending against sybil attacks via social networks. In *ACM SIGCOMM '06*, Sept.

[44] X. Zhang, J. Liu, B. Li, and T. P. Yum. CoolStreaming/DONet: A data-driven overlay network for live media streaming. In *IEEE INFOCOM*, Mar. 2005.