LiveSwarms: Adapting BitTorrent for end host multicast

Michael Piatek, Colin Dixon, Arvind Krishnamurthy, and Thomas Anderson
University of Washington
Technical Report—TR 2006-11-01

Abstract

The lack of universal multicast support in core internet routers has motivated the development of so-called end host multicast systems. These rely on participating hosts to forward data to other users via a carefully designed overlay mesh. While such strategies have the potential to improve scalability and lower bandwidth costs, the unreliability and heterogeneity of end hosts in practice have proven to be substantial obstacles to deployment. To overcome these difficulties, we design, implement and evaluate a *live swarm*—an unstructured overlay to disseminate live multicast data. While swarms have proven highly effective for very large file transfers, ours is the first comprehensive evaluation of a practical swarming design for real-time multicast delivery. Our system is based on a small number of enhancements to the popular file transfer system BitTorrent; we show that the result is highly robust to nodes with asymmetric bandwidth, high rates of churn, flash crowds, resource constraints, and selfish users.

1 Introduction

The problem of widespread, scalable distribution of live video and audio data on the internet has received much attention recently, particularly as the popularity of the internet as a media outlet has increased. At present, content producers rely largely on two methods of solving this problem: centralized, well-provisioned servers with high upload capacity or commercially managed content distribution networks such as Akamai [1]. Neither of these completely solves the problem at hand. Centralized deployments scale poorly and are particularly vulnerable to unexpected flash crowd conditions. Commercial content distribution networks are closed systems that are only economically feasible for large clients. The emergence of individual users as content producers, evidenced by the rising popularity of blogs and podcasts, motivates an alternate approach.

In parallel with infrastructure based approaches, the research community has investigated end host techniques for providing scalable live broadcast [16, 11, 29, 14, 17, 25, 21, 10, 6, 31]. These systems rely on users participating in the broadcast to relay data to other users in order to distribute bandwidth load among all participants. In contrast to commercial solutions or traditional IP multicast, these systems provide easy deployment, scalability without infrastructure support, and economic viability for non-commercial users.

Research to date along these lines has mainly focused on building efficient tree-based structures with the broadcast source at the root and viewers intelligently organized in an overlay network. These systems have proven effective in certain circumstances and have yielded valuable insight into the difficulties of delivering scalable live content, but thus far, a complete solution has not emerged. We consider several difficulties encountered when applying tree-based approaches. These also serve to motivate design goals of any comprehensive end host multicast system.

- *Capacity utilization:* An ideal overlay should make productive use of all available upload capacity of peers in the overlay [10]. In a straightforward tree based design, a single parent supports all download requirements of its children. The upload capacity of peers with no children is not used.
- *Resource availability:* Constructing a viable broadcast tree from the pool of participating users is not possible if the download demand exceeds accessible upload capacity. This circumstance may arise due to heterogeneous users with asymmetric download and upload bandwidths. As this property can change suddenly under churn, any practical system requires a strategy for quickly detecting the problem and either regulating resource poor participants or integrating stabilizing resources into the overlay.

- *Robustness to churn:* Node failure and churn can be particularly disruptive, yet end hosts are notably unreliable [3] and measurements indicate channel surfing is as prevalent online as off [28, 26]. When a parent departs, all of its children must be moved and immediately supported by a new parent. Service is disrupted if this transition cannot be made quickly, or if a parent departs unexpectedly. A successfully designed system should support participants without interruption even when churn is high.

- *Simplicity and maintainability:* It is possible to address many of these challenges by adding mechanism to tree-based solutions. For example, SplitStream improves capacity utilization by setting up a mesh of trees and sending divisions of data down separate trees [10]. However, practicality requires that we also strive for design simplicity—complex coordination is unlikely to work in the heterogeneous and rapidly evolving environment of the internet.

Instead of a tree-based mechanism, Pai et al. [23] propose Chainsaw, an unstructured swarm to distribute live multicast data. Swarming has proven effective for bulk file transfer as demonstrated by the popularity of peer-to-peer systems such as BitTorrent [12] and eDonkey [13], yielding good performance and resource utilization [5]. These programs break up large files into many small *blocks* that are independently distributed among peers. Because blocks are generally distributed out-of-order and with different orders for different peers, each client can redistribute blocks as soon as they are received. The differing orders in which blocks are obtained by each peer allows them to distribute blocks of interest to others immediately—rather than only after obtaining the entire file.

Although swarms are robust and simple, it is not obvious that random fetches of blocks from neighbors would be effective for delivering data in real-time, particularly under resource constraints. Pai et al. show through simulation that their custom swarming algorithm can work for live data delivery in a homogeneous, cooperative, resource rich environment with infrequent joins. Our interest in this paper is to investigate whether this approach can be generalized to a more realistic setting, with self-interested users, heterogeneous and asymmetric node characteristics, high churn, flash crowds, and resource limits.

To study this question, we designed and implemented *LiveSwarms*, a real-time multicast delivery system based on a small number of changes to BitTorrent. Using host characteristics derived from measurement of real deployments of live multicast, we show that with one notable exception, the existing mechanisms in BitTorrent work extremely well for live delivery of multicast data. In particular, BitTorrent's tit-for-tat rule for enforcing reciprocity between peers yields excellent delivery bounds in practice while providing an incentive for users to contribute maximum resources to the swarm. The one exception is that as resources are removed from the swarm, it may no longer be possible for it to achieve real-time delivery at the rate the data is being produced. Even here, we show BitTorrent does fairly well, typically limiting data flow to a few resource poor nodes rather than causing all nodes to miss their deadlines. Further, since new nodes need to catch up to the broadcaster's data production point, adding even well-provisioned nodes to a resource constrained swarm can cause an otherwise well functioning overlay to miss delivery deadlines.

To address this issue systematically, we modify the mechanism by which nodes join and leave *LiveSwarms* broadcasts in order to monitor overall resource usage and predict when the system is in about to become resource constrained. We can then use admission control, limits on the aggressiveness of newly joined nodes, and/or standby nodes to provide spare capacity to restore system balance. We show that the resulting system is robust even under extreme conditions of high churn, asymmetry, and resource limits.

The remainder of this paper discusses these issues in more detail. In Section 2, we put our work in the context of related work on end host multicast and content distribution systems. In Section 3 we describe the salient features of BitTorrent as a basis for describing our modifications to the system in Section 4. We describe our evaluation methods and experimental design in Section 5 before presenting our results in Section 6. We discuss the implications and future directions of our work in Section 7 and conclude in Section 8.

2 Related work

The problem of scaling the delivery of live streaming data has been extensively studied. We first trace the evolution of these systems, followed by a discussion of the seemingly separate issue of content distribution systems focused on getting large quantities of data to many people as quickly as possible.

2.1 Live streaming

The notion of an end system approach to overlay multicast was pioneered by Narada, Overcast and Yoid [16, 17, 14]. Motivated by the lack of end-to-end multicast support in the network core, they explored the potential benefits of an overlay strategy. Their key observation is that while overlay multicast lacks the efficiency of in-core multicast, the penalty of overlay routing in terms of throughput and latency is low enough to justify its use.

While there are differences between these systems, they all share the general property that they construct a single, source-rooted, multicast tree. This technique obviously provides more scalability than a centralized server and is conceptually simple, but it may not make efficient use of the capacity of participants. Since an end host can only function as a parent in a tree-based system if it is capable of completely supporting at least one child, hosts with bandwidth asymmetries that allow them to view a broadcast at a particular rate, but not forward at that rate, cannot contribute any resources to the overlay. Additional capacity is lost because each peer can only support an integral number of children, so each peer is likely to waste some excess upload capacity that is not sufficient to support a full child. Finally, hosts which comprise the leaves of the tree contribute nothing, even if they could support children or portions of children.

The issue of capacity utilization was the primary motivation for the development of SplitStream, a *multi-tree* protocol [10]. The key idea behind a multi-tree design is that to make use of the capacity at the leaves of a tree, the broadcaster can split the data stream into k *stripes* of data that traverse k different trees constructed over the same peer set. The trees are constructed so that each host participating in the broadcast is an interior node in as few trees as possible, thereby spreading the dissemination burden across as many nodes as possible. If there is no data redundancy across the various streams, a node failure in any one of the k trees could result in loss of delivery. To increase resilience, SplitStream transmits the stream encoded using either erasure coding or special video codecs which support multiple description coding (MDC). Both allow the whole stream or a useable portion of it to be recovered without receiving all of the data, albeit at a cost in computational and storage complexity. In the case of erasure coding, increased playback delay is also required [9, 8, 22].

In order to function as desired, SplitStream must maintain several invariants. First, each node must be part of at least some specified number of the k trees. Second, each node must not have more children than its capacity constraint. Third, no cycles can be introduced into the trees when repairing them in response to arrivals and departures. Finally, every effort must be made to keep each peer as an internal node in as few trees as possible to prevent a single point of failure disrupting multiple stripes simultaneously. In practice, considerable protocol complexity is required to maintain these invariants under churn.

2.2 Content distribution

Distribution of static content without real-time performance constraints has received much attention in both the research and software development communities. Recently, *swarming* system designs have become popular as a means to provide scalable file dissemination to many users. BitTorrent is the prototypical example of such as system, and its popularity has surged in recent years [12]. Many other swarming designs have been proposed by the research community. We discuss one of these, Bullet [19], and its successor Bullet$'$ [18].

There are several features which are common to all swarming designs. First is the decomposition of data into small blocks that are exchanged among peers. These blocks, emanating from one or more sources, are distributed out of order from one peer to another. Since peers request blocks in a distinct order, it is likely that each one has interesting data to forward from the perspective of other hosts in the swarm. Thus, out of order delivery is essential to swarming, yet live multicast requires in-order delivery in real-time, making the applicability of swarms to real-time delivery an open question. Fairness is often enforced by tit-for-tat downloading, wherein peers preferentially upload to other peers that have recently provided them with data.

Hybrid approaches that combine tree-based content distribution with distributed swarming techniques have also been proposed. Bullet proposes the creation of a *mesh* of connections in addition to a main tree-based structure [19]. This mesh is coupled with distributed algorithms to ensure efficient distribution of relevant data throughout the augmented multicast tree/mesh. Blocks not received quickly via the originally constructed tree are obtained through the mesh. In following work, the Bullet designers evaluated a large space of design choices for content delivery. The result, an enhanced version of Bullet referred to as Bullet$'$ [18] compares favorably to both SplitStream and BitTorrent. Although our work is based on BitTorrent, we believe many of the performance optimizations in Bullet$'$ could be applied to our system as well.

Finally, Pai et al. demonstrate with Chainsaw that swarms can be used for live multicast delivery [23]. Although inspiring, their study was in a limited setting, using simulation with rare join events, homogeneous overprovisioned and fully cooperative clients, and a custom algorithm for pulling data from the broadcast source. Our interest is in developing a practical system we can deploy, and thus we must consider the impact of a wider variety of real-world factors. We start from the swarming system in widest use today, BitTorrent, and add the minimal set of changes necessary to make it work well for live delivery. Before describing these changes, though, we must discuss BitTorrent's algorithms in more detail.

3 BitTorrent

In deciding which swarming design to augment and evaluate, we were guided by our design goals of simplicity and maintainability. As we will see, BitTorrent offers a relatively simple, open protocol that is likely to be maintained by its currently active development community [4, 7]. BitTorrent has performed quite well in practice, with many public torrents supporting several thousand concurrent users. We were also attracted to the likelihood that future improvements to BitTorrent would be easily integrated into our design, and further discuss some of these potential benefits in Sections 6 and 7.

BitTorrent was designed as a swarming distribution tool for large files and generally operates as we described swarming systems previously. Peers in the swarm exchange data blocks and control traffic with their set of directly connected peers. This set of peers is unstructured and random, requiring no special recovery operation when new peers arrive or existing peers depart. The control traffic required for data exchange is minimal: each peer transmits messages indicating the data blocks they currently possess and messages indicating their interest in the blocks of other peers. These control messages are used to maintain state at each peer that is correspondingly minimal: peers maintain a list of hosts that are currently interested in obtaining data from them and a list of peers with which they are actively exchanging data. These lists need not be identical. From the perspective of one host Y in the swarm, a peer that Y is actively transmitting to is referred to as *unchoked*. Peers connected to Y but not receiving data are said to be *choked*. The possible peer states for a given host are summarized in Table 1.

In addition to the interest state information, BitTorrent uses a rate-based tit-for-tat fairness mechanism to determine when to switch peers between the choked and unchoked states. Rate-based tit-for-tat simply means that peer X will be preferentially unchoked at peer Y based on the recent transfer rate of blocks from X to Y. Finally, peers may also be *optimistically unchoked*. In addition to preferentially unchoking peers selfishly, several unchoked slots will be given to peers who have not "earned" such status in a tit-for-tat sense. This is done in a round robin fashion to bootstrap new peers into the tit-for-tat process and allow peers to discover new, potentially better sources of data.

Now that we have specified how peers exchange data and maintain state information, we turn to their block request strategy. BitTorrent peers obtain data only after requesting it from other peers—a *pull*-based system. Based on the block availability information garnered from the control traffic among directly connected peers, each host requests blocks using one of two strategies: random request order or local rarest first. The random strategy, used when the number of local connections is small, makes block requests in a random order subject to availability. Once many connections are active, BitTorrent switches to a policy of requesting those blocks least available among its local peers. As verified in [5], this essentially eliminates the "last block" problem, wherein peers very close to download completion must wait a disproportionately large amount of time to obtain the last block they require because it is not available among local peers.

Finally, in order to initially connect to a swarm, clients download a metadata information file, called a *torrent*, from the content provider, usually via a normal HTTP request. This metadata specifies the name and size of the file to be downloaded through the BitTorrent swarm, as well as SHA-1 fingerprints of each block of the larger file to verify data integrity. The metadata file also specifies the address of an HTTP *tracker* server for the torrent, which is a specialized web server that coordinates the activity of each client participating in the swarm by maintaining a list of current active peers and periodically delivering a random subset of other peers in the system to each swarm participant.

4 *LiveSwarms*

Despite BitTorrent's widespread success at distributing large files, it is not immediately clear that it can be adapted to the problem of broadcasting live data. There are two main obstacles. First, BitTorrent relies upon the whole file existing before distribution. This is necessary in order for clients to know the cryptographic hash values of the blocks and which blocks are valid to request out-of-order. Second, BitTorrent makes no attempt to satisfy the performance constraints associated with real-time delivery.

To address these challenges, we present *LiveSwarms*, an implementation of a modified BitTorrent protocol designed to meet the demands of sending live, streaming data. We enumerate the modifications made, followed by describing the motivation for each of the changes.

4.1 Protocol changes

- *Push from pull:* We modified the BitTorrent client so that, rather than responding to client requests, the broadcasting host *pushes* data to its immediate peers [18]. We did not change the behavior of peer-peer interactions when neither are the special broadcasting node, but as pointed out in [5], the upload capacity of the source is precious, and a pull

	Choked	Unchoked
Interested	X wishes to download data from Y, but is currently prohibited.	X is currently downloading data from Y.
Uninterested	Y has nothing of interest to X, and only control traffic is exchanged.	Only possible if X has just completed download of all available desired data from Y and is about to send an uninterested message.

Table 1: The possible states of peer X from the perspective of peer Y in BitTorrent. The BitTorrent client maintains a large number of peers, of which only a few are both interested and unchoked at any given time.

based strategy tends to result in the source transmitting the same block many times. In our implementation, the broadcaster allocates its entire upload capacity towards distributing the most recently produced block.

- *Block signing:* The seed may optionally cryptographically signs blocks as they are generated, since cryptographic hashes of live data cannot be distributed in the torrent metadata file. This allows clients to verify data integrity despite blocks being forwarded through potentially untrustworthy peers.
- *Resource monitoring:* Peers report their upload capacity to the tracker as part of joining the broadcast. This information is used to determine how much excess upload capacity exists in the system at a given time and, if possible, to introduce extra resources when a swarm becomes resource constrained.
- *Admission control:* The tracker may provide newly joined peers with a certificate allowing them to request blocks from the past only if there are enough resources available to support this behavior from the new peer.

The modifications described so far require protocol changes to BitTorrent in the form of modifications to control messages and peer handling of them. The push-pull change requires that the broadcaster send data messages without being prompted by requests. On the receiving end, peers must be modified to willingly accept data messages they did not request when sent from the broadcaster. Clients must also include an indication of their upload capacity during their initial negotiation with the tracker to facilitate resource monitoring.

4.2 Policy changes

Beyond these required changes to the protocol, we make two changes to the existing policy of clients. First, we specify that the BitTorrent client always uses random block requests and never switches local rarest first. Since each peer is generally consuming data at the broadcast rate, the number of blocks a live swarm is distributing is relatively small, leaving little chance for different local peer sets to stratify in terms of block availability. In experiments performed both with and without local rarest first, we noted no difference in playback performance, and prefer the conceptual simplicity of a single random block selection strategy.

Our second change in policy involves our client buffering and playback strategy. When a client first connects to the swarm, it begins requesting video data from 30 seconds prior to the current playback point, as determined by most recent data available from its immediate peers. Once 15 seconds of contiguous data are available, playback begins. If a block is ever unavailable when it is required for playback, the client stops and waits until 15 seconds of contiguous data is again present in its playback buffer. We refer to such an event as rebuffering, and such behavior is common among commercial video players. Since blocks are often received out-of-order, rebuffering often takes only a few seconds when it does occur. The initial request point is chosen in the past because past blocks are more likely to be available and quickly obtained, thus reducing the time required for initial buffering.

Finally, *LiveSwarms* is a real-time data delivery substrate that we evaluate largely in the context of streaming media, but we note that we do not yet provide a complete streaming video solution. We argue that this layering approach is the right design decision. Tight integration of the video encoding method and the data distribution method is likely to result in challenging implementation issues when attempting to change either one.

4.3 Resource monitoring

Resource monitoring occurs in *LiveSwarms* by periodically computing the *resource index* of the overlay. We use a form of the definition of resource index presented in [15]. For a swarm with broadcast rate r, peer set S and upload capacity of client p denoted U_p, the resource index is:

$$\text{resource index}(S, U, r) = \frac{\sum_{p \in S} U_p}{r \times (|S| - 1)}$$

This quotient represents the current balance of upload capacity and download demand in the overlay system. The numerator reflects the sum total of upload capacity in the swarm, including the broadcaster. The denominator represents the download demand, which in our context is simply the broadcast rate multiplied by the number of downloading peers. This includes all users except the broadcaster, which need not consume any resources. When this function is 1, the upload capacity of the peers in aggregate exactly matches their download demands. When it dips below 1, the broadcast rate cannot be supported, and when it is above 1 there are excess upload resources in the swarm.

With knowledge of the resource index, the tracker may react to resource constrained situations in two ways: either limit drains on the resources or increase the resources available. *LiveSwarms* provides for both of these through simple modifications to the tracker described below. Additionally, because of BitTorrent's tit-for-tat fairness, peers have an incentive to contribute as much as possible to the system, helping avoid resource poor situations in the first place.

First, in an attempt to reduce the burden of new users, the tracker can prevent new clients from swarming over old data during their initial buffering period. This can be easily accomplished if the tracker issues each joining peer a certificate allowing them to download blocks before the current one. Peers would exchange these certificates, with the net effect that new peers place no additional burden on the system unless the system has excess capacity.

Second, since the tracker is capable of monitoring the overall health of the system via the resource index, it can decide when high resource peers need to be added to the system in order to stabilize it. The strategy of integrating stabilizing resources has been advocated in [15] in the context of the End System Multicast (ESM) project. In overlay multicast strategies where the topology is carefully constructed, excess resources must be carefully placed in order to maximize their usefulness. In the case of the tree based topology used by ESM, stabilizing resources are most effective when placed close to the broadcasting root. However, the shuffling required to maintain a structured topology—like ESM's tree—is likely to be a delicate operation when integrating stabilizing resources, particularly during the resource constrained period when they are needed. By contrast, stabilizing resources are treated just like any other peer in by *LiveSwarms*. When a high capacity peer connects, its influence percolates throughout the overlay. We discuss this further in Section 6.5.

5 Evaluation methodology

Given this design, we seek to establish several fundamental properties regarding the feasibility of a swarming approach to providing scalable live broadcast. We first show that a swarming approach can successfully provide loss-free playback with common case behavior of client arrivals, departures, and upload capabilities. Second, we determine that swarms are naturally robust to flash crowd conditions and sudden user departures. Next, we provide evidence that swarming techniques can successfully cope with periods of high churn. We then explore the limits of our design by studying resource constrained situations and provide guidance for avoiding poor swarm performance. Finally, we further explore how stabilizing resources can be integrated into a swarming system, and restore the ability of our system to cope with the reality of asymmetric connectivity due to NATs and firewalls.

To evaluate *LiveSwarms* we deployed our modified BitTorrent client on the Emulab network testbed [30]. Below, we describe the particular details of how we evaluated our system and what metrics we used to gauge its success.

5.1 Experimental setup

We make several assumptions which reduce the complexity of the emulation environment. First, because the majority of end hosts have several times more download capacity than they do upload capacity, we assume that the only limiting factor in transfers is upload capacity. In doing this, we do not limit the download rate of hosts in any way, but our experiments show that this never results in unrealistic rates.

In running our experimental broadcasts, we evaluate our performance in terms of the amount of time clients spend buffering because this is the likely metric by which an end user will measure the effectiveness of a video broadcast tool.

Because the constraint we explore was access link upload bandwidth, we model connections by putting all of the hosts on a single LAN and using the software upload rate-limiter provided in the official BitTorrent implementation to limit how much each host could contribute. We do not impose any propagation delay or packet loss on our hosts in the experiments presented. These factors had no significant impact on our results for several test broadcasts conducted both with and without having them activated.

The data rate in all our broadcasts is considered to be 250 kilobits per second. Depending on the video codec, this is generally sufficient to provide 30 frames-per-second playback at a resolution of 320×240 and stereo sound. We use a block size of 128 kilobytes.

5.2 Churn model

As pointed out by existing studies of streaming workloads on the internet, system churn can be modeled by characterizing the arrival process of new clients and the duration of their stay in the system. These studies show that the arrival process can be characterized as a Poisson arrival process (i.e., exponential interarrivals) over short timescales, but the arrival rate could have cyclic variations over long periods [2, 28, 27, 26]. The studies also show that the mean time between arrivals could be as low as one second or less for the most popular streams [27, 26]. The session durations are heavy-tailed [27, 26], with just a few clients staying in the system for long durations. We use these findings to generate workloads where interarrival times are drawn from an exponential distribution and the session durations are drawn from a Pareto distribution. The relative ratio of the means of the two distributions determines the average number of users in the system.

5.3 Upload capacity

Upload capacity (Kbps)	Percentage of peers
64	20%
192	40%
500	25%
2500	15%

Table 2: Our upload capacity distribution.

In considering how to allocate upload capacity to hosts in our experiments, we attempted to reconcile observations from several real-world measurement studies. Saroiu, et al. derived capacity estimates for participants in the Gnutella peer-to-peer network [24]. Following Bharambe, et al. [5], we discard the tail of dialup users. Table 2 is the result with each capacity halved. We use this distribution in many of the experiments described in Section 6. We dampened the capacities in this way to more accurately reflect likely real world situations. The resource excess from the original distribution corresponds roughly to the "optimistic" resource availability presented in [26]. In order to provide a more conservative view of likely user capacities, we elected to more closely match the "pessimistic" estimations from that paper.

Unless otherwise specified the broadcaster's upload capacity is fixed at 5000 Kbps in order to represent a reasonably well provisioned source.

6 Results and evaluation

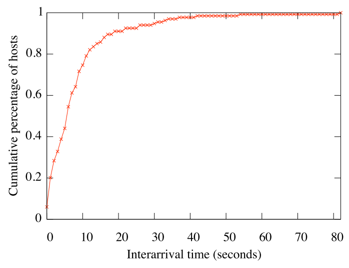
In order to fully understand how well *LiveSwarms* would work as a substrate for live video or audio broadcasts, we evaluated our system under a variety of conditions designed to reflect real-world situations as well as extreme circumstances. Notably, we perform no admission control whatsoever, even when this would certainly help the system. Our results without any admission control are intended to inform such policies during actual deployment.

6.1 Moderate churn

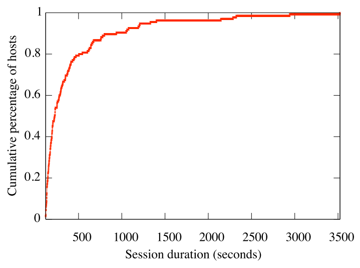
To evaluate *LiveSwarms* in a likely broadcast setting, we first tested a broadcast across a pool of 135 peers with upload capacities drawn from the values of the distribution shown in Table 2. The broadcast duration was 1 hour with interarrival times and durations given in Figures 1(a) and 1(b), respectively. The number of active peers throughout the broadcast is shown in Figure 1(c). The average amount of time required to obtain a 15 second initial buffer was 11 seconds across all peers for the entire broadcast, and all broadcast data was received in time for playback demands.

Our intent in presenting this example is to establish a performance baseline for later evaluations and to describe the behavior of our implementation in the common case of rapid early arrivals and short durations with moderate resource availability as well as to study some basic aspects of our approach. Figure 2(a) displays the aggregate traffic characteristics of the entire swarm. The resource utilization of the peers in aggregate tracks increases in capacity as new users join, and is strictly greater than the download demands for all but the first few seconds of the broadcast, indicating that we quickly integrate new resources into the swarm as they arrive. Figure 2(b) displays the resource index throughout the broadcast. The over-provisioning indicated here corresponds to the gap between upload capacity and broadcast demands in Figure 2(a). As users depart from the system, the resource index is dominated by the large capacity of the broadcaster.

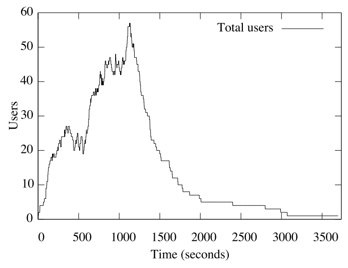
We demonstrate the buffering behavior of new users in the system in Figure 2(c). The download consumption of the first 10 hosts in shown. We observe that as new users join the broadcast, they experience a surge in download performance before leveling off to the broadcast rate. This behavior is a result of selecting an initial playback point 30 seconds in the past to reduce the time needed to fill a 15 second buffer. The rationale is that a resource rich swarm can support new users that initially contribute very little for a brief time; new users can more quickly obtain old data, thus reducing their initial buffering period. Once they have consumed all available old



(a) CDF of interarrival times

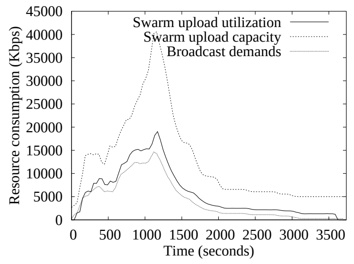


(b) CDF of durations

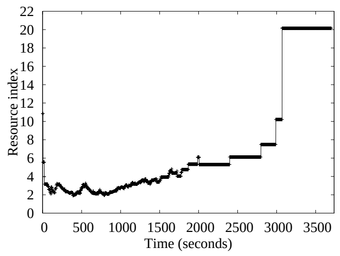


(c) Total users

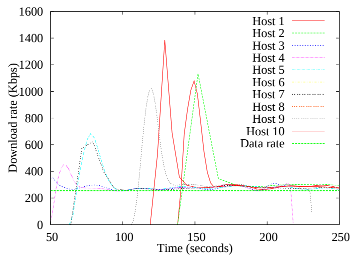
Figure 1: Synthetic churn characteristics for a resource rich 135 user broadcast.



(a) Absolute traffic consumption and capacity



(b) Resource index during broadcast



(c) Download throughput of first 10 hosts

Figure 2: Traffic characteristics for a resource rich 135 user broadcast; upload capacities taken from Table 2.

data, their download rate settles at the production rate, and they can contribute to the swarm. Although we indicate that new users initially contribute nothing to existing users, they are capable of swarming over old data amongst one another. This can be especially helpful in mitigating the effects of flash crowds.

Finally, understanding how blocks actually flow through the swarm is important. Since each host is only actively exchanging data with a small subset of its immediately connected peers, it might be expected that blocks would take long paths during their flow through the swarm. In fact, each block traverses an implicit tree in which the degree at each node is approximately the upload capacity of that node divided by the data rate. While this means that many nodes will only have a child or two in this tree, some nodes (and especially the broadcaster) will have very high degree, drastically reducing the depth of the tree. A critical point is that this tree is highly dynamic and unstructured. The round-robin fashion in which the broadcaster distributes blocks as well as the changing set of unchoked peers at each host ensures that different blocks will likely traverse different trees. As a consequence, different nodes tend to obtain different blocks at different times, encouraging exchange of data throughout the swarm. Both aspects of this behavior can be seen in Figure 3, which depicts the path which two consecutive blocks take through the swarm described by Figures 1 and 2. Even in the case of few users and blocks separated by only a few seconds, the path of each differs.

6.2 Flash crowds and sudden departures

Given our successful playback in a typical broadcast setting, we turn our attention to evaluating *LiveSwarms* at extremes as a means to gauge its robustness. Arguably the most challenging problem in constructing a feasible overlay for supporting real-time traffic is periods of rapid arrival encountered during flash crowds.

We consider the following synthetic flash crowd experiment coupled with sudden drop: first, the broadcast is started and a set of 80 peers with capacities drawn from Table 2 connects immediately. After several minutes, 70 new users, also with capacities drawn from Table 2, begin to connect at a rate of 1 per second. After several more minutes, all the original peers depart, leaving only the peers that joined during the flash crowd. Figures 4(a) and 4(b) depict resource demand and availability during the broadcast, with Figure 4(c) displaying the number of late blocks normalized by the number of actively participating hosts. Although we observe some late blocks during the flash crowd, the actual impact from the viewers' perspective is quite small, with an average rebuffering time of only 3 seconds and no user rebuffering for longer than 7 seconds. Since blocks can be received out-of-order, a rebuffering user may need only a few missing

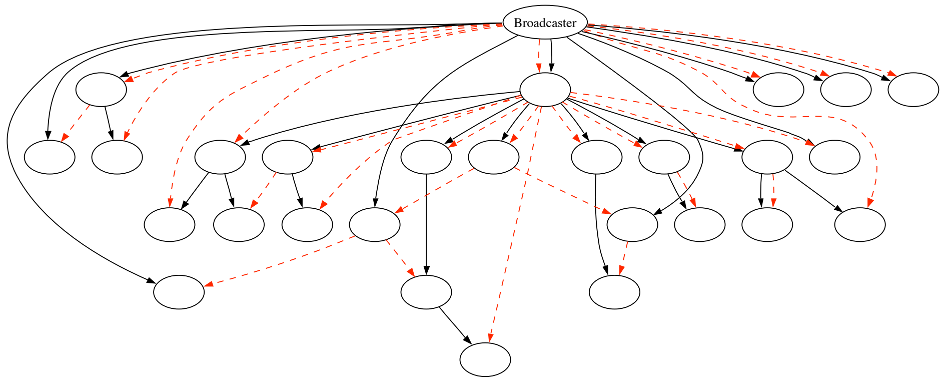
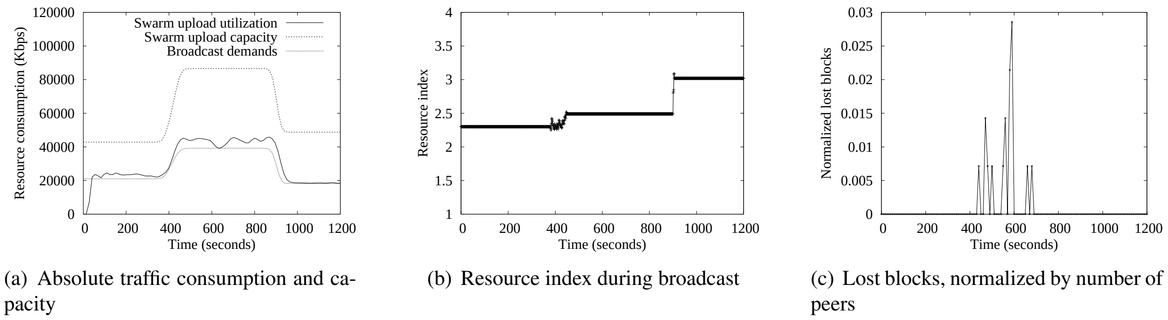


Figure 3: The propagation of two successive blocks early in the broadcast described by Figures 1 and 2.



(a) Absolute traffic consumption and capacity

(b) Resource index during broadcast

(c) Lost blocks, normalized by number of peers

Figure 4: Performance of *LiveSwarms* under a synthetic flash crowd.

blocks in the current 15 second window. This accounts for our quick rebuffering time even in flash crowd conditions.

Interestingly, although rapid arrivals have an impact on playback quality, there are no late block events associated with the sudden departure of the 80 initial users. We attribute this to the large peer pool BitTorrent maintains and its ability to cope with being choked by other peers. In the course of a normal connection, two peers are likely to choke and unchoke each other many times as each independently reasons about the best sources of available data. To a *LiveSwarms* client a peer disconnecting is no different than being choked by that peer—a situation the BitTorrent substrate was designed to handle well. As a result, we inherit robustness to sudden departure through no special effort of our own. This is just one of the benefits of basing our implementation on an existing protocol, and we will return to this issue to discuss further benefits in Section 7.

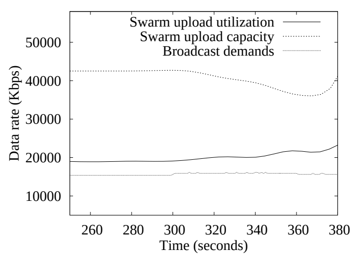
6.3 High churn

Encouraged by the performance of our system under flash crowds and sudden departures, we conducted further experiments to determine the limits of our system under periods of high churn. In the experiment reflected in Figure 5, 60 clients connect to a broadcast and are allowed to stabilize. Afterwards, one client joins every second as a distinct client departs, keeping 61–63 users in the system once the period of high churn begins, as shown in Figure 5(c). Upload capacities are drawn from the distribution of Table 2. In contrast to the flash crowd experiment described in Section 6.2, here there are no stable hosts in the swarm except for the broadcaster. A loss graph is not presented since our results are so positive: only 4 hosts out of 110 experience any rebuffering whatsoever, and those that must do so for only $1, 3, 4$, and 4 seconds, respectively. As expected, this loss occurs during the period of churn and impacts only peers with less upload capacity than the data rate. We attribute this to BitTorrent's tit-for-tat fairness mechanism.

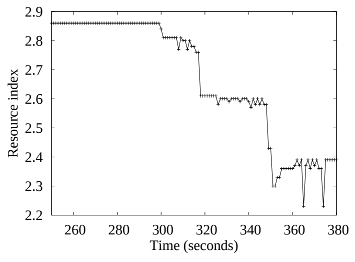
Coupled with the results presented Section 6.2, this experiment suggests that for robustness to high churn and flash crowds, a resource index above 2 is needed in our current implementation. In the next section, we explore the limits of our implementation under more modest churn conditions.

6.4 Periods of constrained resources

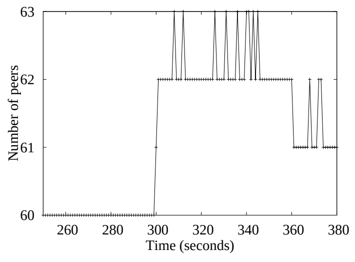
A natural way to evaluate the efficiency of an overlay multicast system is to evaluate its performance in re-



(a) Absolute traffic consumption and capacity

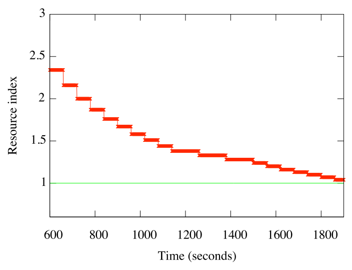


(b) Resource index during broadcast

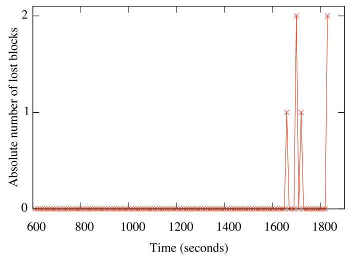


(c) Number of peers

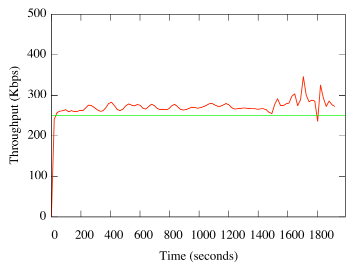
Figure 5: Traffic characteristics for a high churn broadcast.



(a) Gradually reducing resource index



(b) Absolute number of lost blocks across all peers



(c) Broadcast rate with download throughput of the first peer

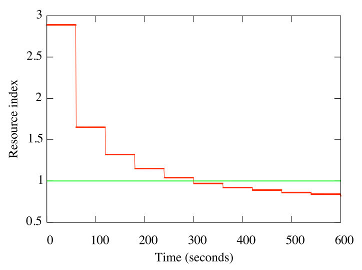
Figure 6: A resource constrained broadcast with one asymmetric host arrival per minute

source constrained situations. We consider the performance of *LiveSwarms* in several of these situations as a means to determine its efficiency limits.

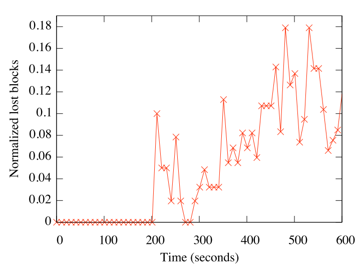
We consider two experiments designed to intentionally cause resource constrained collapse of our overlay method: gradual and sudden arrival of resource-limited peers. Figure 6 displays a summary of the former. In this experiment, users connect to the broadcast at a rate not exceeding one per minute. They each have upload capacity roughly 32% of the broadcast rate, depressing the resource index with each new connection. Comparing Figures 6(a) and 6(b) reveals that peers begin to lose blocks between resource index 1.15 and 1.1. Although we will see that this is not a particularly useful observation when deciding whether a real-world broadcast can be supported, the nature of the losses at the resource index 1 boundary provide interesting insights into the failure properties of *LiveSwarms*. The late blocks displayed in Figure 6(b) are experienced by only two peers among 32—the remaining 30 experience uninterrupted playback. Further, the two peers that experience late blocks essentially starved—each spending more than a minute rebuffering near the end of the broadcast.

In this type of resource constrained situation, we inherit a desirable failure property from BitTorrent's tit-for-tat fairness mechanism. Rather than degrade the performance of all peers, tit-for-tat preferential uploading results in the first peer that cannot competitively provide interesting data being choked very quickly by all its local peers, as they are saturated with transfers that improve their chance of useful data exchange, whereas the choked peer has nothing of interest. Tit-for-tat also tends to disrupt peers in decreasing order of contribution to the system, if possible. In other words, the first users to experience service disruption are the ones that consume more resources than they contribute. Although choked peers can hope for new interesting data from the broadcaster or an optimistic unchoke designed to discover new sources of data, these will not be enough for it to overcome its data deficiency as long as its local peer set is resource constrained. Notice that this has the effect of artificially raising the resource index for the remaining peers, enabling them to achieve superior playback. Since the choked peer consumes little data due to tit-for-tat, its presence does not significantly impact playback quality of its local neighborhood, so long as resource availability does not decrease further.

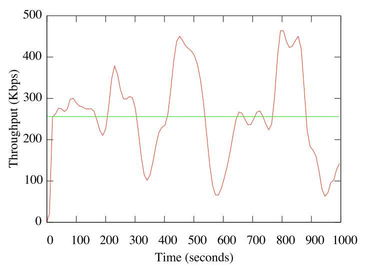
Although gradual arrivals of resource poor hosts allow us to pinpoint the minimum resource index required for steady-state playback, it is not particularly informative in the case of churn. To investigate the failure properties of our design in this case, we slightly modified the previously described experiment in terms of arrival rate. Every minute, a group of 10 peers with identical upload



(a) Suddenly reducing resource index



(b) Lost blocks, normalized by the total number of peers



(c) Broadcast rate with download throughput of the first peer

Figure 7: A resource constrained broadcast with 10 sudden arrivals per minute

capacity at 65% of the data rate connects to the swarm. This has the effect of suddenly reducing the resource index, as seen in Figure 7(a), and, because new peers request old data, suddenly consuming excess swarm upload capacity. These twin drains on swarm resources first impact playback when the resource index makes its abrupt change from roughly 1.3 to 1.2, as shown in Figure 7(b).

We see that even though the swarm is well equipped to provide steady-state playback with 20% more capacity than required, it cannot cope with the resource drain of sudden arrivals. Figure 7(c) shows the download rate of the first peer and gives further insight into this phenomenon. Each dip in the throughput reflects the sudden arrival of new peers suddenly consuming the resources of other swarm hosts. Although new peers can swarm over old data in addition to obtaining it from existing peers, existing users like the one shown are forced to incur a brief drop in download rate due to a sudden increase in control traffic and optimistic unchokes. Once the new users have been integrated into the swarm and catch up to the data production point in their consumption, existing peers can exploit the swarm's true capacity. Peers then briefly download far faster than the data production rate, since there is an excess of interesting blocks. Comparing Figures 7(c) and 6(c) makes evident that the impact of this dip-integration-surge behavior is dependent on the number and arrival rate of new peers and the excess resources present in the swarm. If there are sufficient resources to feed both old and new users, it will not occur at all. Otherwise, existing peers will experience a brief drop in throughput. From a broadcaster's perspective, it is important to provide some form of admission control in the tracker or require clients to set their buffer length based on the minimum resource index expected during playback.

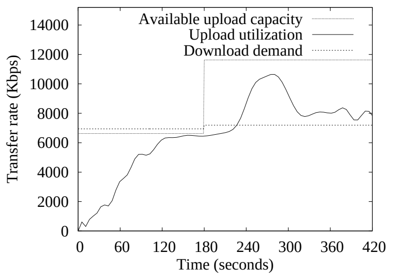


Figure 8: Impact of a high capacity peer connecting to a resource constrained swarm

6.5 Integrating stabilizing resources

Although real-world measurements have indicated that video broadcasts are usually over-provisioned, this is certainly not always the case. Often broadcasts are either temporarily or perpetually under-provisioned, and additional resources are required. From an administrative point of view, a straightforward solution is to simply attach nodes with large upload capacity to the swarm when the resource index dips below a certain threshold. Although we did not discuss it in Section 6.2, our experiment designed to examine flash crowd behavior also demonstrated our successful and rapid integration of new resources when they were added to a system with some excess capacity. The key observation is that stabilizing resources are most effective if they are added well before the resource index approaches 1. We have implemented just such a threshold strategy in the *LiveSwarms* version of the BitTorrent tracker. Because we maintain an up-to-date calculation of the resource index, it is straightforward to determine when to add excess capacity to the swarm.

The tendency of *LiveSwarms* to selectively fail peers that cannot significantly contribute to their local peer set during periods of resource constraint begs the question: what if a high capacity peer joins the swarm when it is already resource constrained? If peers preferentially upload only to those that can offer them interesting data in return, one might suspect that the system could never recover from a resource constrained state even if high capacity peers joined and caused the resource index to rise. Fortunately, this is not the case. Although the integration of stabilizing resource is definitely hastened by introducing them well before the resource index drops below 1, it is still possible to recover from such a situation. Recall that, in order to discover peers that might provide greater transfer rates than those a peer is currently exchanging blocks with, the BitTorrent substrate optimistically unchokes peers in a round-robin fashion without regard to their contribution. We see in Figure 8 that this is enough to integrate a high capacity peer that joins during a period of resource constraint. Initially, 30 peers with upload capacity less than the broadcast data rate connect and attempt to support the broadcast. As expected, most peers are in a state of perpetual rebuffering. After 180 seconds, a high capacity peer joins and raises the resource index to approximately 1.6. Although roughly a minute passes before the new peer can obtain sufficient data to utilize its upload capacity, eventually the swarm learns the value of exchanging data with the new peer. Finally, after a brief surge in download rate during which the existing peers catch up to the current data production point, all peers experience flawless playback.

6.6 NATs and firewalls

One of the most disruptive attributes of end hosts when attempting to construct an efficient overlay network is the presence of network address translation (NAT) and firewalls. Because these devices prevent clients from receiving incoming connections, they can make constructing an overlay difficult, as the capability for bidirectional connectivity is often assumed. Fortunately, the network structure of *LiveSwarms* is naturally robust to such connectivity restrictions, although we conjecture that some further improvements are possible.

We have duplicated several of our experiments with connectivity restricted hosts that can form outgoing connections but not receive incoming ones. Specifically, we repeated the experiment described by Figures 1 and 2 with between 0% and 50% of the hosts having restricted connectivity. We noted no difference in playback performance beyond a slight (3 seconds on average) increase in initial buffering until we reached between 40%-50% nodes with connectivity restrictions. In this range, hosts began experiencing moderate rebuffering, generally between 30 seconds to 1 minute collectively among all swarm participants.

Our results in these duplicate experiments suggest several observations regarding *LiveSwarms* performance in the face of connectivity restrictions. First, BitTorrent graphs are usually highly connected, with each peer maintains a large set of local connections (usually 40 or more in large swarms). This results in restricted hosts usually being connected to several unrestricted hosts with which they can exchange blocks. Second, although the random graph structure provides some resilience against the problem of capacity utilization for restricted hosts, it does not solve the problem completely. During resource constrained periods or if an overwhelming number of hosts are restricted, performance for those hosts will suffer. Finally, our protocol design leaves significant room for improvement. A more intelligent tracker might be designed to return peers intelligently—attempting to build up local neighborhoods with restricted and unrestricted peers in a capacity balance. This would require no special changes to the *LiveSwarms* protocol, and we leave its investigation as future work.

7 Discussion and future work

We have presented and evaluated a complete design for a live broadcast transport layer. However, there are several implications and potential weaknesses of our method that we have not yet treated.

While our experiments were limited to broadcasts of at most several hundred hosts, we believe *LiveSwarms* has the capacity to scale to thousands of hosts. There is some evidence to suggest this. First, the experiments conducted in this paper were limited in size only by the resources available to us in the Emulab testbed. We performed no special tuning of our software when we made the transition from 10 host development experiments to the larger experiments presented in this paper. Second, our modifications to the BitTorrent client were slight, and BitTorrent has been successfully used to support file distribution for several thousand concurrent users.

Regarding scalability, one might suspect that the tracker—a centralized resource—might eventually become overwhelmed. Here again practical experience suggests that this will not be a problem, as popular BitTorrent trackers regularly coordinate multiple swarms with thousands of users.

One might also expect that with a larger user population, the number of peers for each node might be a much smaller fraction of the size of the overlay, thereby making it difficult to find the necessary blocks. However, since the number of blocks a *live swarm* is distributing is generally small, block availability is unlikely to be an issue.

Another improvement to client performance regards initial buffering. Clients may easily observe the rate at which they fill their initial buffer and if it was substantially faster than the data rate, they could begin playback nearly immediately. This exploits the fact that when requesting old data for initial buffering, we explicitly request blocks in order. This has the potential to drastically decrease the time which new users wait to begin playback provided there are sufficient resources.

The question of tracker scalability provides an example of the benefits of relying on an existing and actively developed system as a basis for data distribution. The BitTorrent community has successfully implemented an alternate tracker system based on the distributed hash table system Kademlia [20]. Although we did not adapt our tracker modifications to this facility, it serves as an additional message passing layer than we inherit from the BitTorrent code base.

Finally, we point out two related avenues of future work. Our ability to quickly integrate additional resources and monitor the resource index of the swarm makes overprovisioning a broadcast a straightforward solution to the challenging realities of churn. However, our current strategy for monitoring the health of the swarm—requiring clients to report upload capacity—is simplistic. In practice, it is likely that users' upload capacity will vary over the course of a broadcast, and some users may simply inaccurately report their capacity. Client upload characteristics may also be intentionally misreported as a means to achieve service without contribution or to undermine the broadcast. If users are certain that a given broadcast will be overprovisioned at a particular level, regardless of client contribution, they might limit their reported capacity intentionally. Although this would not impact client performance, it may nonetheless prove to be a problem in practice, leaving open an interesting question regarding a game theoretic analysis of BitTorrent-style content distribution with performance guarantees.

8 Conclusion

In this paper, we have presented the design, implementation, and evaluation of *LiveSwarms*, an end host multicast system using unstructured overlays for the delivery of streaming data with a small playback delay. *LiveSwarms* is derived from the popular file download system BitTorrent, enhanced to work for streaming data. The resulting system is quite simple to implement and understand. By leveraging BitTorrent, we hope to bring the promise of end host multicast to a broad user community.

We demonstrate via trace-based experiments that *LiveSwarms* can successfully support the real-time delivery demands of hundreds of hosts with heterogeneous and asymmetric bandwidth capacities, and provided the swarm has sufficient resources, it can support very high churn rates, flash crowds, and sudden departures. Like any real-time delivery system, deadlines can be missed when the resources of the system are inadequate to meet the demand, but we show that the system is self-healing, able to discard a few resource poor nodes in order to keep the rest of the swarm healthy, and able to meet their deadlines. We also show that *LiveSwarms* can detect when deadlines are about to missed, proactively adding stabilizing resources to preserve delivery guarantees.

References

[1] Akamai Technologies, Incorporated. `http://www.akamai.com`.

[2] ALMEIDA, J. M., KRUEGER, J., EAGER, D. L., AND VERNON, M. K. Analysis of educational media server workloads. In *Proceedings of NOSSDAV* (2001).

[3] ANDERSEN, D. G. Improving end-to-end availability using overlay networks. MIT Ph.D. dissertation, 2005.

[4] Azureus. `http://azureus.sourceforge.net/`.

[5] BHARAMBE, A. R., HERLEY, C., AND PADMANABHAN, V. N. Analyzing and improving bittorrent performance. Tech. rep., Microsoft Research, 2005.

[6] BIRMAN, K. P., HAYDEN, M., OZKASAP, O., XIAO, Z., BUDIU, M., AND MINSKY, Y. Bimodal multicast. *ACM Trans. Comput. Syst. 17*, 2 (1999).

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

[8] BYERS, J., CONSIDINE, J., MITZENMACHER, M., AND ROST, S. Informed content delivery across adaptive overlay networks. In *Proceedings of SIGCOMM* (2002).

[9] BYERS, J. W., LUBY, M., MITZENMACHER, M., AND REGE, A. A digital fountain approach to reliable distribution of bulk data. In *Proceedings of SIGCOMM* (1998), pp. 56–67.

[10] CASTRO, M., DRUSCHEL, P., KERMARREC, A.-M., NANDI, A., ROWSTRON, A., AND SINGH, A. SplitStream: high-bandwidth multicast in cooperative environments. In *Proceedings of Symposium on Operating System Principles* (2003), pp. 298–313.

[11] CHU, Y., RAO, S., SESHAN, S., AND ZHANG, H. Enabling conferencing applications on the internet using an overlay muilticast architecture. In *Proceedings of SIGCOMM* (2001), pp. 55–67.

[12] COHEN, B. Incentives build robustness in bittorrent. `http://www.bittorrent.com/bittorrentecon.pdf`, 2003.

[13] eDonkey. `http://www.edonkey2000.com`.

[14] FRANCIS, P. Yoid: Extending the internet multicast architecture. Available at `http://www.icir.org/yoid/docs/`, 2000.

[15] HUA CHU, Y., GANJAM, A., NG, T. E., RAO, S. G., SRIPANIDKULCHAI, K., ZHAN, J., AND ZHANG, H. Early experience with an internet broadcast system based on overlay multicast. In *Proceedings of USENIX Annual Technical Conference* (2004).

[16] HUA CHU, Y., RAO, S. G., AND ZHANG, H. A case for end system multicast. In *Proceedings of SIGMETRICS* (2000), pp. 1–12.

[17] JANNOTTI, J., GIFFORD, D. K., JOHNSON, K. L., KAASHOEK, M. F., AND O'TOOLE, JR., J. W. Overcast: Reliable multicasting with an overlay network. In *Proceedings of Symposium on Operating Systems Design and Implementation* (2000), pp. 197–212.

[18] KOSTIC, D., BRAUD, R., KILLIAN, C., VANDEKIEFT, E., ANDERSON, J. W., SNOEREN, A. C., AND VAHDAT, A. Maintaining high bandwidth under dynamic network conditions. In *Proceedings of USENIX Annual Technical Conference* (April 2005).

[19] KOSTIC, D., RODRIGUEZ, A., ALBRECHT, J., AND VAHDAT, A. Bullet: High bandwidth data dissemination using an overlay mesh. In *Proceedings of Symposium on Operating System Principles* (2003).

[20] MAYMOUNKOV, P., AND MAZIERES, D. Kademlia: A peer-to-peer information system based on the xor metric. In *International Workshop on Peer-To-Peer Systems* (2002).

[21] PADMANABHAN, V., WANG, H., CHOU, P., AND SRIPANIDKULCHAI, K. Distributing streaming media content using cooperative networking. In *Proceedings of NOSSDAV* (2002).

[22] PADMANABHAN, V. N., WANG, H. J., AND CHOU, P. A. Resilient peer-to-peer streaming. In *Proceedings of International Conference on Network Protocols* (2003).

[23] PAI, V., TAMILMANI, K., SAMBAMURTHY, V., KUMAR, K., AND MOHR, A. Chainsaw: Eliminating trees from overlay multicast. In *International Workshop on Peer-To-Peer Systems* (2005).

[24] SAROIU, S., GUMMADI, P. K., AND GRIBBLE, S. D. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking* (2002).

[25] SHERWOOD, R., BRAUD, R., AND BHATTACHARJEE, B. Slurpie: A cooperative bulk data transfer protocol. In *Proceedings of INFOCOM* (2004).

[26] SRIPANIDKULCHAI, K., GANJAM, A., MAGGS, B., AND ZHANG, H. The feasibility of supporting large-scale live streaming applications with dynamic application end-points. In *Proceedings of SIGCOMM* (2004).

[27] SRIPANIDKULCHAI, K., MAGGS, B., AND ZHANG, H. An analysis of live streaming workloads on the internet. In *Proceedings of SIGCOMM Conference on Internet Measurement* (2004), pp. 41–54.

[28] VELOSO, E., ALMEIDA, V., MEIRA, W., BESTAVROS, A., AND JIN, S. A hierarchical characterization of a live streaming media workload. In *Proceedings of SIGCOMM Workshop on Internet measurement* (2002), pp. 117–130.

[29] WANG, W., HELDER, D. A., JAMIN, S., AND ZHANG, L. Overlay optimizations for end-host multicast. In *Networked Group Communication* (2002), pp. 154–161.

[30] WHITE, B., LEPREAU, J., STOLLER, L., RICCI, R., GURUPRASAD, S., NEWBOLD, M., HIBLER, M., BARB, C., AND JOGLEKAR, A. An integrated experimental environment for distributed systems and networks. In *Proceedings of Symposium on Operating Systems Design and Implementation* (Dec. 2002), pp. 255–270.

[31] ZHANG, B., JAMIN, S., AND ZHANG, L. Host multicast: A framework for delivering multicast to end users. In *Proceedings of INFOCOM* (2002).