Chunkyspread: Multi-tree Unstructured Peer-to-Peer Multicast *

Vidhyashankar Venkataraman, Paul Francis[†] John Calandrino[‡]

{vidya, francis}@cs.cornell.edu, jmc@cs.unc.edu

ABSTRACT

The latest debate in P2P and overlay multicast systems is whether or not to build trees. The main argument on the anti-tree side is that tree construction is complex, and that trees are fragile. The main counter-argument is that non-tree systems have a lot of overhead. In this paper, we argue that you can have it both ways: that one can build multi-tree systems with simple and scalable algorithms, and can still yield fast convergence and robustness. This paper presents Chunkyspread, a multi-tree, heterogeneous P2P multicast algorithm based on an unstructured overlay. Through simulation, we show that Chunkyspread can control load to within a few percent of a heterogeneous target load, and how this can be traded off for improvements in latency and tit-for-tat incentives.

1. INTRODUCTION

In 1997 and 1998, Francis and Zhang respectively independently argued that IP multicast was going nowhere, and that some form of end-system (P2P) multicast is needed to bring multicast to the masses ([3], [4]). Nearly a decade later, P2P multicast has itself gone nowhere, this in spite of the success of other P2P technologies such as file sharing and swarming. Part of the reason for this is surely that multicast is something of a niche application. It is only really needed for live or near-live streaming, whereas most content distribution is non-live. Nevertheless, there are some multicast applications out there, which today are largely handled by infrastructure-based overlays (i.e. Akamai) or IP multicast (in enterprise settings). We believe, however, that there are still substantial improvements that can be made to P2P multicast algorithms, and that these improvements may yet lead to widespread use of this technology.

In this paper, our focus is on non-interactive multicast applications that can grow to very large scale (many thousands of recipients) and can handle a wide range of volumes. A canonical application for us is the broadcast of a sports event, where the content may be a simple text description of the score and important events (low volume), an audio play-by-play (medium volume), or video (high volume). Here, 10 or 20 seconds of delay is tolerable, indeed even necessary in the form of receiver play-out buffer to smooth over short-term disruptions in network or OS performance [17]. This delay-tolerance also means that we can assume one or a small number of content streams. Even if there are many sources, they can first transmit to a tree root, and from there the aggregated stream can be multicast.

In addition to large scale, an important requirement is to have fine-grained control over member load. The need for this stems from fairness, utility, and performance arguments. Fairness suggests that each member node should transmit the same volume that it receives. Where utility is valued over fairness, nodes with more capacity should transmit more, should they be willing to do this. Good performance requires that any given node should not be a bottleneck (that is, not be called on to transmit more than it can). For all of these factors, a single-tree approach comes up short. It is certainly not fair: many nodes transmit nothing, while other nodes transmit at least two times the stream volume[1]. Nor does a single tree provide enough granularity to effectively utilize transmit capacity. A node with enough capacity to transmit say 150% of the stream volume must nevertheless serve as a leaf and transmit nothing. As a result, we believe, as others have argued ([7], [8]), that a *multi-path* approach is necessary. By multi-path, we mean where each node receives portions of the multicast stream via different routes. Multi-path may be achieved through multiple trees, as in SplitStream [7], or through a so-called *treeless* approach, as in Bullet [5] or Chainsaw [8]. We say "so-called" treeless, because the goal of Bullet or Chainsaw is nevertheless that each individual packet or block of packets traverses a tree.

The question, then, is not whether to build trees, but rather at what granularity: per slice, or per block or packet? The difference is important. With a per-block granularity, each node explicitly informs its neighbors of which blocks it has, and explicitly requests from each neighbor which blocks it would like to receive. This represents a substantial overhead: with an average node degree of 20 (as used in [8]), this means an additional 20 packets (10 sent and 10 received on average), for every data block received. If the stream is low volume, this overhead can be many times the stream volume. For higher volume applications, which Bullet and Chainsaw target, the overhead is more acceptable, but is nevertheless worth trying to avoid.

With a per-slice granularity, nodes maintain a long-term parent-child relationship with respect to each slice (where a slice is defined as every M^{th} packet of a data stream, M being the number of slices). As a result, once the trees are established, there is virtually no per-packet overhead. On the other hand, if a node crashes or otherwise stops performing

*This work is supported in part by National Science Foundation grant ANI-0338750, and DARPA project FA8750-04-2-0011.

[†]Cornell University

[‡]University of North Carolina at Chapel Hill

[1]Assuming all interior nodes have a fanout of at least two. Otherwise, paths may be unacceptably long.

adequately, all of its offspring in the tree will suddenly stop receiving some packets until the tree can be re-built. Furthermore, building a tree isn't instantaneous or free: in case of high churn, a complex tree-building algorithm may take a long time to repair completely.

Fortunately, tree-building algorithms do not need to be complex. There are two aspects to building trees: parent selection and loop detection and avoidance. In P2P environments, where one can afford a moderate amount of processing overhead per packet (in contrast with high-speed routers, for instance), loop detection is fast and easy. One needs to only tag all data packets with the identities of every node that forwarded the packet. This can be made quite efficient in terms of packet size as well as processing using bloom filters [16] (with a small probability of false positives). Loops are detected immediately by the first packet that traverses the loop. This packet can be either a data packet sent by the application, or, in the absence of such packets, a probe packet transmitted by a node to its children. Loops can be avoided by having nodes advertise the bloom filters they receive to their neighbors. A given node does not select a neighbor as a parent if the node itself appears in the parent's received bloom filter.

Using this method of loop avoidance and detection, tree building is simple. For each stream, each node selects a loop-free parent from among its neighbors. When a loop is detected (rarely), the node selects a different parent. Flooding or swarming (as in Chainsaw) may be used to kick start any given tree. While tree-building this way is indeed fast and easy, the problem is that inefficient trees may result. Some nodes may have too much transmit load, and some paths may be longer than necessary. These inefficiencies, however, can be improved over time (where time here is measured in seconds, not tens of seconds) by switching to different parents when doing so improves some measure of efficiency, such as load, latency, tit-for-tat, path disjointness, link stress, and so on.

In this paper, we present Chunkyspread, a simple multi-tree multicast protocol and show through simulations that it achieves good robustness and a fine-grained control over heterogeneous load, yet avoiding the per-packet overhead of treeless approaches. We also show through simulations that without affecting load by much, we can still fine-tune the trees with respect to other constraints such as latency and tit-for-tat.

2. THE CHUNKYSPREAD APPROACH

Given this background, we can now describe our approach. As with Splitstream, we use striping over M multiple trees to get fine-grained control over heterogeneous load[2]. Each node has a target and a maximum load, both expressed as a number of stripes from zero upwards. There are no constraints on which stripes a node transmits: it may transmit any given stripe any number of times as long as it operates below its maximum.

As with Yoid [3], all nodes form a random graph. Unlike Yoid, the node degree of any node in the random graph is proportional to that node's target load (with some small probabilistic error). This requires that each node knows the minimum target load of all nodes, and the node degree associated with that minimum. We assume that this information is known by the application. The actual set of neighbors changes with membership churn such that the number of neighbors for any given node stays roughly the same over time. We discuss later how these neighbors are found. In addition to its random neighbors, a node may know of some number of nearby nodes (probably as measured by latency).

The node (or nodes) that is the source of the multicast stream also joins the random graph. This node, called the *true source*, has M random neighbors, and transmits each slice to one neighbor. These neighbors are called *slice sources*, and each is effectively the root of a multicast tree.

With this structure in place, each node runs an ongoing process whereby it periodically exchanges local information (load, latency, and looping) with its neighbors, and uses this information to determine the appropriate parent-child relationships for each tree. Within a set of *constraints*, each node pair locally tries to determine the *best* parent-child configuration. The constraints we currently have are as follows. First, there must of course be no loops. These are avoided and detected as described above. Second, any pairwise volume-based tit-for-tat constraints in force must be satisfied. Third, no node may transmit more than its maximum load. Other than the looping constraint, it is conceivable that nodes may choose to violate these constraints for short periods of time, for instance to respond quickly to node failures. We currently do not implement this.

The criteria we use to converge to the *best* parent-child configuration are latency and load. These criteria do not always align: there may be situations where improving latency hurts load, or vice versa. As load is the more important of the two criteria, we first insure that load is good (each node's actual load is near its target load), and only then do we try to improve latency.

In the remainder of this section, we provide more detail about how to build the random neighbor graph, and how load and latency are fine tuned.

2.1 Random Graph Construction:

The random neighbor graph is the underpinning of Chunkyspread (in much the same way as RanSub [14] is the underpinning of Bullet). We use an algorithm called Swaplinks [1], which uses weighted random walks to build random graphs and to discover random nodes over that graph. The walks are weighted in such a way that nodes have fine-grained statistical control over their node degree, as well as the probability with which they are selected by random walks. Swaplinks is simple and unstructured, scales easily to any number of members, and reacts quickly to churn. Note that both Bullet and Chainsaw could utilize Swaplinks for neighbor discovery, especially if they required heterogeneity.

2.2 Load:

Each node lets its neighbors know the following information: its current load (the number of slices it is transmitting), its target load TL, its maximum load ML, and upper and lower load thresholds (ULT and LLT) within which it is considered "satisfied", and outside of which it is considered "overloaded" or "underloaded". Typical numbers are $TL = 16$, $ML = 20$, $ULT = 14$, and $LLT = 18$. Each node

[2]While by load we mean transmit load, nodes in Chunkyspread can choose to receive fewer than all the slices, for instance by using Multiple Description Codes, and can independently set transmit load and receive load. Nevertheless, in this paper and in our current implementation, we assume that all nodes are receiving all stripes.

periodically checks to see if it has any overloaded parents and underloaded *potential parents* (neighbors that would not produce a loop were they chosen as parents) for any given slice. If so, then there is an opportunity for a *switch*: dropping the current overloaded parent (thus reducing its load) and adding the underloaded neighbor (thus increasing its load).

To do this, the node informs its overloaded parent of the loads of a subset of its *best* underloaded potential parents. The parent, which may receive similar information from multiple children, picks the best candidate (the child's neighbor with the least load), and instructs the selected child to make the switch. Note that, because conditions may have changed during this exchange, each request for a switch contains the parameter in force (either load or latency, as discussed later), when the switch decision was made. If this parameter has changed considerably, the switch is aborted. Note also that, in order to prevent packet loss or duplication, the actual switch is coordinated in that the child identifies the future packet at which the old parent should stop transmitting, and the new parent should start transmitting.

2.3 Latency:

Once each node's neighbors are satisfied with respect to load, the node looks for parent switches that can improve the latency with which it receives packets. We use a novel trick that allows us to measure the relative latency with which each neighbor receives each slice without requiring synchronized clocks. Specifically, each node measures the delay at which it receives packets from each stream *relative to each other*. The idea is simple: a node close to a slice source in a tree will receive packets for that slice relatively *sooner* than it will receive comparable packets of other slices. If a node has a parent that is receiving a given slice *late* (relative to its other slices), and a potential parent that is receiving the same slice relatively *early*, then it should switch parents (as long as both neighbors' loads remain satisfactory). Note that nodes only make such switches if the expected improvement in latency is beyond a certain threshold.

In our current implementation, as long as all of a node's neighbors' load is in the satisfactory range, no improvements in load are made even if such improvements could be made without hurting latency. Because of this, the load balancing algorithm always makes forward progress and is guaranteed to converge. We could relax this restriction and potentially improve load further. However, we would then need to maintain a short history of recent switches to prevent rapid oscillations. The latency improving algorithm will almost always converge. On those occasions where it does not, the oscillation will be relatively slow because of the time required to produce a confident measure of delay.

3. EVALUATION

We have performed a series of experiments on a packet-level, event-driven simulator. Our simulation topologies are created by placing member nodes at random edge locations on network topologies having 5050 routers, generated by the GT-ITM topology generator [10]. Message delays are determined using the resulting distance metric. We did not simulate message loss. To scale the simulation, the simulator does not explicitly generate data packets. Control decisions, however, do take into account the delay that transmitted data packets would have seen. The random overlay is constructed using a trace file generated offline by a SwapLinks simulator. This trace generates node joins and leaves, and triggers neighbor add and delete events.

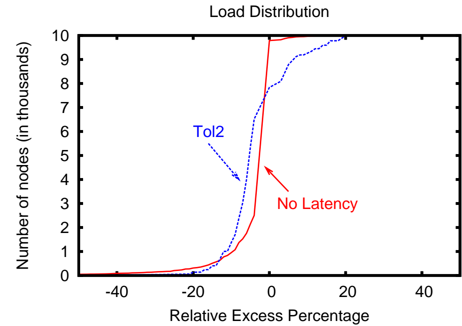

Figure 1: Excess Load Distribution

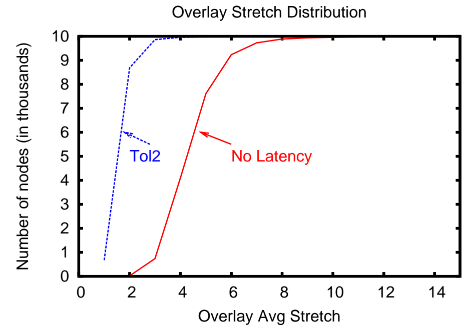

Figure 2: CDF of average overlay stretch

The nodes participating in the multicast in all the simulations have a uniform download of 16 slices. For the heterogeneity experiments, we assign each node a target upload between 5 and 28 slices uniformly at random (representing total node degrees in Swaplinks between 8 and 50). This results in an average upload capacity of 16.5, thus giving the total system adequate capacity to support the multicast volume. We set ULT and LLT to be within $\frac{2}{16}^{th}$ of the target load TL (denoted as *Tol2*). Maximum load ML is set to be $\frac{5}{16}^{th}$ above the target load TL. For example, with target load $TL = 16$, ML, ULT, and LLT are 21, 18, and 14 respectively. The heartbeat period is one second, and nodes are declared down after four seconds of no heartbeat. Parent switching decisions are made every second.

3.1 Load-latency evaluation

In the first set of experiments, 7500 nodes join the multicast at the 0th second. After the 20th second, 2500 additional nodes join at the rate of 100 joins per second. Figure 1 shows load accuracy measured as the ratio of the node's measured load to its target load. The two CDF curves correspond to the cases when there is no latency reduction ($NoLatency$), and when latency is reduced as long as load is within $\frac{2}{16}^{th}$ of the target load ($Tol2$). We see that Chunkyspread provides excellent control over load. Even with latency reduction, only 10% of nodes exceed their target by more than 10%, and virtually all nodes are within 20% of the target load.

The algorithm fully converged 95 seconds after the last join. It is to be noted that this is the time taken for the load-

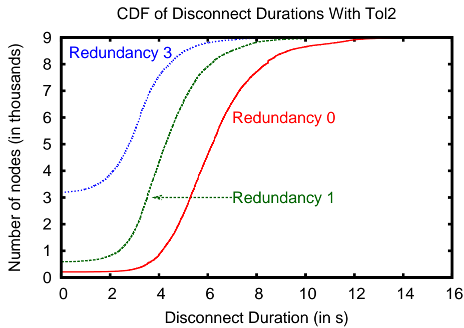

Figure 3: Disconnect Durations

latency fine-tuning algorithm to converge; more than 95% of the nodes establish parents for all slices within their first 10 seconds of joining the network. Fewer than 1% of parent switches lead to a cycle. Given the extreme churn, this speaks well of the bloom filter method of loop avoidance. Control message overhead averaged 250 parent switch messages per node over the entire *Tol2* run.

Next, we examine the performance of latency reduction. We use two parameters to characterize the latency, the network stretch and the overlay stretch. Network stretch is defined as the ratio of the actual measured latency to the network latency between the true source and the node. Overlay stretch is the ratio of the actual measured latency experienced by the node to the shortest path latency in the overlay graph between the true source and the node. The CDF of the overlay stretch is presented in figure 2. While *NoLatency* does poorly, latency reduction results in 90% of the nodes having an overlay stretch of two and more than 95% of the nodes having an overlay stretch within three. The corresponding network stretch (not shown) has an 80th percentile network stretch around seven. 90th percentile hop length is 12.

To get some intuition about how adding locality to the random graph would improve stretch, we ran a 10000-node *static* simulation where, in addition to the random neighbors selected by Swaplinks, the five nearest neighbors were added to each node's neighbor set. In this case, the 90th percentile network stretch was 5 (versus 7 without locality), a noticeable but not dramatic improvement over the pure random graph.

3.2 Robustness against Node Failures

In this experiment, 10000 nodes join at the first second of the simulation, and 10% of the nodes fail at the 50th second. Figure 3 shows the CDF of the disconnection times for the *Tol2* case, where the disconnection time is the total time when a node is not getting the full stream. The three curves in the graph show the effect of adding varying degrees of redundancy (for instance using forward-error correction). For example, the curve denoted *Redundancy 1* is the case where receiving any 15 out of the total 16 slices produces the full stream. We observe that the 90th percentile of the disconnection time when there is no redundancy is less than 10 seconds (6 seconds beyond the timeout period). We see that as redundancy is increased to 3 slices, the 90th percentile of the disconnection time is reduced to just over four seconds.

3.3 Effect of tit-for-tat

Till now, we have assumed that nodes do not lie about their loads to each other. In some environments, however, there may be free-loaders. Chunkyspread provides a natural framework for applying incentive-based constraints. To build an intuition as to how tit-for-tat may affect load and latency, we simulated a simple "weak" tit-for-tat model whereby the volume received from each neighbor must be at least within some percentage of the volume sent to that neighbor[3]. For instance, with 25% tolerance, a node that supplies 4 slices to its neighbor requires that it serves at least 3 slices back. 3-2 or 2-1 ratios are not allowed. In addition, nodes assign an initial small *credit* to new neighbors, to allow the parent-child relationships to get started, and give additional credits over time if a neighbor sends more than is received. Tit-for-tat constraints are enforced only when the credits are used up.

We tested how this simple tit-for-tat scheme works with the *Tol2* parameters. We used a 10000-node *static, homogeneous* setting in which each node has a target load of 16 slices. Each node periodically checks whether any of its neighbors is violating tit-for-tat, and withdraws uploaded slices as necessary. Only parent switches that fall within tit-for-tat constraints are allowed. We compared tit-for-tat ratios of 50%, 33% and 25% (corresponding to 1:2, 2:3 and 3:4 relationships respectively). We find that decreasing the ratio improves load balance, but at the expense of latency. For instance, the 90th percentile average overlay stretch for 50% tit-for-tat is 2, while for 33% it is 2.7. There are also longer and more frequent disconnections in the 25% case than in the 50% case. These experiments are encouraging in that they show that tit-for-tat constraints can be incorporated to an extent, though at the expense of other performance measures.

4. RELATED WORK

There has been considerable work in the past on single-tree multicast protocols. Since none of these effectively support heterogeneity, we restrict our discussion of related work to multi-path multicast protocols.

Splitstream [7] is a DHT-based multicast protocol that tries to resolve the heterogeneity problem by splitting the stream into *stripes* and sending them across multiple interior-node-disjoint trees built over the DHT. But under heterogeneous environments and node churn, there is a large number of non-DHT links formed making the system complex and inefficient [13].

Our preliminary simulations of Splitstream (using the Microsoft version), indicate that control over heterogeneous load is quite poor. Unlike Chunkyspread, which allows both a target and maximum load, Splitstream only allows the maximum load to be set. In Splitstream simulations of a 1000 node network with the maximum load values having the same range as that of the ML values mentioned in Section 3, we found that about 35% of the nodes are loaded to the maximum, while close to 10% of the nodes had zero load. All other performance measures (stretch, disconnect times, etc.) were similar to or worse than those reported for Chunkyspread. While we don't believe it is necessary to build node-disjoint paths in Chunkyspread, this constraint could be added.

Bullet [5] splits the stream into multiple blocks and uses a single tree on top of a mesh. Nodes receive only a subset of the blocks from their parents in the tree, the remaining

[3] [12] and [18] argue that strict tit-for-tat is impractical, and our simulations corroborate this.

blocks retrieved from other nodes randomly chosen using a distributed algorithm called *RanSub*. Bullet however incurs a high control overhead due to this scheme of orthogonally retrieving packets.

Chainsaw [8] is a multicast protocol that does away with trees to improve node resilience in the presence of churn. Each Chainsaw node employs a simple controlled flooding mechanism to notify neighbors of data arrivals and a pull-based approach to retrieve blocks. However, Chainsaw can potentially incur high network and CPU overheads due to per-packet notifications. We note that Chainsaw could benefit from Swaplinks for its random graph construction, and that this could make Chainsaw more heterogeneity-aware.

[11] assessed the feasibility of overlay multicast protocols supporting large-scale live streaming applications by analyzing real-world Akamai traces; using these traces along with online and offline bandwidth measurements, they concluded that real-world hosts indeed have enough bandwidth to support themselves in most cases.

Incentive-based p2p protocols try to enforce end-hosts to contribute resources. There have been many proposals in the literature that apply to file-sharing and streaming applications. Bittorrent [9] is a popular file-sharing protocol in widespread use that divides the file into multiple pieces and lets the peers download the pieces from one another. Peers employ a tit-for-tat mechanism to limit free-riding in the system. There have also been a number of proposals([19], [18]) in which peers maintain credit with each of their neighbors to enable fair sharing of resources in a content distribution network. [12] adopts a taxation model on peer-to-peer streaming multicast applications to encourage resourceful peers to contribute bandwidth to the system and subsidize for the poor peers. [15] employs a credit-based technique on Splitstream to detect free-riders. According to this scheme, trees are reconstructed periodically so that each pair of neighbors gets opportunities to donate and receive between each other on successive reconstructions. The protocol does not fully answer how to tackle heterogeneity in the system.

5. CONCLUSION AND FUTURE WORK

Chunkyspread represents a new point in the P2P multicast design space: one that has the efficiencies associated with trees and the simplicity and scalability associated with unstructured networks. At the foundation of Chunkyspread is the ability to build random sparse overlay graphs with tight statistical control over heterogeneous node degrees. This foundation, combined with a simple loop-detection mechanism based on bloom filters, provides a framework whereby different constraints and optimizations can be emphasized, depending on the application.

To date, we have focused on large-scale, non-interactive applications like the broadcast of a sporting event, at a range of volumes (text, audio, or video formats). Here, control over load is more important than latency, though in this paper we show nevertheless that significant improvements in latency can be made if load control is relaxed slightly. We also show that a certain amount of pairwise tit-for-tat can be added, though mainly at the expense of latency. Finally, we show that Chunkyspread operates well with churn.

As part of our future work, we plan to further explore this class of application. We will look at more churn scenarios. We will look at additional mechanisms for improving latency while controlling load, such as favoring large fan-out. We will look at a range of tit-for-tat mechanisms, including both social and irrational behavior. We also plan to do detailed apples-to-apples comparisons with SplitStream and Chainsaw. Beyond this, we believe that the generalized constraints-and-optimizations framework of Chunkyspread allows us to explore different types of applications and environments. These include low-latency applications, pub-sub applications (where nodes may join a large number of groups), and high-reliability applications where packets much traverse multiple disjoint paths.

6. ACKNOWLEDGMENTS

We would like to thank Kaoru Yoshida for the preliminary simulations of Splitstream. We would also like to thank M. Castro and A. Rowstron for providing us the simulator code for Splitstream.

REFERENCES

[1] V. Vishnumurthy and P. Francis. On Heterogeneous Overlay Construction and Random Node Selection in Unstructured P2P Networks. In *Proceedings of IEEE Infocom*, Barcelona 2006.

[2] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware*, November 2001.

[3] P. Francis. Yoid: Extending the Internet Multicast Architecture. *http://www.icir.org/yoid/*.

[4] Y. Chu, S.G. Rao, and H. Zhang. A Case for End System Multicast. In *Proceedings of ACM Sigmetrics*, June 2000.

[5] D. Kostic, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: High Bandwidth Data DisseminationUsing an Overlay Mesh In *Proceedings of the 19th ACM symposium on Operating systems principles (SOSP'03)*.

[6] M. Castro, P. Druschel, A. M. Kermarrec, and A. Rowstron, SCRIBE: A Large-Scale and Decentralized Application-Level Multicast Infrastructure. *IEEE Journal on Selected Areas in Commmunications (JSAC),* 2002.

[7] M. Castro, P. Druschel, A. M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. Splitstream: High-Bandwidth Multicast in Cooperative Environments. In *SOSP*, 2003.

[8] V. Pai, K. Kumar, K. Tamilmani, V. Sambamurthy, and A. E. Mohr. Chainsaw: Eliminating Trees from Overlay Multicast. In *The Fourth International Workshop on Peer-to-Peer Systems,* February 2005.

[9] B. Cohen. Incentives Build Robustness in BitTorrent. In *The First Workshop on Economics of Peer-to-peer Systems,* June 2003.

[10] E. W. Zegura, K. Calvert, and S. Bhattacharjee. How to Model an Internetwork. In *Proceedings of IEEE Infocom' 96.*

[11] K. Sripanidkulchai, A. Ganjam, B. Maggs, and H. Zhang. Feasibility of Supporting Large-Scale Live Streaming Applications with Dynamic Application End-Points. In *The Proceedings of ACM SIGCOMM,* August 2004.

[12] Y. H. Chu, J. Chuang, and H. Zhang. A Case for

Taxation in Peer-to-Peer Streaming Broadcast. In *ACM SIGCOMM Workshop on Practice and Theory of Incentives and Game Theory in Networked Systems (PINS),* August 2004.

[13] A. R. Bharambe, S. G. Rao, V. N. Padmanabhan, S. Seshan, and H. Zhang. The Impact of Heterogeneous Bandwidth Constraints on DHT-Based Multicast Protocols. In *The Fourth International Workshop on Peer-to-Peer Systems,* February 2005.

[14] D. Kostic, A. Rodriguez, J. Albrecht, A. Bhirud, and A. Vahdat. Using Random Subsets to Build Scalable Network Services. In *USENIX USITS,* 2003.

[15] T. W. J. Ngan, D. S. Wallach, and P. Druschel. Incentives-Compatible Peer-to-Peer Multicast. In *The Second Workshop on the Economics of Peer-to-Peer Systems,* July 2004.

[16] A. Whitaker and D. Wetherall. Forwarding without loops in Icarus. In *Proceedings IEEE OPENARCH*, 2002.

[17] W. A. Montgomery. Techniques for packet voice synchronization. In *IEEE J Select Areas Commun* 6(1):10221028.

[18] K. Tamilmani, V. Pai, and A. E. Mohr. SWIFT: A system with incentives for trading. In *Second Workshop on the Economics of Peer-to-Peer Systems*, 2004.

[19] A. Nandi, T. W. Ngan, A. Singh, P. Druschel, and D. S. Wallach. Scrivener: Providing Incentives in Cooperative Content Distribution Systems. In *The Proceedings of the Sixth International Middleware Conference*, November 2005.