Chunkyspread: Heterogeneous Unstructured Tree-Based Peer-to-Peer Multicast

Vidhyashankar Venkataraman
Cornell University
Ithaca, NY
vidya@cs.cornell.edu

Kaouru Yoshida
University of Tokyo
Tokyo, Japan
kyoshida@hongo.wide.ad.jp

Paul Francis
Cornell University
Ithaca, NY
francis@cs.cornell.edu

***Abstract*— The rising popularity of live IPTV has triggered renewed interest in P2P multicast. In particular, the simple and robust 'swarming' style of P2P multicast is currently favored over more traditional tree-based approaches, which are seen to be complex and fragile. Swarming approaches, however, exhibit a basic control-overhead-versus-latency tradeoff that gears it more towards high-volume, latency-tolerant applications. This paper presents a new unstructured P2P multicast protocol called Chunkyspread[1] that is tree-based yet simple and robust. Chunkyspread uses multiple trees to provide fine-grained control over member load, reacts quickly to membership changes, scales well, and has low overhead. This paper gives a detailed description of Chunkyspread and an apples-to-apples comparison with the DHT-based Splitstream multi-tree P2P multicast algorithm. We show that Chunkyspread exhibits far better control over transmit load than Splitstream, while exhibiting comparable or better latency and responsiveness to churn. This comparison establishes Chunkyspread as the 'best of breed' among tree-based P2P multicast algorithms, thus setting the stage for future comparisons with swarming-based approaches.**

I. Introduction

With the recent emergence of a number of P2P IPTV startup companies ([24], [25], [26] to name a few), P2P multicast has again become a hot topic. These products are based on a 'data-driven' or 'swarming' style of multicast similar to Chainsaw [11] or Coolstreaming [2]. In the swarming approach, each overlay node advertises to its neighbors which packets (or blocks of packets) it has received, and the neighbors explicitly request blocks as needed. This approach is in contrast with the more traditional 'tree-based' style, whereby one or more delivery trees are constructed by the overlay nodes, and packets are delivered along the trees without any explicit requests.

The primary stated advantages of the swarming approach are its simplicity and its robustness. The simplicity stems from the fact that that it requires no 'complex' distributed algorithm to build trees. The robustness stems from the fact that any neighbor can be called upon to contribute blocks of data, so the loss of any given neighbor does not cause a discontinuity in data delivery. These benefits, however, come at a cost: there is a basic tradeoff between *control overhead* and *delay*. This tradeoff is easy to see. Imagine that, in order to minimize delay, each node informed its neighbors as soon as it received any given packet so that those neighbors could request that packet as soon as possible. This clearly results in a considerable overhead. To reduce this overhead, each node instead waits until it has received some number of packets, and then advertises a bit-map indicating which packets it has. The longer the node waits, the more packets it can efficiently advertise in its bit-map. Coolstreaming [2], for instance, encodes 60 seconds worth of packets in its bit-map.

Tree-based approaches don't exhibit this control-overhead-versus-delay tradeoff. Rather, they have a *continuity-versus-delay* tradeoff. Namely, if a node's parent in the tree crashes or leaves the tree, then the node won't receive packets until it can find a new parent. If the node wishes to continuously play out packets to the application, then it must buffer enough packets to bridge the gap. The faster the tree-building algorithm can discover and fix a broken tree, the smaller this buffer has to be.

Given this difference in tradeoffs, it is premature to broadly declare one approach better than the other. To take an extreme example, a tree-based approach would clearly be superior for a low-bandwidth application with a very stable membership. A swarming approach, on the other hand, might perform better for a high-bandwidth application with significant membership churn. Furthermore, these are not the only trade-offs. Tree-based approaches can mitigate the effect of broken trees by constructing multiple trees and transmitting redundant FEC codes over some of them, which results in a *data-overhead-versus-delay* trade-off. Other criteria, such as fine-grained control over each node's load or tit-for-tat, are also important and have not been adequately studied for either swarming or tree-based approaches.

In this paper, we present a new tree-based multicast algorithm, called Chunkyspread, that is far simpler than previous tree-based algorithms. Like swarming approaches, Chunkyspread is unstructured. Like the DHT-based Splitstream [10], Chunkyspread uses multiple trees to balance load among nodes, and indeed exhibits far better control over load than Splitstream. Chunkyspread reacts quickly to membership changes, scales well, and has low overhead. Furthermore, Chunkyspread is designed such that it provides a framework for adding new performance optimizations and constraints, such as tit-for-tat.

[1]This material was partially supported by DARPA under agreement number FA8750-04-2-0011 and DARPA/AFRL FA8750-04-2-0011, and by the National Science Foundation under Grant No. 0338750.

We believe that the relative simplicity and good performance of Chunkyspread makes it a viable candidate for P2P multicast. While ultimately this requires detailed and thorough comparisons with swarming approaches, this paper focuses on comparison with Splitstream. Doing so allows us to answer the question of finding the best tree-based P2P multicast algorithm, thus paving the way for later comparisons with swarming approaches.

This paper makes the following contributions:

1) We give a detailed description of Chunkyspread[2], the first unstructured P2P multicast algorithm with fine-grained control over load.
2) We present a thorough simulation analysis of Chunkyspread's load control, latency optimization, responsiveness, and overhead. Using the MSPastry simulation of Splitstream, we present an analysis of Splitstream over the same metrics, and compare Splitstream with Chunkyspread.
3) Again through simulation, we present preliminary and limited analysis of Chunkyspread for the basic trade-off of buffer size, data redundancy, and packet loss in the face of churn.
4) We present limited results of a complete implementation of Chunkyspread running on Emulab. These results partially validate our simulation results.

This paper is organized as follows. Section II describes our approach in detail. Section III gives an overview of the existing multi-tree approach, namely Splitstream. Section IV presents evaluations of both Chunkyspread and Splitstream, Section V discusses related work in our area while Section VI concludes the paper and presents future directions to our work.

II. Protocol description

We start with a high-level overview of Chunkyspread, followed by a detailed description of its various components.

Chunkyspread constructs a single-source multicast group among a set of member end-systems. We haven't yet provided support for multiple senders but it is a simple extension, and we do not discuss it any further in the paper. Like Splitstream, the source (which we call the *true source*) transmits the multicast stream as M distinct slices. Each slice is transmitted over a separate multicast tree. But, quite unlike Splitstream, the trees are not necessarily node-disjoint; as we explain in a later section, node-disjointness is a property hard to achieve even in Splitstream especially in heterogeneous environments.

Applications can access Chunkyspread through an API that provides *join()*, *quit()*, *send()*, and *receive()* primitives, typical to any multicast protocol. Of particular interest is the *join()* primitive that takes the following parameters: the group name, the member type (true source or receiver), the target load, and the maximum load. The two load parameters refer to the transmit load of a member, and may be expressed by the application as absolute throughput values (e.g. 100Kbps), or as a percentage of the stream volume (e.g. 75% or 250%). The maximum load is the absolute maximum volume that the member[3] will transmit at any time while the target load is the volume that the member would prefer sending at steady state. The expectation is that the steady state volume sent by the application will be near the target load. Of course, the application should choose its maximum load judiciously and there should be enough capacity in the system to transmit the stream. No P2P multicast system can operate otherwise. While an application could adapt its target load dynamically to the current network congestion and competing traffic, but we have not included this capability. Chunkyspread internally expresses load in units of the number of slices, and not bandwidth or percentage of stream volume. Chunkyspread uses the following parameters: the number of slices *M*, the latency threshold, minimum node degree *MND*, and minimum load *MinL*. These might be set by the true source and communicated to all members. We will postpone the discussion on the last two parameters to later in this section.

The default value for *M* that the stream is split[4], is 16. The *latency threshold* is a value that determines how the system should weigh the trade-off between achieving target load and minimizing latency. It is expressed as a percentage of the target load. For instance, assume that a given Chunkyspread application requests a target load of 100%, and that $M = 16$ and the latency threshold=10%. 10% above and below 16 slices is 18 and 14 slices respectively after rounding to the nearest slice. The lower edge of the range (14 slices in this case) is called the *Lower Latency Threshold LLT* while the upper edge is called the *Upper Latency Threshold ULT*.

Given the *LLT* and the *ULT*, load balancing and latency reduction work as follows. As long as a given member node's load is outside this range, the system adjusts to move the load within the range. If a node X's load is below its *LLT*, other nodes will try to become a child of X, thus increasing X's load. If X's load is above its *ULT*, existing children of X will try to find other parents, thus decreasing X's load. Once nodes' loads are within the *LLT-ULT* range, they will no longer try to improve load, but rather try to optimize latency. Whenever a change of parent for a given slice improves latency by a certain margin without causing the load to fall outside this range, that change is made. From this, we can see that a larger *LLT-ULT* range will improve latency at the expense of nodes not getting as close to their target load, while a smaller range has the opposite effect.

To join a Chunkyspread multicast group, nodes must first contact a rendezvous node at a well-known location (DNS name or IP address). This rendezvous node must know of at least one existing member of the multicast group. This style of joining a P2P group is a fairly standard practice, and not further discussed here.

Once a joining member node or the true source finds at least

[2]In [1], we presented an overview of Chunkyspread and some preliminary simulation results.

[3]Note that the term member refers to receiving members only, not the true source. We use the terms member and node interchangeably.

[4]We experimented with more and less and this value gave a satisfactory load control as well as an acceptable overhead.

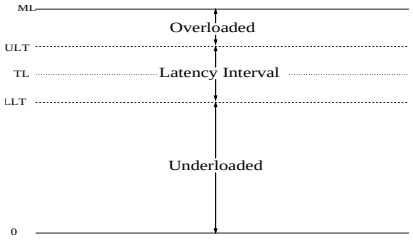

Fig. 1. The load-latency thresholds

one existing node, it participates in a continuously running distributed algorithm called Swaplinks [5] that produces a random graph among all nodes using simple weighted random walks. This random neighbor graph is the underpinning of Chunkyspread in much the same way as RanSub [16] is the underpinning of Bullet. Swaplinks is able to statistically control the node degree of each node, and Chunkyspread exploits this to give nodes with higher target loads proportionally higher node degrees. The idea here is that nodes with higher load should have more neighbors to transmit slices, and nodes with lower load should have proportionally fewer neighbors. With network churn, the neighbor set of each node changes, but the number of neighbors stays roughly the same. In addition to these random neighbors, nodes may discover other nodes that are nearby with respect to latency. These nodes may be added to the neighbor set to improve latency.

This is where the system-wide parameters *minimum node degree MND*, and *minimum load MinL* come into play. *MND* is the smallest node degree in the random graph that any node may have. Its default value is 8, and as far as we know, this value is universally appropriate. Since node degree is set proportionally to the target load, the node degree of any nodes is set to be *ND = max[8, (TL/MinL)*MND]*, where *TL* is the target load. As with ensuring that a given Chunkyspread group has enough capacity, the application must also ensure that *MinL* is set to an appropriate value: i.e., the expected smallest capacity of a host in the system. It may also be possible to set *MinL* dynamically, for instance by having nodes remember the lowest *TL* they've seen in the network, and setting *MinL* accordingly. We have not explored this possibility.

Unlike the receiving nodes, the true source discovers exactly *M* (the number of slices) neighbors with the help of Swaplinks. The true source transmits one slice to each of these neighbors. These neighbors become the roots of *M* multicast trees, and are called the *slice sources*. When the true source discovers the failure of a slice source it selects a new random node as the slice source. Note that a node may be a slice source for more than one slice.

A node, upon joining the random graph, tries to find a parent for each slice without forming a loop. We avoid and detect loops using bloom filters ([18]) in the data packets. In selecting parents, each node tries to maintain a set of constraints, as well as its performance goals and those of its neighbors. The performance goals we have implemented and studied in this paper are target and maximum load, and latency, as described above. Other constraints may include tit-for-tat and path-disjointness.

The basic process is straightforward. Each node lets its neighbors know initially about its *LLT*-*ULT* range and its maximum load (*ML*). Further, each node periodically advertises to all of its neighbors the following: its bloom filters for each slice, information about the arrival time of each slice, its current load (i.e. the number of children it has). Additional performance constraints may be added to this list. Each node takes this information into consideration to determine which neighbors would make appropriate parents for each slice. As conditions change, for example, due to churn, or due to load changes, nodes may select different neighbors as parents for each slice. Note that as a result of this process, a neighbor may be the child for some slices, and the parent for others. Figure 1 shows the thresholds used by Chunkyspread in fine tuning the load and latencies in the trees.

Given this overview, the following subsections provide additional detail.

Loop avoidance and detection: Bloom filters offer a spatially efficient method to detect and avoid loops, with a tunable rate of false positives[18]. Each node selects a bloom mask with an appropriate number of bits. A node, before forwarding a data packet, adds its bloom mask to the bloom filter that is tagged along with the data packet. Loops are avoided by having nodes advertise the bloom filters they receive for every slice to their neighbors. A given node does not select a neighbor as a slice parent if the node itself appears in the neighbor's received bloom filter.

Loops are detected immediately by the first packet that traverses the loop[5]. This packet can either be a data packet sent by the application, or, in the absence of such packets, a probe packet transmitted by a node to its children. The first node to detect the looping packet drops it and immediately selects a new parent.

Fine-tuning Load: As described above, each node periodically checks to see if it has an overloaded parent (above the parent's *ULT*), and an underloaded neighbor (whose load is below *LLT* and satisfies the loop-free condition), and if so attempts to *switch* parents. Since multiple nodes are doing this at the same time, multiple potential switches may be possible. To encourage only the best such switches take place, each node with a potential switch informs its overloaded parent of the loads of all (or a subset of the most) 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. In our system, the overloaded parent usually picks one amongst a set of good candidates so as to avoid implosion of switch requests to such nodes.

The potential parent accepts or rejects the request from the child node depending on its load and its bloom filter for that slice (these parameters may have changed from the time

[5]A loop can happen in spite of maintaining a bloom filter. A node that is not yet aware of a bloom filter change in its ancestors, can accept one of the ancestors as its child.

since the child had made the request). If the switch request is accepted, the child informs the previous parent of the switch completion.

The switch messages that the child sends to its future and the current parent, identify the sequence number of a future data packet at which the current parent should stop transmitting, and the new parent should start. This minimizes packet loss or duplication during the switch itself.

It is important to note that, in the absence of churn and switches due to fine-tuning latency, the algorithm for balancing load will converge. Every load balancing switch results in a node above *ULT* reducing its load and a node below *LLT* increasing its load. Once within the *LLT-ULT* range, there are no load-balancing switches that can push a node out of that range, and no load-balancing switches take place between nodes already in the *LLT-ULT* range. The period when the load-balancing switches take place predominantly in a node is called the load-phase of the algorithm.

Fine-tuning Latency: Once all of a node's parents are within their *LLT-ULT* range, the node looks for parent switches that can improve the latency with which it receives packets while keeping loads within the *LLT-ULT* range. This constitutes the latency phase of our algorithm. 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 slice *relative to other slices*. 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 within range). Note that nodes only make such switches if the expected improvement in latency is beyond a certain threshold. The latency measure described above should be calculated as a moving average to smooth out transient changes due to congestion.

We have not used the overlay path length as a measure for latency reduction for obvious reasons: small path lengths do not necessarily yield low latencies, especially if the underlying graph is locality-aware. A smaller path length does, however, mean that the packet has to traverse fewer nodes which reduces the chances of disconnections in the path. If this is desired, path length can be used as a metric for parent selection.

Initial Tree Construction and Forced Parent Selection: In Chunkyspread, new trees must be 'kick-started' when the true source first starts the multicast stream or when a slice source quits and the true source chooses a new one. Initial tree construction involves a simple controlled flooding mechanism similar to the one used in Chainsaw. Shortly after a node starts receiving flooded packets for a given slice, it selects a parent from among the neighbors from which it received the flooded packet and the parent accepts the node if its load has not exceeded *ML*.

Apart from this, a node may join a multicast session whose trees have already been constructed through the flooding mechanism described above. In that case, the node periodically requests its neighbors to be parents for each of its slices until it finds them. As a result of these cases, the parent's load may exceed the upper latency threshold *ULT*. Normally, the ongoing load balancing process will bring the load back to or below *ULT*, though on the rare occasion a node's load may stay above *ULT* for a period of time due to the lack of availability of potential parents for its children (though there may be underloaded nodes elsewhere in the system).

There are three other cases where a node may request a parent even though doing so pushes the parent's load above its *ULT*. All three are cases where the node is forced to change its parent. This may happen when a loop is detected, when the parent quits the group, and when the Swaplinks algorithm changes the neighbor set as part of its normal operation[5]. While the first two is effectively a temporary disconnection from the tree, the third is usually similar in effect of any normal switch. Note that a node may only reject a request to become a parent if doing so pushes its load above *ML*, or it does not satisfy the looping constraint (and any other constraints where suitable).

III. Overview of Splitstream

Since we make simulation comparisons of Chunkyspread and Splitstream, a brief overview of Splitstream is provided here. Splitstream builds multiple trees on top of Scribe, a single-tree multicast protocol that constructs its tree using the overlay routes of the underlying DHT (Pastry). However, a node may not have enough capacity to serve all its in-neighbors that want to join the multicast group. In order to avoid nodes getting loaded beyond their capacities, Scribe resorts to two other mechanisms, namely pushdown and anycast operations. When a fully loaded Splitstream node is requested to parent another node, it may preempt one child node for another based on ID constraints [15]. The resulting orphaned node recursively contacts the parent's descendants (called *pushdown*) to find a parent and if it still cannot find one, *anycasts* to the group of nodes that have spare capacity.

Splitstream works well in homogeneous cases with usually the Pastry neighbors serving the nodes. However, in heterogeneous environments, the pushdown and anycast operations happen more often and this leads to frequent disconnections of nodes. The two operations lead to the formation of parent-child links that are apart from the underlying Pastry neighbors. Hence, Splitstream starts losing the benefits of the underlying DHT as the number of non-Pastry neighbors increases. In short, Splitstream prefers ID-based constraints over load constraints when initially creating the tree and this leads to further complications in the tree-building protocol.

IV. Results

We have performed a series of experiments on a packet-level, event-driven simulator coded in C++. We have also implemented the system and made some simple deployment experiments on Emulab. The default number of member nodes

in each simulation is 5000. The Chunkyspread simulation could operate with more nodes and higher join rates than the ones specified in the paper, but the Splitstream simulator could not, so we limit our simulations to 5000 members. To calculate the latencies between members, we placed member nodes at random edge locations on GT-ITM network topologies having 5050 routers [12], and set delays proportional to the distance metric of the resulting topology. We assume that control messages are sent over TCP, and so ignore message loss in our simulations.

The random overlay is constructed using a packet-level trace file generated offline by a Swaplinks simulator. The trace file allows us to determine the delays associated with the neighbor selection in Swaplinks. The trace file was used in order to avoid running the random neighbor selection as part of the simulator, hence making the simulations faster. To further scale the simulations, the simulator does not explicitly generate data packets.

Member nodes in the simulation receive all slices[6]. The default number of slices in our simulations is $M = 16$. To represent heterogeneity in upload, each node is assigned a *total degree* selected randomly between 8 and 50. This represents a moderate level of heterogeneity, representing say a population of users behind dial-up modems and broadband, or behind broadband and T1. The upload for each node is then calculated as the number of slices per stream times the ratio of the node's degree to the average degree of the network. Each node chooses $ML = (1.5)TL$ so that there is enough upload capacity in the system to supply the full stream to all the nodes.

We experiment with two settings for the *LLT-ULT* range. One is when there is no latency range (*i.e., ULT=LLT=TL*), resulting in no latency optimizations whatsoever. This is denoted *Lat0*. In the other, they are set to *2(TL)/16* slices from *TL* (rounded up for *ULT*, and down for *LLT*). In other words, if *TL=16*, then *LLT=14* and *ULT=18*. This is denoted *Lat2*.

We chose a bloom filter size of 128 bits and a bloom mask size of 6. This yields a false positive rate of 0.25% after insertion of 10 keys. The heartbeat period is set to 1 second and the timeout period to detect a node failure is set to 4 seconds. Parent switching decisions are made every second.

For the Splitstream simulations, we used a simulator coded in C# that was provided to us by Miguel Castro. The simulations are run over the same GT-ITM synthetic routing topology as used in Chunkyspread simulations and have 16 slices. Unlike Chunkyspread, Splitstream provides a single parameter, the maximum load (*SML*). *SML* is analogous to Chunkyspread's *ML* in that the load never exceeds *SML*. It is unlike Chunkyspread's *ML*, however, in that a Splitstream node may easily settle on a sustained transmission rate of *SML*, whereas a Chunkyspread node may temporarily transmit at *ML*, but will quickly move towards the *LLT-ULT* range. As a result, we need to interpret *SML* differently from *ML*, and an apples-to-apples comparison is not really possible.

Because of this difference, in one case we treat *SML* to be equivalent to *ML* (denoted *SS(1.5)*). That is, we set it to be 50% above the number of slices (*SML=1.5TL*) where *TL* is the target load for the corresponding nodes in Chunkyspread. In the other case, however, we try to treat *SML* as though it were equivalent to *ULT*. As such, we set *SML=(1.2)TL* to compare with *Lat2* (denoted *SS(1.2)*). To compare with *Lat0*, we tried setting *SML=TL*, but Splitstream does not converge in this case, so instead we use *SML=(1.1)TL*, denoted *SS(1.1)*. Splitstream has a time-out parameter that determines how long a node should wait for the result of an anycast operation before trying again. This parameter is set to 4 seconds. A value less than this tended to result in too many unnecessary anycast operations.

We have broadly considered four scenarios to evaluate our protocol. The *static* scenario is when all overlay nodes are already part of the random graph and the tree building starts from the first instant; this is useful in analyzing the load-latency algorithm without any churn. The *join* scenario happens when there are 3750 overlay nodes already in the network and the rest (1250 nodes) join at a rate of 50 joins per second from the 20^{th} second by which time most of the originally present nodes have reached a steady state. This scenario is more realistic and can possibly be a live event that attracts a large audience within a short span of time. The *bursty* scenario is the pathological case when a certain percentage of the nodes fail at the *same time instant*; this is helpful in analyzing the robustness of the protocol against node failures. To understand the effect of more realistic scenarios on our protocol, we simulated Chunkyspread under *continuous churn* in which nodes join and leave at the same time. The Swaplinks simulator did not have functionality provided for locality-awareness. To determine the effect of adding locality to the random graph, in addition to the random neighbors selected by Swaplinks, some number of nearest neighbors were added to the neighbor set of each Chunkyspread node only for the simple static scenarios.

The static and the join scenarios We first present a comparison study between Splitstream and Chunkyspread followed by an evaluation on the convergence and the control overhead of Chunkyspread.

1) Comparisons with Splitstream: In the first set of experiments, we analyze the tradeoff between load balance and latency in Chunkyspread and compare them with Splitstream. We introduce the term *excess load percentage* to quantify load in the protocols. It is defined for every node as follows.

$$Excess\ Load\ Percentage\ =\ \frac{Node's\ Load\ -\ TL}{TL}\% \quad (1)$$

This parameter quantifies how close nodes reach their target load and hence the degree of fairness provided by the protocol. A value of 0% implies that the node has perfectly reached its TL, while a value of -100% means that the node has zero load. The maximum value of this parameter is bound by $\frac{(ML-TL)}{TL}\%$

[6]In principle, it would be possible for nodes to receive some fraction of the slices and still be able to reproduce the stream, for instance, by using Multiple Description Codes[20]. We neither implemented nor simulated this.

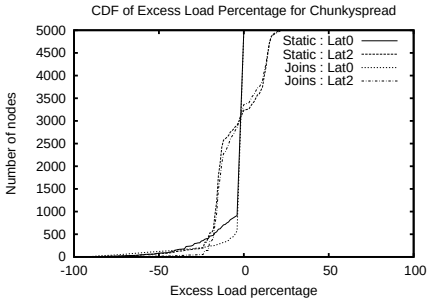

Fig. 2. Load distribution in Chunkyspread

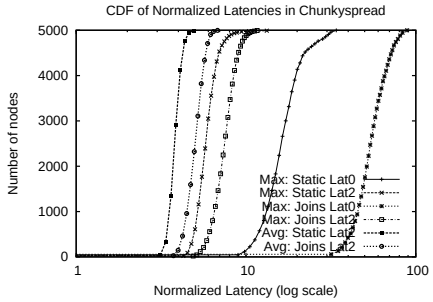

Fig. 3. Maximum and average latency distribution in Chunkyspread

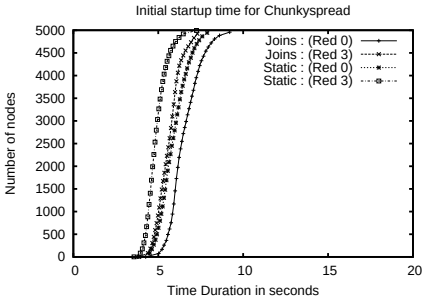

Fig. 4. Initial Startup time in Chunkyspread

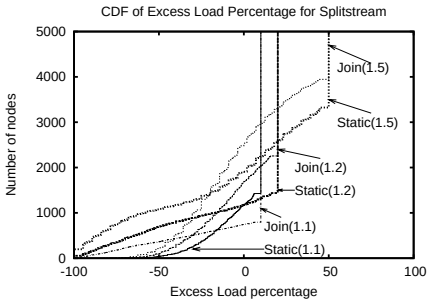

Fig. 5. Load distribution in Splitstream

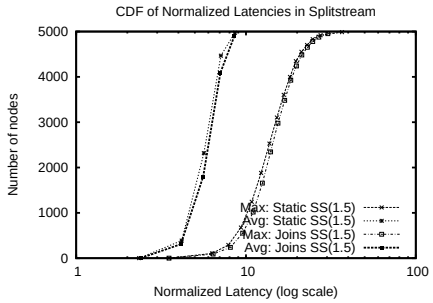

Fig. 6. Maximum and average latency distribution in Splitstream

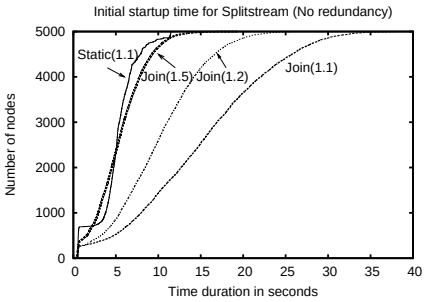

Fig. 7. Initial startup time in Splitstream

which is 50% in our Chunkyspread simulations.

We use two parameters to evaluate the latencies: the maximum and the average overlay latencies over the slices obtained at each node. The latencies are normalized with respect to the median value of the network latencies between overlay nodes. We chose not to use the *network stretch*[7] parameter to evaluate our latencies. Network stretch may not give a true picture of what the latencies are: for example, a high network stretch could actually be due to high latency or could be due to a low network latency with the true source.

Figure 2 shows the cumulative distribution function (cdf) of the excess load percentage of nodes in Chunkyspread after steady state was reached. We observe that *Lat0* performs quite well in both the static and the join scenarios: more than 80% of the nodes reach exactly their *TL* in the static scenario while around 90% of the nodes reach their *TL* in the join scenario. With the latency phase added, Chunkyspread still performs well: almost 90% of the nodes are within 25% of their *TL* values in the *Lat2* case in both the join and static scenarios. The maximum fraction of excess load that any node reaches is about 20%. Apart from the good load balance, we observe comparable performances of the join and the static cases which indicates that the protocol can function at high join rates as well as in cases without any churn at all.

Figure 3 shows the cdf of the maximum and average overlay latencies normalized with the median of the network latencies between nodes in the network. The x-axis is shown in log scale. The cdfs have been plotted for the *Lat0* and the *Lat2* cases. We first observe that *Lat0* yields very high latencies in both the static and the join scenarios, which is expected since *Lat0* is completely 'latency-blind'; this can be seen from the maximum latencies of *Lat0* in both the static and the join cases. We observe significant improvements in latencies with *Lat2*. The 90^{th} percentile values of the maximum latencies in both the static and the join cases are around 7 and 9 respectively while the same for the average latencies are around 4 and 6 respectively. The difference between the maximum and the minimum latency values incurred by a node across the sixteen slices gives us an idea of how long it takes to receive all the slices needed for reconstructing the packet and hence the size of the application buffer required while waiting for the slices. We note that the latency for any slice experienced by a node is bounded below by its network latency to the true source. Then, for example, if we assume the median network latency were around 50 milliseconds, then a 500 millisecond buffer is necessary to successfully play out the stream *in the steady state* even if losses due to factors such as churn or congestion are not considered. We also found that 90% of the nodes experience average latencies of at most twice the minimum overlay path length between the true source and the nodes for *Lat2* which again indicates its effectiveness.

Figure 5 shows the cdfs of the excess load percentage values for SS(1.1), SS(1.2) and SS(1.5) for each of the join and static cases in Splitstream. As expected, a considerable number of nodes get saturated to their *SML* values and the percentage of such saturated nodes increases as $\frac{SML}{TL}$ values decrease. For example, the percentage is 35% for SS(1.5), 60% for SS(1.2)

[7]A common term used in the literature that is defined as the ratio of the measured overlay latency to the network latency between the true source and the node

and 85% for SS(1.1) in the join cases. This is in stark contrast to the excess load percentage distribution that Chunkyspread's *Lat2* and *Lat0* yielded. We also find that the join case has a worse load balance than the static case, since the newly joined nodes are not provided enough opportunities to supply the slice unless an orphaned node or another newly joined node requests for a slice. In Chunkyspread, the load fine-tuning algorithm ensures the newly joined nodes also participate in supplying the slices.

The graph in Figure 6 shows cdfs of the average and maximum latencies in the static and the join cases of Splitstream. We note that both the average and the maximum latencies showed very marginal improvements as $\frac{SML}{TL}$ was increased with both the static and the join scenarios performing comparably. The comparable performances show that curbing the spare capacities do not have a significant effect on the latencies. We have presented only SS(1.5) here for clarity. The 90^{th} percentile values of the average latencies for both the static and the join scenarios are close to 8; this is greater than Chunkyspread's $Lat2$ values but still quite comparable. However, the maximum latencies show really high values. SS(1.5) yields 90^{th} percentile values of around 20 in both the static and the join scenarios; it also displays a heavy tail, almost reaching 30. These are in fact comparable with (static) Chunkyspread's *Lat0* values. The reason for the high maximum latencies is that with heterogeneity, more (random) non-DHT parent-child links are formed which are not necessarily latency-optimized unlike their DHT counterparts. The huge difference between the average and the maximum latencies requires an application buffer of considerable size and this buffer is to just make up for the delays in the slice arrivals for the same stream. In the example that we had considered for Chunkyspread above, Splitstream nodes may require a 1.5-second buffer in the steady state just to counter losses due to late arrival of slices.

We observe a similar trend with the maximum hop length (from the true source) at each overlay node. While Chunkyspread's *Lat2* yields a 90^{th} percentile hop length of around 8 in both the join scenarios, Splitstream's values are as high as 30. This reflects poor resilience in Splitstream's trees.

We define the *initial startup time* of a node as the time taken since its joining the multicast session, for it to start receiving the entire stream. While this quantity is clearly defined in Chunkyspread, it is not in Splitstream, since a node that has started to receive its stream from all its trees can potentially get orphaned from one or more trees. Hence, we include all the time durations during which nodes are disconnected from the tree due to such preemptions, into the initial startup time. Note that the disconnection due to orphaning a node will lead to disconnections of its descendants in that tree, if any.

Figure 4 shows the cdf of the initial startup time for Chunkyspread. We find that the 90^{th} percentile value in the join scenario is about 8 seconds while it is 7 seconds in the static scenario. The reason for the difference is the fact that the static scenario is run with locality which enables faster tree construction. In the graph, *Red 3* denotes the case where the stream is encoded with 3 redundant slices, meaning it is enough if the node gets any 13 out of the 16 slices to obtain the full stream. We find that in the static case, the 90^{th} percentile value for *Red 3* is less than 6 seconds.

Figure 7 shows the initial startup times of Splitstream. As claimed in [10], the system performs well in the static case with even SS(1.1) yielding a 90^{th} percentile value of around 8 seconds which is comparable with Chunkyspread's values. Expectedly, as spare capacities decrease, performance worsens. SS(1.5) performs comparable to Chunkyspread in the join scenario, with a 90^{th} percentile value of around 9 seconds. But with decreasing $\frac{ML}{TL}$ values, the startup time shoots up to 17 and 26 seconds for SS(1.2) and SS(1.1) respectively in the join case. The reason for the poor performance is that as nodes join, many of the existing nodes have already been saturated to their *SML* values and the newly joined nodes result in more anycast and pushdown operations. We note that with Chunkyspread, the load fine-tuning algorithm ensures that the spare capacities are distributed across nodes even when nodes are joining at a high rate.

2) Time to convergence: The convergence time is defined in this paper as the time taken till the last switch is successfully completed. We noted for every node the last time instant that it had completed a switch in the system. We observed that *Lat0* converged quite well in both the static (18 seconds) and the join (70 seconds) scenarios. We also saw that while *Lat2* converged within 60 seconds in the static scenario, it took around 120 seconds to converge in the join scenario, which was 75 seconds after the last join took place. In contrast, Splitstream reaches a steady state as soon as the last orphan node gets a parent. Hence its convergence time is actually the startup time that we discussed earlier.

Figure 8 shows the excess load percentage per node as the simulation proceeds in the case of *Lat2* for the join scenario. The maximum and the 95^{th} percentile curves peak to *ML* during the first 10 seconds of the simulation after which the algorithm brings both the curves down to within the target upload interval in the next few seconds. The second peak arises after nodes start joining and stays till 10 seconds after the last node had joined the network. Though there are nodes saturated to their *ML* values (50%) during this time period, the 95^{th} percentile and the median curves are close to the target loads (30% and 10% respectively) which show that there is a considerable number of nodes with spare capacity that can serve a newly joined node quite fast.

Figure 9 shows the normalized average latency over the slices of nodes as the simulation proceeds in the static scenario. We observe that the load phase of the algorithm shoots the latency up initially, but then, the latency phase of the algorithm steadily brings it down. The peaks in the 95^{th} percentile curves of the average and the maximum latency values show that Chunkyspread may need to maintain an application buffer of a considerable size for the temporary period of time when the load phase of the algorithm is more dominant than the latency phase; such cases happen after

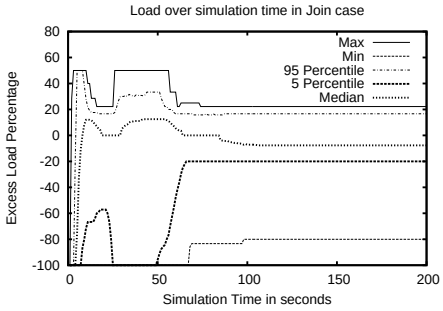

Fig. 8. Load of nodes over the simulation time in join case (Lat2)

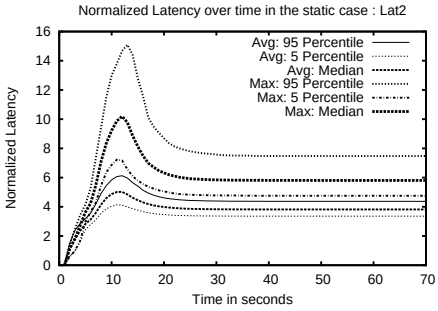

Fig. 9. Maximum and average latencies over the simulation time in the static case

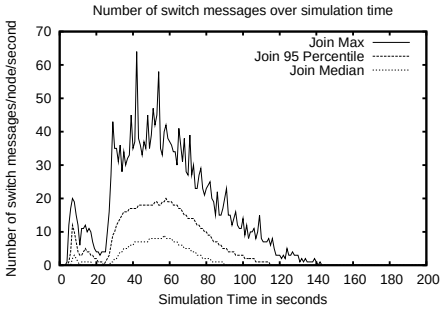

Fig. 10. Control Overhead in Chunkyspread: Switch messages over simulation time

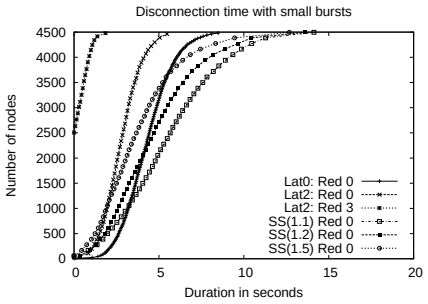

Fig. 11. Recovery duration for 10% burst

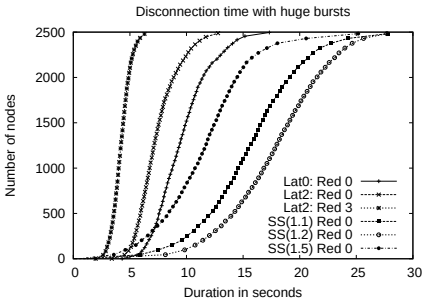

Fig. 12. Recovery duration for 50% burst

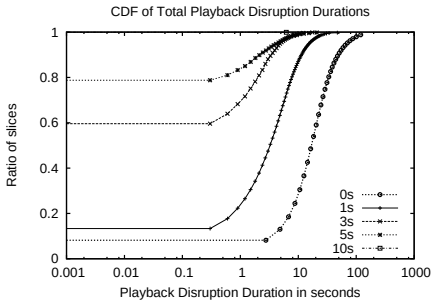

Fig. 13. Total playback disruption duration across slices for various buffer sizes

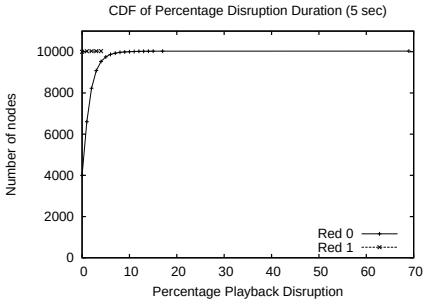

Fig. 14. Percentage playback disruption duration with 5s buffer

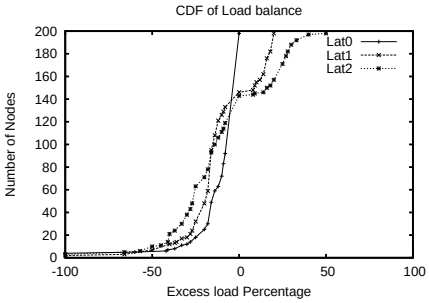

Fig. 15. Emulation result: CDF of Load Distribution

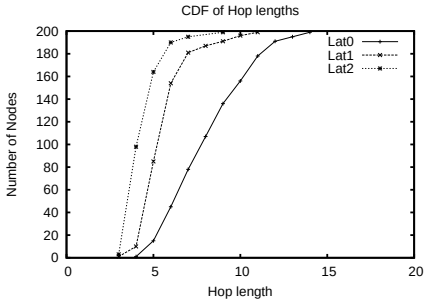

Fig. 16. Emulation result: CDF of Hop lengths

there is churn or after the true source kick-starts the multicast session.

3) Control Overhead: Next, we evaluate the the control overhead in the form of switch messages incurred by nodes in the network. Periodic advertisements constitute an average overhead of 29 messages per second as per the degree distribution and the heartbeat frequency that we have chosen. Figure 10 shows the number of switch messages sent per node per second over the simulation time of 200 seconds when *Lat2* is run in the join scenario. The peaks correspond to the time when nodes are joining the system and also after the true source kick-starts the multicast session. Though the dominant peak value of the maximum number of switch messages sent by any node is 60, the peak values of the 95^{th} percentile and the median are about 20 and 8 messages per second per node respectively. This indicates a modest overhead amongst Chunkyspread nodes even at a high join rate. Apart from this, we observed that around 50% of the switch messages sent during the joining phase account for failed switch requests.

Bursty failures: To quantify data losses due to node failures, we measure the time during which nodes are disconnected from one or more trees. We measure the *recovery duration* for each node, which is defined as the time duration calculated from the instant nodes *detect* failures of their neighbors till they get connected back to the trees. It is to be noted that during the recovery period, nodes are disconnected from the tree and so are its descendants. Hence, while a node is trying to recover from a parent's failure, this duration that its descendants are disconnected also gets accounted to the descendants' recovery duration (since an ancestor is trying to recover on their behalf).

Figure 11 shows the cdf of the recovery duration when 10% of the 5000 nodes fail at the 30^{th} instant, at various levels of slice redundancy. We find that both the protocols recover

quite fast with 90^{th} percentile values of about 5 seconds and 8 seconds in *Lat2*, and SS(1.2) respectively. *Lat2* performs better than *Lat0* primarily because the former yields lesser hop lengths which, as mentioned before, leads to better resilience. On adding redundant slices, we find a drastic improvement in the recovery times. For example, with a redundancy of 3 slices, more than 50% of the Chunkyspread nodes are not disconnected at all and the maximum recovery duration is around 2.5 seconds. The maximum control overhead experienced by any Chunkyspread node is 42 messages per second per node just after failures were detected while the median value is just 12 messages per second per node during this time.

Splitstream performs worse than Chunkyspread when 50% of the nodes fail at the same instant. Figure 12 shows the recovery duration in such a scenario. With *Lat2*, the 90^{th} percentile recovery time is 10 seconds while it is at least 15 seconds for Splitstream. When redundancy is added, there is a good improvement in the recovery duration: the 95^{th} percentile value for Chunkyspread is just 5 seconds in the case when 3 redundant slices are added. This just goes to show Splitstream's inability to handle a huge failure burst. The problem, we suspect, is the high hop length that Splitstream incurs, which affects its robustness to node failures.

Churn scenario: The more realistic churn scenario that we have studied is similar to the one tested in [15]. We consider Poisson arrivals at 10 joins per second, and pareto stay times with a minimum duration of 90 seconds and a mean of 300 seconds (which implies the pareto parameter $\alpha = \frac{10}{7}$). Pareto is a heavy-tailed distribution which is typical of the behavior of users in such environments[13]. The churn happens for the first 1000 seconds after which the remaining live nodes are allowed to settle down for the next 200 seconds.

The disconnection time intervals are noted at every node for every slice; these are the time intervals when the node is disconnected from the slice tree due to an ancestor's failure. After obtaining the disconnection durations at every slice, we simulated an application playback buffer offline for each slice at every node to calculate the duration when there is no playback. This parameter is called the playback disruption duration. Figure 13 shows the cdf of the total playback disruption duration at every slice of all nodes for various buffer sizes. With no buffer at all (which corresponds to the 0 second buffer size), we find that the 90^{th} percentile value is 20 seconds and this value decreases steadily as the buffer size is increased. For example, with a 5 second buffer size 85% of the slices are not disrupted at all and the 90^{th} percentile disruption duration is 1 second. From this graph, we infer that most of the disruptions are of short duration and can be recovered using a buffer of modest sizes[8].

To better show this fact, we observe the cdfs of the percentage of disruption duration over the lifetime of nodes in the system, for various levels of redundancy in Figure 14. For example, at a redundancy of 1 slice, a node is said to be disrupted if its playback buffers are disrupted for at least two slices. With no buffer at all (not shown), almost 60% of the nodes are disrupted at the first slice for more than 60% of the time. But as more redundant slices are added, we find that the disruption percentage decreases. In particular, with a redundancy of 4 slices, 90% of the nodes are not disconnected at all. Further, with a 5-second buffer, we find that no node (barring the heavy tail) is disrupted for more than 10% of its lifetime, as can be observed in Figure 14. From these graphs, we observe the tradeoff between the buffer size, redundancy and the playback disruption duration, which is fundamental to any streaming protocol.

Emulation: We have also made small deployment experiments in Emulab and have tested our protocol on a cluster of machines. The system was tested on 200 nodes emulated on a set of 50 machines, with the delays obtained from a 100-router transit-stub graph. A 100 Kbps stream was split into eight 12.5 Kbps streams and sent across multiple trees. The stream was multicast by the true source after it received its first set of 8 neighbors. As a first step, we have used hop length as the latency reduction parameter. The system was run for 20 minutes and a snapshot of the data was taken at the 10^{th} minute. We chose a moderate level of heterogeneity with the degree distributed uniformly between 8 and 40 neighbors. Figures 15 and 16 show the load distributions and hop lengths for the *Lat0*, *Lat1* and the *Lat2* cases. The trends in the graphs are quite similar to the ones that we had obtained in our simulations.

Effect of tit-for-tat constraints and other parameters: We have conducted preliminary simulations on the case when tit-for-tat constraints are applied in the system, the results of which have been reported in [23]. We also observed the effect of altering parameters such as the number of slices (beyond 16 slices), degree of heterogeneity and the number of neighbors on the protocol. We largely observed that these parametric changes do not result in significant changes to the protocol performance. More details can be found in [23].

V. Related work

There has been considerable work in the past on single-tree multicast protocols[7], [6], [21], [9], [22]. Since none of these effectively support heterogeneity, we restrict our discussion of related work to multi-path multicast protocols.

Bullet [8] 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 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 [11] and Coolstreaming [2] are swarming-style data-driven multicast protocols that do away with trees to improve resilience. Each overlay node (proactively or reactively) notifies neighbors of data arrivals and employs a pull-based approach to retrieve blocks. Though Coolstreaming has been

[8]The heavy tail in the graph was due to one particular slice of a node for which it was not able to find a parent as the bloom filter condition yielded false positives for the parents which could have supplied the slice. An obvious solution to prevent this from happening is to either request for more neighbors or join all over again.

used in the Internet for TV broadcasts, much remains unknown about its performance, for instance its delay characteristics or its control over load.

[13] assessed the feasibility of overlay multicast protocols supporting large-scale live streaming applications by analyzing real-world Akamai traces; the work concluded that real-world hosts indeed have enough bandwidth to support themselves in most cases. [3] describes a probabilistic scheme to improve resilience in tree-based multicast, according to which each node apart from the usual tree forwarding, probabilistically forwards data to a random node in the overlay. [15] points out the limitations in the applicability of Scribe in heterogeneous environments especially with respect to its anycast and push-down operations. [4] uses trace-based simulations to show that placing nodes with desirable properties higher up in the trees can improve the performance of tree-based multicast protocols.

VI. Conclusion and Future work

Chunkyspread represents a new point in the P2P multicast design space: one that has the efficiencies associated with tree-based multicast and the scalability and much of the simplicity associated with swarming-style multicast. 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.

While our results so far are promising, what is needed is a broad and deep comparison of Chunkyspread and a swarming-based approach over a wide range of applications (low- and high-bandwidth), churn scenarios, node heterogeneity, and social cooperation (freeloading). We need to measure such metrics as latency, packet loss and discontinuity, overhead (control and redundant stream), and simplicity. Ideally we want to do this for existing deployments. Since much of the P2P multicast activity is now happening in a proprietary commercial setting, we need to be creative about measurement studies.

Ultimately such a comparison study will produce improvements in both approaches, and may very well lead to a hybrid approach that works well under a wide range of scenarios.

VII. Acknowledgments

We would like to thank M. Castro and A. Rowstron for providing us the simulator code for Splitstream.

References

[1] V. Venkataraman and P. Francis. Chunkyspread: Multi-tree Unstructured Peer-to-Peer Multicast. In *The Fifth International Workshop on Peer-to-Peer Systems*, February 2006.

[2] X. Zhang, J. Liu, B. Li, and T.-S. P. Yum. DONet/CoolStreaming: A Data-driven Overlay Network for Live Media Streaming. In *Proceedings of IEEE Infocom*, Miami 2005.

[3] S. Banerjee, S. Lee, B. Bhattacharjee, and A. Srinivasan. Resilient multicast using overlays. In *Proceedings of ACM SIGMETRICS*, San Diego 2003.

[4] M. Bishop, S. Rao, and K. Sripanidkulchai. Considering Priority in Overlay Multicast Protocols Under Heterogeneous Environments. In *Proceedings of IEEE Infocom*, Barcelona 2006.

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

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

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

[8] 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)*.

[9] 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.

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

[11] V. Pai, K. Kumar, K. Tamilmani, V. Sambamurthy, and A. E. Mohr. Chainsaw: Eliminating Trees from Overlay Multicast. In *IPTPS' 05*.

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

[13] 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.

[14] 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.

[15] 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.

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

[17] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *The Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, 2002.

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

[19] 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.

[20] P. A. Chou, H. J. Wang, and V. N. Padmanabhan. Layered Multiple Description Coding. In *IEEE Packet Video Workshop*, Nantes, France, April 2003.

[21] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. W. OToole, Jr. Overcast: Reliable multicasting with an overlay network. In *Proc. of the 4th Usenix Sympp on Operating System Design and Implementation (OSDI 2000)*, October 2000.

[22] D. A. Helder, and S. Jamin. End-host multicast communication using switch-tree protocols. In *Proceedings of the GP2PC* May 2002.

[23] V. Venkataraman, K. Yoshida and P. Francis. Chunkyspread: Heterogeneous Unstructured Tree-based End System Multicast. Tech. Rep. cul.cis/TR2006-2027, Cornell University, Ithaca, NY.

[24] www.pplite.com

[25] www.veoh.com

[26] www.zattoo.com