# Chunkyspread: Heterogeneous Unstructured End System Multicast

Vidhyashankar Venkataraman, Kaoru Yoshida, Paul Francis
{vidya, kyoshida, francis}@cs.cornell.edu
Cornell University

## Abstract

**In order to maximize throughput in end-system multicast, it is necessary to have fine-grained control over the transmit load of each participating member. This both avoids bottlenecks where members are overloaded, and allows heterogeneous members to contribute as much transmit capacity as they are able or willing to. In this paper, we describe and simulate an unstructured end-system multicast protocol called Chunkyspread that provides members with fine-grained control over their transmit load, scales well, has relatively low latencies, and can tolerate high membership churn. Chunkyspread is designed as a flexible framework that easily incorporates different constraints and optimizations. For instance, it is straightforward to add tit-for-tat or path disjointness as constraints to the system. This paper demonstrates the performance of Chunkyspread through extensive simulations, and provides partial validation of these simulations on Emulab. It also provides detailed comparisons with Splitstream, a structured heterogeneous end-system multicast protocol. The simulations show that Chunkyspread provides far better control over transmit load than Splitstream, while exhibiting comparable or better latency and responsiveness to churn.**

## I. Introduction

In 1997 and 1998, Francis and Zhang 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 and a plethora of multicast protocols 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 [25]). 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, we focus on non-interactive multicast applications that can grow to a very large scale (many thousands of recipients), can tolerate high-churn, 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). To be honest, we focus on non-interactive applications because the delay tolerance required by interactive applications such as video conferencing, a few hundred milliseconds, seems extremely difficult to achieve at arbitrarily large scale and with high membership churn, without using IP multicast. In essence, we are sacrificing interactivity for scale and membership churn.

Once we accept that we can't achieve extreme low latencies, a few seconds of delay becomes tolerable. Indeed in the case of streaming media applications, a few seconds of delay is necessary in the form of a receiver play-out buffer to smooth over short-term disruptions in network or OS performance [18]. Allowing this much delay buys us considerable flexibility in the design, and in particular, allows us to exploit randomness in the overall structure of the protocol. Like other unstructured P2P applications, this allows us to work with relatively simple approaches. Having said that, all other things being equal, a low latency is still preferable, and we do provide

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

mechanisms to reduce latency.

In addition to large scale and robustness to churn, a critical requirement is to have fine-grained control over member transmit 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, control over load allows the system to exploit the heterogeneous capacities of members, thus maximizing the throughput of the system. Good performance requires that there be no bottlenecks: no node should be called on to transmit more than it can.

It is widely accepted that only *multi-path* approaches can lead to high utilization, since they allow all nodes to participate in transmission of the data stream (as opposed to single-tree approaches, which necessarily require that a large fraction of nodes be leaves and therefore contribute nothing) [7], [8]. By multi-path, we mean where each node receives portions of the multicast stream via different routes. A 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. This gives rise to the question of which approach to adopt: *per-block* (or packet) or *per-slice*.

In the case of protocols that build trees with per-block granularity, each node explicitly informs its neighbors of which blocks it has, and requests from each neighbor which blocks it would like to receive. This kind of a push-pull swarming strategy 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. Swarming also results in added delay to execute the push-pull, requiring that packets be buffered long enough to accomodate the delay and avoid packet loss.

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 adequately, all of its offspring in the tree will suddenly stop receiving some packets until the tree can be re-built. In an environment with constant churn, trees are continuously being destroyed and rebuilt, resulting in a considerable control message overhead. In order to avoid packet loss due to disruptions in the trees, nodes must buffer packets for the period of time it takes to repair a tree.

What all this means is that both swarming and tree-building approaches exhibit the same types of trade-offs. Both have control message overheads (though for different reasons), both suffer from substantial delays in packet reception (though for different reasons) and require some amount of buffering to prevent packet loss. To succeed, tree-building approaches must have simple tree creation and repair algorithms that converge very fast. Swarming approaches, on the other hand, must adopt strategies that minimize the overhead and delay of the push-pull. It is not at all clear which approach might emerge as the best by these measures. Ultimately some kind of hybrid strategy may be appropriate.

An important consideration is simplicity. In spite of the fact that we, the research community do not have good measures for "simplicity", it seems clear to us that swarming strategies are simpler than tree-building strategies. We believe that this simplicity makes swarming approaches easier to build and deploy, and ultimately results in more robust systems.

Despite the above arguments, we have chosen to place a stake in the ground, and that stake is a tree-building approach. Our reasoning for this boils down to two arguments. First, we believe that tree-building approaches can in fact be made quite simple, even if not as simple as swarming. For instance, we have chosen an unstructured approach that exploits bloom filters in the data path [17]. Second, as already discussed we believe that fine-grained control over transmission load is critical. We also believe that the multicast system should be able to easily incorporate other performance criteria and constraints such as tit-for-tat. Our intuition, as well as our experience so far, suggests that a certain amount of fine-tuning is required to consistently achieve a desired load balance, and that this fine-tuning inevitably takes a certain amount of time and overhead. Enforcing tit-for-tat constrains this fine-tuning even further. The long-term parent-child relationships inherent in trees allows us to amortize the cost of this fine-tuning over a relatively long

period of time.

By contrast, swarming, in its purest form, constantly reformulates what is exchanged between neighbors. Fine-tuning load balance or establishing enforceable tit-for-tat in this environment seems problematic. This may seem an odd thing to say given BitTorrent, whose success arguably derives from its tit-for-tat capability. The difference, however, is that BitTorrent is a file sharing protocol, not a real-time multicast protocol. The issues of delay and sustainable load don't come into play with file sharing, thus giving BitTorrent a form of flexibility that multicast doesn't have.

This paper makes the following contributions:

1) We give a detailed description of Chunkyspread[1], a new end-system multicast protocol that gives fine-grained control over each member's transmit load, reacts quickly to membership changes, exhibits relatively low latencies, 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.
2) We present a thorough simulation analysis of Chunkyspread's load control, latency optimization, responsiveness, and overhead.
3) Using the MSPastry simulation of Splitstream, we present an analysis of Splitstream for the same metrics, and compare Splitstream with Chunkyspread.
4) Again through simulation, we present preliminary and limited analysis of Chunkyspread for tit-for-tat, and for the basic trade-off of buffer size, data redundancy, and packet loss in the face of churn.
5) We present limited results of a complete implementation of Chunkyspread running on Emulab. These results 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, V discusses related work in our area while 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 detailed descriptions of its various components.

Chunkyspread constructs a single-source multicast group among a set of member end-systems. In other words, there is one sender, (which we call the *true source*), and multiple receivers. If the application requires multiple senders, then either multiple groups must be formed, or the multiple senders must first send to a designated single sender acting as the true source, which then transmits to the multicast group. Our implementation does not currently provide this latter capability nor does it provide a capability to change the true source in the middle of a multicast, although doing so would be relatively straightforward.

Like Splitstream, the true source transmits the multicast stream as $M$ distinct slices, where the $0^{th}$ packet is the first slice, the $1^{st}$ packet is the second slice, and so on until the $M^{th}$ packet which is the first slice, the $M + 1^{th}$ packet the second slice, and so on. Each set of M slices constitutes a *block* of stream. Each slice is transmitted over a separate multicast tree. The trees are not node-disjoint. Failure recovery is fast enough that node-disjointness doesn't help much. Note that the true source transmits each packet from each slice exactly once. It does not need to send greater than the stream volume (though it can if it wishes).

Applications can access Chunkyspread through an API that provides $join()$, $quit()$, $send()$, and $receive()$ primitives. The $quit()$ primitive provides functionality for both abrupt and graceful quits, where in the latter case, the member may briefly continue to transmit packets to its neighbors while they find alternates. (Note that the term member refers to receiving members only, not the true source. We use the terms member and node interchangeably.) The $join()$ primitive 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 will transmit at any time while the target load is the volume that the member would like to be sending at steady state. The expectation is that the steady state volume sent by the application will be near the target load: in fact, it may be slightly above or below.

Clearly, it is possible for members to set their maximum loads such that there is not enough capacity in

[1]Chunkyspread is a pseudonym required for blind submission.

the system to transmit the stream. For instance, if each member sets its maximum load to 50%, no member could receive the complete stream, and the system would fail to operate. It is up to the application to insure that this does not happen. One way to do this might be to ”hardwire” the application to always set the maximum load at 150% and target load at 100%. This, of course requires that the member host actually have the capacity to transmit at this rate. Alternatively, the application could measure the capacity of its host, and set the maximum and target loads accordingly. Nevertheless there must be enough capacity overall to transmit all streams to all members. No P2P multicast system can operate otherwise.

While the application (and therefore the application developer or user) operates in terms of a target load, the Chunkyspread protocol itself does not. What's more, Chunkyspread internally expresses load in units of the number of slices, not bandwidth or percentage of stream volume. Internally, 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 the number of slices that the stream is split, is 16. We experimented with more and less and found that performance is not very sensitive to this number, as we shall see later. 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*. The upper edge of the range 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 $LLT$, 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 an 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 one existing node, it participates in a continuously running distributed algorithm called Swaplinks [1] 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 [14] 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 nodes may have. It's default value is 8, and as far as we know, this value is universally appropriate. Since node degree is set proportionally to target load, the node degree of any nodes is set to be $ND = (TL/MinL) * MND$, where $TL$ is the target load. The system may choose to allow nodes' target load to be less than $MinL$. In this case, $ND$ is set to $MND$ (8). 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

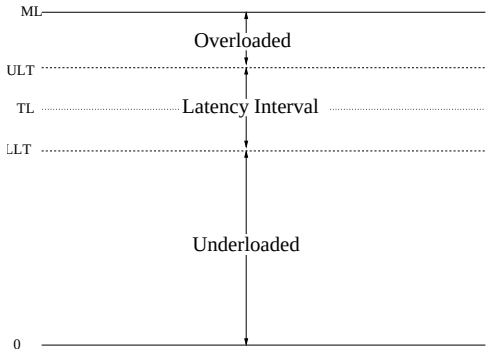



Fig. 1. The load-latency thresholds

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. 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*. If a slice source quits, then the true source discovers this and selects a new random node as the slice source. Note that a node may be a slice source for more than one slice.

Once a node is in the random graph, its job is to find a parent for each slice without forming a loop. We avoid and detect loops using bloom filters 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.

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 per-slice bloom filters, 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 neighborhood alterations, load or latency 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.

### A. Loop avoidance and detection

Bloom filters offer a spatially efficient method to detect and avoid loops, with a tunable rate of false positives[17]. 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[2]. 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.

### B. 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 that the best such switch takes 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.

The child then sends a switch message to the potential parent which accepts or rejects the request depending on its load and its bloom filter for that slice (these parameters may have changed from the time since the child had made

[2]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.

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. The switch message also contains the load parameters that were in force when the decision to switch was made. If these parameters have changed significantly in the interim, the switch is aborted.

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

### C. 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.

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 (in addition to or instead of latency).

Finally, requesting the best parent (either in terms of latency or load) to supply a slice can lead to an implosion of switch requests at such nodes. This implosion will not just increase the control overhead at such potential parents but will also lead to many of the switches to fail. To prevent this from happening, nodes choose one amongst a set of good potential parents instead of choosing the best parent.

### D. 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. The selected parent may reject the request if not doing so would push its load above its maximum load $ML$, but otherwise must accept the child.

Apart from this, a node that joins a multicast session whose trees have already been constructed through the flooding mechanism described above, may have to periodically request 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[1]. 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 if needed).

## III. Overview of Splitstream

Since we make simulation comparisons of Chunkyspread and Splitstream, a brief overview of Splitstream is provided here. Splitstream is a DHT-based multicast protocol that splits the stream into slices and transmits them over multiple trees. Splitstream is built 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 node C gets a request from a potential child A, it can choose to either drop one of its current children B based on whether A overlaps with C's ID more than B. The orphaned or preempted node (A or B) then contacts one of C's children and the process continues recursively. This is called the pushdown operation. If there still remains an orphaned node after the pushdown operation, it contacts a group maintained by Scribe comprising of nodes that have excess capacity left. A depth-first traversal is made on this group to find a node that can provide the stream to the orphaned node[6]. This is the anycast operation.

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: not only is the preempted node disconnected, but so are its descendants in the tree. 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 cycle-free and route-convergence guarantees offered by the underlying DHT [13] 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, 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 [10], and set delays proportional to the distance metric of the resulting topology. We chose to select a very pessimistic value for network latencies: the median latency is around 400ms, and the maximum is roughly 650ms. As a result, the convergence times shown are worse than one might expect over the commercial Internet (for both Chunkyspread and Splitstream). 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. To scale the simulation, the simulator does not explicitly generate data packets. We do, however, calculate the amount of time it would have taken for a packet to travel node to node. This calculation is needed both in determining when bloom filter information arrives at each node, and for calculating the relative slice arrival time used to improve latency.

Member nodes in the simulation receive all slices. 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. The default number of slices in our simulations is $M = 16$. To model heterogeneity, each node is assigned a random node degree within a specified range. By default, the range is from 8 to 50 inclusive, thus producing a roughly 6x range of loads. 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 target load $TL$ for each node is derived from its node degree in such a way that the sum of target loads across all nodes is approximately equal to the total volume needed to transmit the stream to all nodes.

This results in default values for $TL$ being distributed uniformly between 4 and 28 slices (the median of 4 and 28 is 16, the number of slices). The default setting for maximum load is $ML = (1.5)TL$. In other words, the total capacity of the system is 50% more than what is needed to transmit the stream to all nodes. This represents a well-provisioned system: something required in any event to get good performance[11].

We experiment with two settings for $LLT$ and $ULT$. In one, both are set to 0 slices from $TL$ ($ULT = LLT = TL$), 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 $ULT = 18$. If $TL = 28$, then $LLT = 24$ and $ULT = 32$. 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.

We compared Chunkyspread simulations with those of Splitstream, which uses a simulator coded in C# that was provided to us by Miguel Castro. Wherever possible, we provide apples-to-apples comparisons of Chunkyspread and Splitstream. For instance, the Splitstream simulations are run over the same GT-ITM synthetic routing topology and have 16 slices. Splitstream does not, however, have parameters analogous to target load $TL$ and upper and lower thresholds $ULT$ and $LLT$. Rather, Splitstream provides a single parameter, 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 an Chunkyspread node may temporarily trasmit 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. Specifically, $SML$ means "a transmission rate at which I would be perfectly happy to operate," whereas $ML$ means "a transmission rate that I am capable of achieving for brief periods, but would rather not".

Because of this difference, in one case we treat $SML$ as though it were equivalent to $ML$. That is, we set it to be 50% above the number of slices ($SML = 1.5TL$) where $TL$ is the target loads for the corresponding nodes in Chunkyspread. This is denoted as $SS(1.5)$. In the other case, however, we try to treat $SML$ as though it were equivalent to $ULT$. As such, we set $SML = 1.2TL$ 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 $SML = 1.1TL$, 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 considered the following scenarios to evaluate our protocol.

**Static scenario:** This corresponds to the case when all overlay nodes are present in the network right from the beginning of the simulation. This means that the random graph is constructed completely, even before the *true source* starts building trees to kick-start the multicast session. This scenario is not a very realistic one but is useful in analyzing just the performance of our load-latency algorithm without the influence of any churn. The Swaplinks simulator did not have functionality provided for locality-awareness. To determine the effect of adding locality to the random graph, we have run static simulations where, in addition to the random neighbors selected by Swaplinks, some number of nearest neighbors were added to the neighbor set of each node. For all the other scenarios which involve churn, we did not incorporate locality since the nearest neighborhood set alters with churn, and we did not want these changes to affect the degree invariant guarantees offered by Swaplinks.

**Join scenario:** There are 3750 overlay nodes in the network (similar to the static case) 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 had reached a steady state. This scenario depicts a more realistic picture than the previous one; it can possibly be a live event that attracts a large audience within a short span of time.

**Bursty failures:** There are 5000 nodes in the network and a percentage of the nodes fail at the *same time instant*. We consider two cases: One when 10% of the nodes fail (small burst) and the other when 50% of the nodes fail (large burst). These pathological cases may not be very close to realistic scenarios, but do help in analyzing the robustness of the protocol against node failures. Such a high failure rate could potentially lead to

network partitions, but Swaplinks was resilient enough to prevent them from happening.

**Churn scenario:** 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 scenario that we have studied is similar to the one tested in [13]. 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[11]. The churn happens for the first 1000 seconds after which the remaining live nodes are allowed to settle down for the next 200 seconds.

### A. 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.

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

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 $\frac{100.(ML-TL)}{TL}$% which is 50% in our Chunkyspread simulations.

To evaluate the latencies, we first observe two parameters: 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 parameter to evaluate our latencies. Network stretch is defined as the ratio of the measured overlay latency to the network latency between the true source and the node. 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(a) 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 good as in cases without any churn at all. The heavy tails observed on the negative side of the x axis in these curves are because of the imperfect configurations that we had mentioned earlier.

Figure 2(b) 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 gives us an idea of how long it takes to receive all the slices for the same block of the stream and hence the size of the application buffer required to counter losses while waiting for the full block. 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.

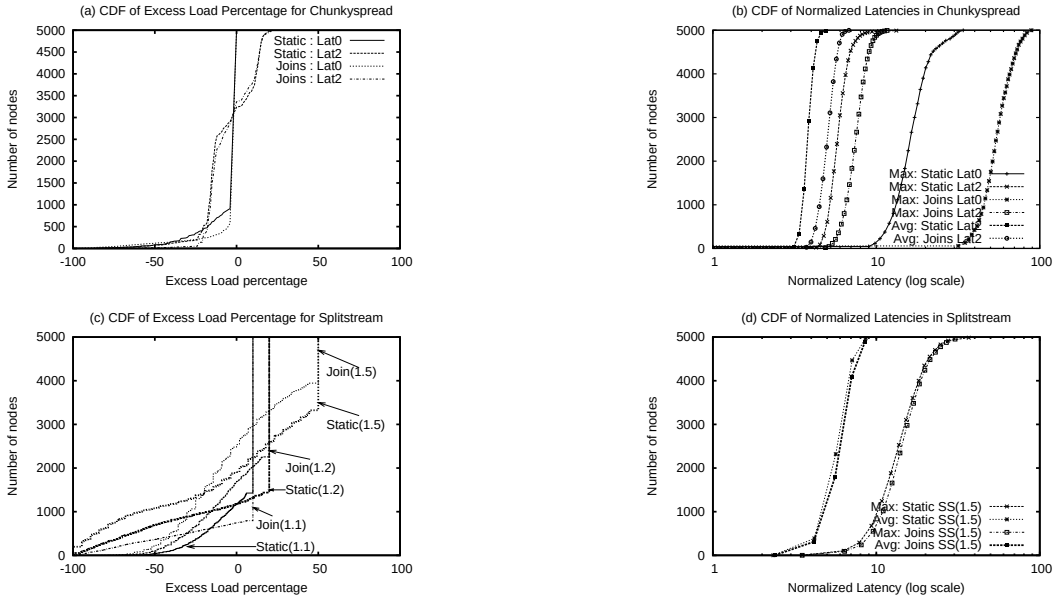



Fig. 2. Load distributions and the corresponding latencies in the static and the join cases of Chunkyspread and Splitstream

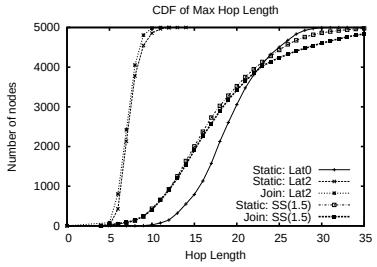



Fig. 3. Hop length in Chunkyspread and Splitstream

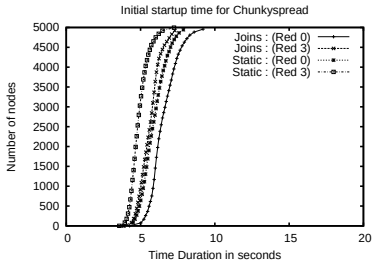



Fig. 4. Initial Startup Times for Chunkyspread

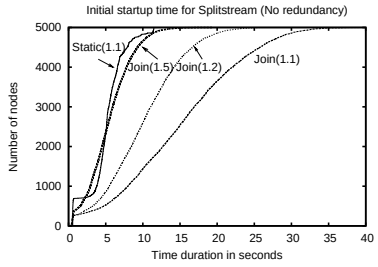



Fig. 5. Initial Startup Times for Splitstream

The overlay stretch of a node is defined as the ratio of the average latency observed over its slices to its minimum latency in the overlay graph with the true source. It is a measure of the quality of the latency algorithm just with respect to the underlying random graph constructed. We see that over 90% of the nodes have overlay stretches between 1 and 2 for $Lat2$ in the static scenario; the join scenario also incurs similar values. This shows the good performance of the latency phase of our algorithm.

Let us now see how Splitstream fares with respect to load and latency. Figure 2(c) 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. As expected, a considerable number of nodes get saturated to their $ML$ values and the percentage of such saturated nodes increases as $\frac{ML}{TL}$ values decrease. For example, the percentage is 35% for SS(1.5), 60% for SS(1.2) 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 balance algorithm ensures the newly joined nodes also participate in supplying the slices.

The graph in Figure 2(d) shows cdfs of the average and maximum latencies in the static and the join cases. We note that both the average and the maximum latencies showed very marginal improvements as $\frac{ML}{TL}$ was increased with both the static and the join scenarios performing comparably[3]. Hence 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

[3]The comparable performances show that curbing the spare capacities do not have a significant effect on the latencies.

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 counter losses due to 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 cdfs of maximum hop length (from the true source) across all slices received by each overlay node as shown in Figure 3. As we had already mentioned, higher hop lengths relate to a lower tree resilience, since nodes are more prone to disruptions from the trees due to the failure of one or more ancestors in the path to the true source. From the graph, we see that $Lat0$ yields very high hop lengths. We also see that there is a good improvement with *Lat2* which yields $90^{th}$ percentile hop lengths of around 8 in both the static and the join scenarios. We note that the static scenario performs comparably with the join scenario, though it had outperformed the latter in the latency figures, as we had discussed above. This is because the static case builds locality-aware graphs which usually yield lower latencies at the cost of greater hop lengths. Again, Splitstream performs much worse than Chunkyspread's $Lat2$: with $90^{th}$ percentile values as high as 30. The reason why this happens has been discussed above.

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. In Chunkyspread, a newly joined node initially gets its stream by periodically pinging its neighbors (as described in Section II-D) and this mechanism is independent of the choice of load-latency parameters used. In Splitstream, this quantity is not useful enough 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[4] from the tree due to such preemptions, into the initial startup time.

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, hence 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 5 shows the initial startup times of Splitstream. As claimed in [7], 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. We see that as the spare capacities in the system decrease, the initial startup time increases as seen by the curves for the join scenario. This is expected, since, with lesser spare capacity, more time has to be spent during the pushdown and the anycast operations. 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. Such a difference is not observed in the static case, since all nodes start with zero load and many nodes, for many of the slices, obtain their parents without even resorting to pushdown or anycast operations. This is not the case in the join scenario, since many of the nodes may have been already saturated to their $ML$ values and the newly joined nodes result in more anycast and pushdown operations. We note that with Chunkyspread, the load balance algorithm ensures that the spare capacities are distributed across nodes even when nodes are joining at a high rate, as we will show below.

*2) Time to convergence:* We now assess the convergence properties of our algorithm. A system is said to have converged if it has reached a steady state. 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*

[4]Disconnection due to orphaning a node will lead to disconnections of its descendants in that tree, if any.

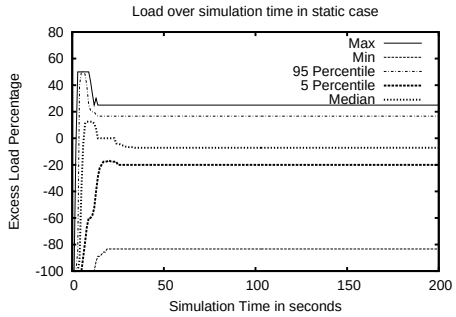



Fig. 6. Load of nodes over the simulation time in static case (Lat2)

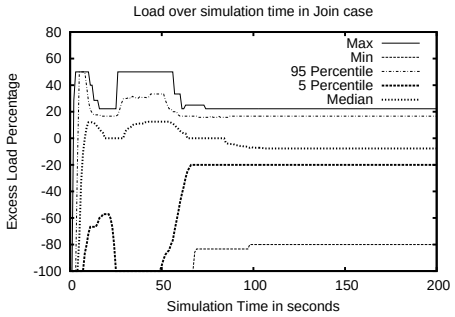



Fig. 7. Load of nodes over the simulation time in join case (Lat2)

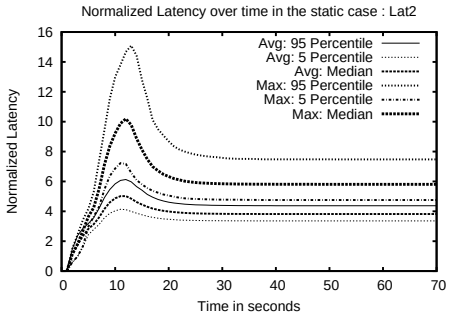



Fig. 8. Maximum and average latencies over the simulation time in the static case

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. It is important to note that Splitstream reaches a steady state as soon as the last orphan node gets a parent. Hence the convergence time is actually the startup time that we had discussed previously.

Figures 6 and 7 show the excess load percentage per node as the simulation proceeds in the case of *Lat2* for the static and the join scenarios respectively. The maximum and the $95^{th}$ percentile curves in the static case peak to $ML$ during the first 10 seconds of the simulation after which the load phase of the algorithm brings both the curves down to within the target upload interval. This shows that our algorithm can distribute the loads fairly across nodes quite fast, so that if more nodes were to join, there is a good chance that there is spare capacity within their neighborhoods. This is in fact depicted in Figure 7. In the join scenario, we can observe a peak during the first ten 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 curve stays roughly at 30% while the median hovers around 10% during this time which show that there are a considerable number of nodes with spare capacity that can serve a newly joined node quite fast.

Figure 8 shows the normalized average latency over the slices of nodes as the simulation proceeds in the static scenario. We observe that the load phase shoots the latency up initially, but then, the latency phase of the algorithm steadily brings the 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

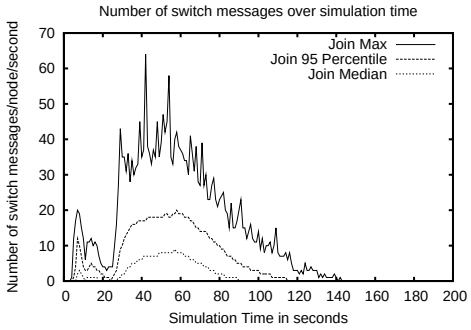



Fig. 9. Control Overhead in Chunkyspread: Switch messages over simulation time

algorithm is more dominant than the latency phase; such cases happen after there is churn or after the true source kickstarts the multicast session.

*3) Control Overhead:* Next, we evaluate the the control overhead incurred by nodes in the network due to switch messages. Figure 9 shows the number of switch messages sent per node per second over the simulation time of 200 seconds when *Lat2* is run on the join scenario. The peaks correspond to the time when nodes are joining the system and also after the true source kickstarts 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 values of the switch messages are about 20 and 8 messages per second per node respectively. This indicates a modest overhead amongst Chunkyspread nodes even at the time when there is a high join rate.

We observe that a switch is usually a three-party negotiation but is asymmetric in the number of switch messages sent at each node: A load-based switch originates from the original parent and ends at it; this involves a total of 4 messages. A latency-based switch involves 5 messages since it starts from the child seeking the current parent to let it switch. Other kinds of parent selections

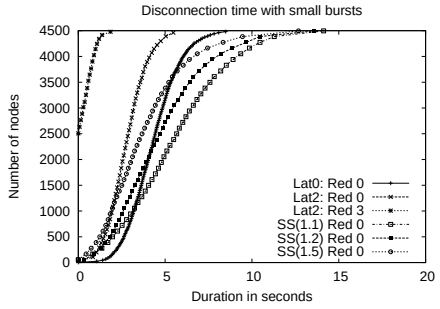



Fig. 10. Recovery duration for 10% burst

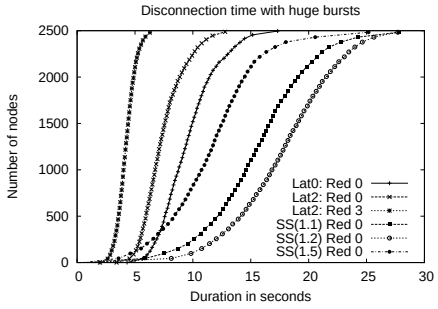



Fig. 11. Recovery duration for 50% burst

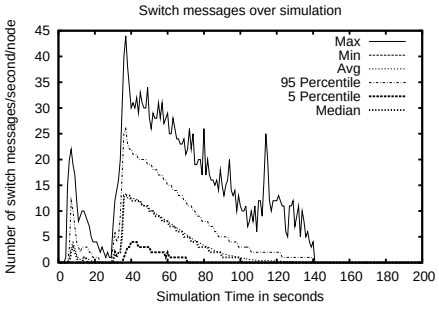



Fig. 12. Switch messages over time in 10% case

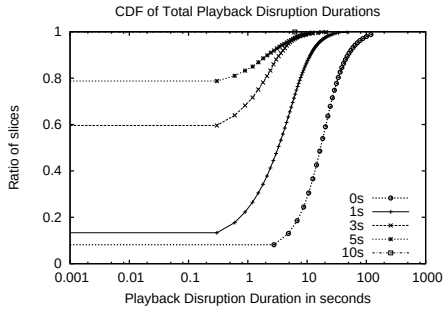



Fig. 13. Total playback disruption duration across slices for various buffer sizes

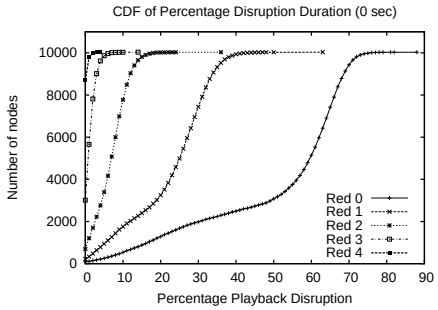



Fig. 14. Percentage playback disruption duration without any buffer

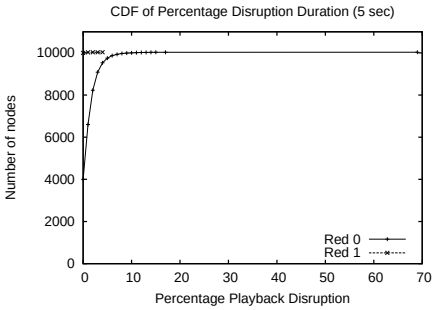
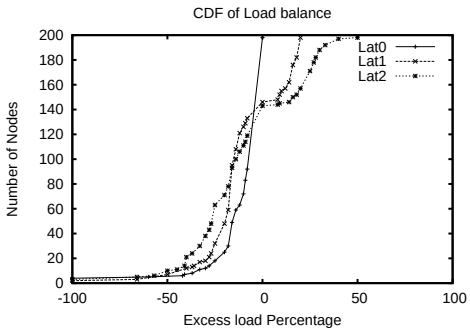



Fig. 15. Percentage playback disruption duration with 5s buffer



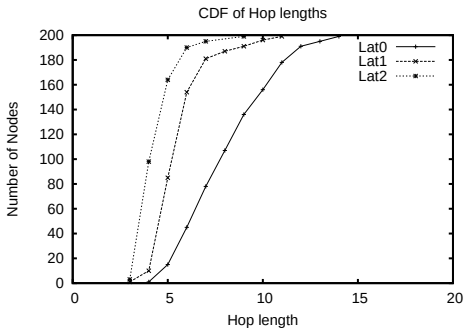




Fig. 16. Emulation results: (a) CDF of Load Distribution (b) CDF of Hop lengths

(the forced ones) involve two or three switch messages depending on whether the old parent is also involved in the switch or not.

Moreover, there can be switches which are rejected either at the child or at the potential parent. Hence the number of switch messages may not exactly correspond to the number of successful switches. We examined the number of successful switches at each node over the course of the simulation time. We observed that in the join scenario, the dominant peak value of the maximum number of switches is around 12 per second per node and happens during the time nodes join, while the $95^{th}$ percentile value is around 3 switches. We found during this period that almost 50% of the switch requests fail due to conditions that we had already discussed.

### B. 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 in both the cases, 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 get accounted to their recovery duration (since an ancestor is trying to recover on their behalf). We note that, with no redundancy in the 16 slices, if a node is disconnected from even one slice

tree it gets accounted in the recovery duration. Figure 10 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. We also find that Splitstream nodes are disconnected for longer durations as $\frac{ML}{TL}$ values decrease.

Splitstream performs worse than Chunkyspread when 50% of the nodes fail at the same instant. Figure 11 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 lengths that Splitstream incur (as we had seen in Figure 5), which affects its robustness to node failures.

Figure 12 shows the number of switch messages over the course of the simulation in the 10% burst case with $Lat2$. We see that the maximum control overhead peaks at 42 messages per second per node just after failures are detected while the median value peaks to 12 messages per second per node. This represents a moderate amount of overhead.

### C. Effect of other parameters

We tried to see the effect of altering parameters such as the number of 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.

We studied the performance of our protocol on varying the number of slices that the stream is split. We found that on increasing the number of slices from 8 to 32, there was an improvement in the load balance: in the static case of $Lat0$, around 90% of the nodes reached their $TL$ values when multicast with 32 slices while 90% of the nodes were within 20% of their $TL$ values when multicast with 8 slices. The more striking feature between the two cases was the tail; the least loaded node had an excess load percentage of -25% for 32 slices and -60% for 8 slices. The better performance with increased slices can be explained by the fact that more slices offer a finer granularity in controlling load. But this is at the cost of proportionally more number of switch messages, hence more control overhead. Increasing the number of slices does not have any bearing on the latencies though.

Another set of experiments was conducted to study how neighborhood sizes affect our algorithms. To simplify our study, we considered a static homogeneous scenario with all nodes having a target upload of 16 slices. We tested our protocol with graphs having constant degrees of 12, 24 and 36 for the $Lat2$ case. We observed that as the number of neighbors was increased, more nodes were closer to their target loads, though the improvement was not significant. In the 10% burst case, we found that when the number of neighbors was increased from 12 to 24, the recovery duration improved the $90^{th}$ percentile values from 4 seconds to just 2 seconds. This is because with more neighbors, a node disconnected from the tree has a greater chance of picking up a parent from its neighbors that can supply the slice.

We also experimented with varying levels of heterogeneity in the network. We tested the simulator with three scenarios: (a) the homogeneous scenario (in which number of neighbors is assigned to 32 for all nodes), (b) moderate heterogeneity (similar to our default case), and (c) high heterogeneity (in which the number of neighbors is distributed between 8 and 200 neighbors). As expected, we observed that (a) did better than (b) and (c) with respect to load balance, but the improvement was not significant. In scenario (a) all the nodes were within 12.5% (or $\frac{2}{16}^{th}$) of the $TL$ value which is 16 for all the nodes. Scenario (b) is the curve obtained in 2(a) for the Static $Lat2$ case. Scenario (c) performed very close to (b); from these numbers, we can infer that our protocol performance is independent of the degree of heterogeneity in the system.

### D. Churn scenario

We have so far considered isolated node joins and failures in our simulations. As we had already noted, a more realistic churn scenario would be to consider one

in which nodes join and leave at the same time. We had already discussed in the beginning of this section, the join and stay time parameters used in simulating the churn.

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[5] 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.

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 Figures 14 and 15. 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, 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 observed in Figure 15. From these graphs, we observe the tradeoff between the buffer size, redundancy and the playback disruption duration, which is fundamental to any streaming protocol.

### E. 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[6]. 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 16(a) and 16(b) 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.

### F. Intuition on 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[7]. 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 *Lat2* 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

[5]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.

[6]Swaplinks does not retrieve locality-aware neighbors, hence hop length can still be a reasonable parameter.

[7][12] and [19] argue that strict tit-for-tat is impractical, and our simulations corroborate this.

that decreasing the ratio improves load balance, but at the expense of latency. For instance, the $90^{th}$ 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.

## V. Related work

There has been considerable work in the past on single-tree multicast protocols[4], [3], [21], [6], [22]. Since none of these effectively support heterogeneity, we restrict our discussion of related work to multi-path multicast protocols.

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 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. A protocol that has a similar approach to Chainsaw has been deployed in the real world and has met with some success. However, the real-world deployment depended on a considerable number of dedicated servers, hence it did not still show enough evidence on a true decentralized deployment.

[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-ridings the system. [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. [16] 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.

## VI. 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 apples-to-apples comparisons with Splitstream, and find that Chunkyspread performs better across the board, and significantly better with respect to control over load.

While preliminary results with severe churn are promising, more work needs to be done to understand the trade-offs between packet loss, packet delay (buffering), and stream volume (packet coding schemes). This understanding needs to be developed for both tree-based and for treeless approaches such as Chainsaw. Our intuition is that neither approach in its pure form will perform really well, and that some form of hybrid approach is called for.

Preliminary results with tit-for-tat also show promise, though once again there is much work still to be done. We hope to explore a range of tit-for-tat mechanisms, including both social and irrational behavior. Tit-for-tat also needs to be examined for both tree-based and treeless approaches.

While we believe that gaining a better understanding of severe churn and tit-for-tat represent the most fruitful areas of research, we still need to consider ways to improve Chunkyspread. For instance, we feel that Chunkyspread as designed, has too many parameters that need to be set. Is it possible for Chunkyspread nodes to self-tune based on observations within the overlay, possibly achieving parameter-less operation? Also, while the Chunkyspread framework does provide something of a generic constraints-and-optimizations framework, we still find ourselves selecting specific parameters for specific optimizations. Can we generalize the framework further, for instance allowing application developers to simply supply high-level policies about various criteria of interest?

Beyond this, we would like to explore different types of applications and environments. These include low-latency applications, reliable delivery, and pub-sub applications where nodes may join a large number of groups.

## VII. Acknowledgments

We would 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, San Francisco, CA*.

[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] 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.

[16] 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.

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

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

[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 2nd Workshop on Global and Peer-to-Peer Computing on Large Scale Distributed Systems (GP2PC)* May 2002.

[23] P. Eugster, S. Handurukande, R. Guerraoui, A.-M. Kermarrec, and P. Kouznetsov. Lightweight probabilistic broadcast. In *Proc. of the Intl. Conf. on Dependable Sys. and Networks (DSN 2001)*, July 2001.

[24] Y. Tang, M. Zhang, J. Luo, Y. Zhong. Experience on Peer-to-Peer based Live Video Streaming. In *Proceedings of 12th International Multimedia Modelling Conference*, pp.80-87, Beijing, China, January 2006.

[25] Private conversation with Bruce Davie.