SplitStream: High-Bandwidth Multicast in Cooperative Environments *

Miguel Castro[1] Peter Druschel[2] Anne-Marie Kermarrec[1] Animesh Nandi[2]
Antony Rowstron[1] Atul Singh[2]

[1]Microsoft Research, 7 J J Thomson Avenue, Cambridge, CB3 0FB, UK.
[2]Rice University, 6100 Main Street, MS-132, Houston, TX 77005, USA[†]

ABSTRACT

In tree-based multicast systems, a relatively small number of interior nodes carry the load of forwarding multicast messages. This works well when the interior nodes are highly-available, dedicated infrastructure routers but it poses a problem for application-level multicast in peer-to-peer systems. SplitStream addresses this problem by striping the content across a forest of interior-node-disjoint multicast trees that distributes the forwarding load among all participating peers. For example, it is possible to construct efficient SplitStream forests in which each peer contributes only as much forwarding bandwidth as it receives. Furthermore, with appropriate content encodings, SplitStream is highly robust to failures because a node failure causes the loss of a single stripe on average. We present the design and implementation of SplitStream and show experimental results obtained on an Internet testbed and via large-scale network simulation. The results show that SplitStream distributes the forwarding load among all peers and can accommodate peers with different bandwidth capacities while imposing low overhead for forest construction and maintenance.

Categories and Subject Descriptors

C.2.4 [**Computer-Communications networks**]: Distributed Systems—*Distributed applications*; C.2.2 [**Computer-Communications networks**]: Network Protocols—*Applications, Routing protocols*; D.4.5 [**Operating Systems**]: Reliability—*Fault-tolerance*; D.4.8 [**Operating Systems**]: Performance

General Terms

Algorithms, Measurement, Performance, Reliability, Experimentation

*This paper contains improved Planetlab experimental results obtained after the Camera Ready version.

[†]Supported in part by NSF (ANI-0225660) and by a Texas ATP (003604-0079-2001) grant.

SOSP'03, October 19–22, 2003, Bolton Landing, New York, USA.

Keywords

Peer-to-peer, application-level multicast, end-system multicast, content distribution, video streaming

1. INTRODUCTION

End-system or application-level multicast [8, 16, 22, 19, 40, 32, 13, 6, 23] has become an attractive alternative to IP multicast. Instead of relying on a multicast infrastructure in the network (which is not widely available), the participating hosts route and distribute multicast messages using only unicast network services. In this paper, we are particularly concerned with application-level multicast in peer-to-peer (p2p) or *cooperative* environments where peers contribute resources in exchange for using the service.

Unfortunately, conventional tree-based multicast is inherently not well matched to a cooperative environment. The reason is that in any multicast tree, the burden of duplicating and forwarding multicast traffic is carried by the small subset of the peers that are interior nodes in the tree. The majority of peers are leaf nodes and contribute no resources. This conflicts with the expectation that all peers should share the forwarding load. The problem is further aggravated in high-bandwidth applications, like video or bulk file distribution, where many peers may not have the capacity and availability required of an interior node in a conventional multicast tree. SplitStream addresses these problems by enabling efficient cooperative distribution of high-bandwidth content in a peer-to-peer system.

The key idea in SplitStream is to *split* the content into k stripes and to multicast each stripe using a separate tree. Peers join as many trees as there are stripes they wish to receive and they specify an upper bound on the number of stripes that they are willing to forward. The challenge is to construct this *forest* of multicast trees such that an interior node in one tree is a leaf node in all the remaining trees and the bandwidth constraints specified by the nodes are satisfied. This ensures that the forwarding load can be spread across all participating peers. For example, if all nodes wish to receive k stripes and they are willing to forward k stripes, SplitStream will construct a forest such that the forwarding load is evenly balanced across all nodes while achieving low delay and link stress across the system.

Striping across multiple trees also increases the resilience to node failures. SplitStream offers improved robustness to node failure and sudden node departures like other systems that exploit path diversity in overlays [4, 35, 3, 30]. Split-

Stream ensures that the vast majority of nodes are interior nodes in only one tree. Therefore, the failure of a single node causes the temporary loss of at most one of the stripes (on average). With appropriate data encodings, applications can mask or mitigate the effects of node failures even while the affected tree is being repaired. For example, applications can use erasure coding of bulk data [9] or multiple description coding (MDC) of streaming media [27, 4, 5, 30].

The key challenge in the design of SplitStream is to construct a forest of multicast trees that distributes the forwarding load subject to the bandwidth constraints of the participating nodes in a decentralized, scalable, efficient and self-organizing manner. SplitStream relies on a structured peer-to-peer overlay [31, 36, 39, 33] to construct and maintain these trees. We implemented a SplitStream prototype and evaluated its performance. We show experimental results obtained on the PlanetLab [1] Internet testbed and on a large-scale network simulator. The results show that SplitStream achieves these goals.

The rest of this paper is organized as follows. Section 2 outlines the SplitStream approach in more detail. A brief description of the structured overlay is given in Section 3. We present the design of SplitStream in Section 4. The results of our experimental evaluation are presented in Section 5. Section 6 describes related work and Section 7 concludes.

2. THE SPLITSTREAM APPROACH

In this section, we give a detailed overview of SplitStream's approach to cooperative, high-bandwidth content distribution.

2.1 Tree-based multicast

In all multicast systems based on a single tree, a participating peer is either an interior node or a leaf node in the tree. The interior nodes carry all the burden of forwarding multicast messages. In a balanced tree with fanout f and height h, the number of interior nodes is $\frac{f^h-1}{f-1}$ and the number of leaf nodes is f^h. Thus, the fraction of leaf nodes increases with f. For example, more than half of the peers are leaves in a binary tree, and over 90% of peers are leaves in a tree with fanout 16. In the latter case, the forwarding load is carried by less than 10% of the peers. All nodes have equal inbound bandwidth, but the internal nodes have an outbound bandwidth requirement of 16 times their inbound bandwidth. Even in a binary tree, which would be impractically deep in most circumstances, the outbound bandwidth required by the interior nodes is twice their inbound bandwidth. Deep trees are not practical because they are more fault prone and they introduce larger delays, which is a problem for some applications.

2.2 SplitStream

SplitStream is designed to overcome the inherently unbalanced forwarding load in conventional tree-based multicast systems. SplitStream strives to distribute the forwarding load over all participating peers and respects different capacity limits of individual peers. SplitStream achieves this by splitting the multicast stream into multiple stripes, and using separate multicast trees to distribute each stripe.

Figure 1 illustrates how SplitStream balances the forwarding load among the participating peers. In this simple example, the original content is split into two stripes and multicast in separate trees. For simplicity, let us assume that the original content has a bandwidth requirement of B and that each stripe has half the bandwidth requirement of the original content. Each peer, other than the source, receives both stripes inducing an inbound bandwidth requirement of B. As shown in Figure 1, each peer is an internal node in only one tree and forwards the stripe to two children, which yields an outbound bandwidth of no more than B.

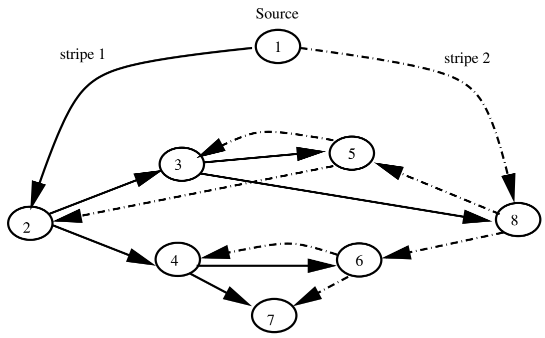


Figure 1: A simple example illustrating the basic approach of SplitStream. The original content is split into two stripes. An independent multicast tree is constructed for each stripe such that a peer is an interior node in one tree and a leaf in the other.

In general, the content is split into k stripes. Participating peers may receive a subset of the stripes, thus controlling their inbound bandwidth requirement in increments of B/k. Similarly, peers may control their outbound bandwidth requirement in increments of B/k by limiting the number of children they adopt. Thus, SplitStream can accommodate nodes with different bandwidths, and nodes with unequal inbound and outbound network capacities.

This works well when the bandwidth bottleneck in the communication between two nodes is either at the sender or at the receiver. While this assumption holds in many settings, it is not universally true. If the bottleneck is elsewhere, nodes may be unable to receive all desired stripes. We plan to extend SplitStream to address this issue. For example, nodes could monitor the packet arrival rate for each stripe. If they detect that the incoming link for a stripe is not delivering the expected bandwidth, they can detach from the stripe tree and search for an alternate parent.

2.3 Applications

SplitStream provides a generic infrastructure for high-bandwidth content distribution. Any application that uses SplitStream controls how its content is encoded and divided into stripes. SplitStream builds the multicast trees for the stripes while respecting the inbound and outbound bandwidth constraints of the peers. Applications need to encode the content such that *(i)* each stripe requires approximately the same bandwidth, and *(ii)* the content can be reconstructed from any subset of the stripes of sufficient size.

In order for applications to tolerate the loss of a subset of stripes, they may provide mechanisms to fetch content from other peers in the system, or may choose to encode content in a manner that requires greater than B/k per stripe, in return for the ability to reconstitute the content from less than k stripes. For example, a media stream could be encoded using MDC so that the video can be reconstituted from any subset of the k stripes with video quality proportional to the number of stripes received. Hence, if an interior node in a stripe tree should fail then clients deprived of the stripe are able to continue displaying the media stream at reduced quality until the stripe tree is repaired. Such an encoding also allows low-bandwidth clients to receive the video at lower quality by explicitly requesting less stripes.

Another example is the multicasting of file data with erasure coding [9]. Each data block is encoded using erasure codes to generate k blocks such that only a (large) subset of the k blocks is required to reconstitute the original block. Each stripe is then used to multicast a different one of the k blocks. Participants receive all stripes and once a sufficient subset of the blocks is received the clients are able to reconstitute the original data block. If a client misses a number of blocks from a particular stripe for a period of time (while the stripe multicast tree is being repaired after an internal node has failed) the client can still reconstitute the original data blocks due to the redundancy. An interesting alternative is the use of rateless codes [24, 26], which provide a simple approach to coordinating redundancy, both across stripes and within each stripe.

Applications also control when to create and tear down a SplitStream forest. Our experimental results indicate that the maximum node stress to construct a forest and distribute 1 Mbyte of data is significantly lower than the node stress placed on a centralized server distributing the same data. Therefore, it is perfectly reasonable to create a forest to distribute a few megabytes of data and then tear it down. The results also show that the overhead to maintain a forest is low even with high churn. Therefore, it is also reasonable to create long-lived forests. The ideal strategy depends on the fraction of time that a forest is used to transmit data.

2.4 Properties

Next, we discuss necessary and sufficient conditions for the feasibility of forest construction by any algorithm and relate them with what SplitStream can achieve.

Let N be the set of nodes and k be the number of stripes. Each node $i \in N$ wants to receive I_i ($0 < I_i \leq k$) distinct stripes and is willing to forward a stripe to up to C_i other nodes. We call I_i the node's desired indegree and C_i its forwarding capacity. There is a set of *source nodes* ($S \subseteq N$) whose elements originate one or more of the k stripes (i.e., $1 \leq |S| \leq k$). The forwarding capacity C_s of each source node $s \in S$ must at least equal the number of stripes that s originates, T_s.

DEFINITION 1. *Given a set of nodes N and a set of sources $S \subseteq N$, forest construction is* feasible *if it is possible to connect the nodes such that each node $i \in N$ receives I_i distinct stripes and has no more than C_i children.*

The following condition is obviously necessary for the feasibility of forest construction by any algorithm.

CONDITION 1. *If forest construction is feasible, the sum of the desired indegrees cannot exceed the sum of the forwarding capacities:*

$$\sum_{\forall i \in N} I_i \leq \sum_{\forall i \in N} C_i \tag{1}$$

Condition 1 is necessary but not sufficient for the feasibility of forest construction, as the simple example in Figure 2 illustrates. The incoming arrows in each node in the figure correspond to its desired indegree and the outgoing arrows correspond to its forwarding capacity. The total forwarding capacity matches the total desired indegree in this example but it is impossible to supply both of the rightmost nodes with two distinct stripes. The node with forwarding capacity three has desired indegree one and, therefore, it can only provide the same stripe to all its children.

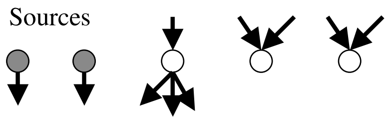


Figure 2: An example illustrating that condition 1 is not sufficient to ensure feasibility of a SplitStream forest.

Condition 2 prevents this problem. It is sufficient to ensure feasibility because it prevents the concentration of forwarding capacity in nodes that are unable to forward all stripes.

CONDITION 2. *A sufficient condition for the feasibility of forest construction is for Condition 1 to hold and for all nodes whose forwarding capacity exceeds their desired indegree to receive or originate all k stripes, i.e.,*

$$\forall i : C_i > I_i \Rightarrow I_i + T_i = k. \tag{2}$$

This is a natural condition in a cooperative environment because nodes are unlikely to spend more resources improving the quality of service perceived by others than on improving the quality of service that they perceive. Additionally, inbound bandwidth is typically greater than or equal to outbound bandwidth in consumer Internet connections.

Given a set of nodes that satisfy Condition 2, the SplitStream algorithm can build a forest with very high probability provided there is a modest amount of spare capacity in the system. The probability of success increases with the minimum number of stripes that nodes receives, I_{min}, and the total amount of spare capacity, $C = \sum_{\forall i \in N} C_i - \sum_{\forall i \in N} I_i$. We derive the following rough upper bound on the probability of failure in Section 4.5:

$$|N| \times k \times (1 - \frac{I_{min}}{k})^{\frac{C}{k-1}} \tag{3}$$

As indicated by the upper bound formula, the probability of success is very high even with a small amount of spare capacity in the system. Additionally, we expect I_{min} to be large for most applications. For example, erasure coding for reliable distribution of data and MDC for video distribution perform poorly if peers do not receive to most stripes. Therefore, we expect configurations where all peers receive all stripes to be common. In this case, the algorithm can guarantee efficient forest construction with probability one even if there is no spare capacity.

In an open cooperative environment, it is important to address the issue of free loaders, which appear to be prevalent in Gnutella [2]. In such an environment, it is desirable to strengthen Condition 1 to require that the forwarding capacity of each node be greater than or equal to its desired indegree (i.e., $\forall i \in N : C_i \geq I_i$). (This condition may be unnecessarily strong in more controlled settings, for example, in a corporate intranet.) Additionally, we need a mechanism to discourage free loading such that most participants satisfy the stronger condition. In some settings, it may be sufficient to have the SplitStream implementation enforce the condition in the local node. Stronger mechanisms may use a trusted execution platform like Microsoft's Palladium or a mechanism based on incentives [28]. This is an interesting area of future work.

3. BACKGROUND

SplitStream is implemented using tree-based application-level multicast. There are many proposals for how application-level multicast trees can be built and maintained [8, 19, 40, 13, 32, 22, 6, 23]. In this paper, we consider the implementation of SplitStream using Scribe [13] and Pastry [33]. It could also be implemented using a different overlay protocol and group communication system; for example, Bayeux on Tapestry [39, 40] or Scribe on CAN [15]. Before describing the SplitStream design, we provide a brief overview of Pastry and Scribe.

3.1 Pastry

Pastry is a scalable, self-organizing structured peer-to-peer overlay network similar to CAN [31], Chord [36], and Tapestry [39]. In Pastry, nodes and objects are assigned random identifiers (called *nodeIds* and *keys*, respectively) from a large id space. NodeIds and keys are 128 bits long and can be thought of as a sequence of digits in base 2^b (b is a configuration parameter with a typical value of 3 or 4). Given a message and a key, Pastry routes the message to the node with the nodeId that is numerically closest to the key, which is called the key's *root*. This simple capability can be used to build higher-level services like a distributed hash table (DHT) or an application-level group communication system like Scribe.

In order to route messages, each node maintains a routing table and a leaf set. A node's routing table has about $log_{2^b} N$ rows and 2^b columns. The entries in row r of the routing table refer to nodes whose nodeIds share the first r digits with the local node's nodeId. The $(r+1)$th nodeId digit of a node in column c of row r equals c. The column in row r corresponding to the value of the $(r+1)$th digit of the local node's nodeId remains empty. At each routing step, a node normally forwards the message to a node whose nodeId shares with the key a prefix that is at least one digit longer than the prefix that the key shares with the present node's id. If no such node is known, the message is forwarded to a node whose nodeId shares a prefix with the key as long as the current node's nodeId but is numerically closer. Figure 3 shows the path of an example message.

Each Pastry node maintains a set of neighboring nodes in the nodeId space (called the leaf set), both to ensure reliable message delivery, and to store replicas of objects for fault tolerance. The expected number of routing hops is less than $log_{2^b} N$. The Pastry overlay construction observes proximity in the underlying Internet. Each routing table entry is chosen to refer to a node with low network delay, among all nodes with an appropriate nodeId prefix. As a result, one can show that Pastry routes have a *low delay penalty*: the average delay of Pastry messages is less than twice the IP delay between source and destination [11]. Similarly, one

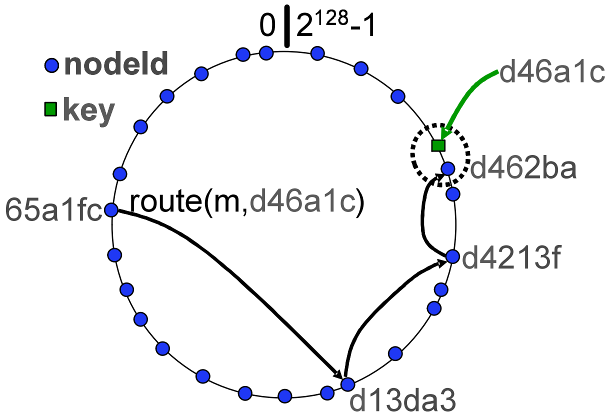


Figure 3: Routing a message from the node with nodeId $65a1fc$ to key $d46a1c$. The dots depict the nodeIds of live nodes in Pastry's circular namespace.

can show the *local route convergence* of Pastry routes: the routes of messages sent to the same key from nearby nodes in the underlying Internet tend to converge at a nearby intermediate node. Both of these properties are important for the construction of efficient multicast trees, described below. A full description of Pastry can be found in [33, 11, 12].

3.2 Scribe

Scribe [13, 14] is an application-level group communication system built upon Pastry. A pseudo-random Pastry key, known as the *groupId*, is chosen for each multicast group. A multicast tree associated with the group is formed by the union of the Pastry routes from each group member to the groupId's root (which is also the root of the multicast tree). Messages are multicast from the root to the members using reverse path forwarding [17].

The properties of Pastry ensure that the multicast trees are efficient. The delay to forward a message from the root to each group member is low due to the low delay penalty of Pastry routes. Pastry's local route convergence ensures that the load imposed on the physical network is small because most message replication occurs at intermediate nodes that are close in the network to the leaf nodes in the tree.

Group membership management in Scribe is decentralized and highly efficient, because it leverages the existing, proximity-aware Pastry overlay. Adding a member to a group merely involves routing towards the groupId until the message reaches a node in the tree, followed by adding the route traversed by the message to the group multicast tree. As a result, Scribe can efficiently support large numbers of groups, arbitrary numbers of group members, and groups with highly dynamic membership.

The latter property, combined with an anycast [14] primitive recently added to Scribe, can be used to perform distributed resource discovery. Briefly, any node in the overlay can anycast to a Scribe group by routing the message towards the groupId. Pastry's local route convergence ensures that the message reaches a group member near the message's sender with high probability. A full description and evaluation of Scribe multicast can be found in [13]. Scribe anycast is described in [14].

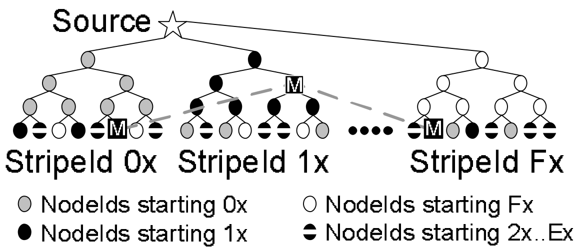


Figure 4: SplitStream's forest construction. The source splits the content and multicasts each stripe in its designated tree. Each stripe's *stripeId* starts with a different digit. The nodeIds of interior nodes share a prefix with the stripeId, thus they must be leaves in the other trees, e.g., node M with a nodeId starting with 1 is an interior node in the tree for the stripeId starting with 1 and a leaf node in other trees.

4. SPLITSTREAM DESIGN

In this section, we describe the design of SplitStream. We begin with the construction of interior-node-disjoint trees for each of the stripes. Then, we discuss how SplitStream balances the forwarding capacity across nodes, such that the bandwidth constraints of each node are observed.

4.1 Building interior-node-disjoint trees

SplitStream uses a separate Scribe multicast tree for each of the k stripes. A set of trees is said to be interior-node-disjoint if each node is an interior node in at most one tree, and a leaf node in the other trees. SplitStream exploits the properties of Pastry routing to construct interior-node-disjoint trees. Recall that Pastry normally forwards a message towards nodes whose nodeIds share progressively longer prefixes with the message's key. Since a Scribe tree is formed by the routes from all members to the groupId, the nodeIds of all interior nodes share some number of digits with the tree's groupId. Therefore, we can ensure that k Scribe trees have a disjoint set of interior nodes simply by choosing groupIds for the trees that all differ in the most significant digit. Figure 4 illustrates the construction. We call the groupId of a stripe group the stripeId of the stripe.

We can choose a value of b for Pastry that achieves the value of k suitable for a particular application. Setting $2^b = k$ ensures that each participating node has an equal chance of becoming an interior node in some tree. Thus, the forwarding load is approximately balanced. If b is chosen such that $k = 2^i$, $i < b$, it is still possible to ensure this fairness by exploiting certain properties of the Pastry routing table, but we omit the details due to space constraints. Additionally, it is fairly easy to change the Pastry implementation to route using an arbitrary base that is not a power of 2. Without loss of generality, we assume that $2^b = k$ in the rest of this paper.

4.2 Limiting node degree

The resulting forest of Scribe trees is interior-node-disjoint and satisfies the nodes' constraints on the inbound bandwidth. To see this, observe that a node's inbound bandwidth is proportional to the desired indegree, which is the number of stripes that the node chooses to receive. It is assumed that each node receives and forwards at least the stripe whose stripeId shares a prefix with its nodeId, because the node may have to serve as an interior node for that stripe.

However, the forest does not necessarily satisfy nodes' constraints on outbound bandwidth; some nodes may have more children than their forwarding capacity. The number of children that attach to a node is bounded by its indegree in the Pastry overlay, which is influenced by the physical network topology. This number may exceed a node's forwarding capacity if the node does not limit its outdegree.

Scribe has a built-in mechanism (called "push-down") to limit a node's outdegree. When a node that has reached its maximal outdegree receives a request from a prospective child, it provides the prospective child with a list of its current children. The prospective child then seeks to be adopted by the child with lowest delay. This procedure continues recursively down the tree until a node is found that can take another child. This is guaranteed to terminate successfully with a single Scribe tree provided each node is required to take on at least one child.

However, this procedure is not guaranteed to work in SplitStream. The reason is that a leaf node in one tree may be an interior node in another tree, and it may have already reached its outdegree limit with children in this other tree. Next, we describe how SplitStream resolves this problem.

4.3 Locating parents

The following algorithm is used to resolve the case where a node that has reached its outdegree limit receives a join request from a prospective child. First, the node adopts the prospective child regardless of the outdegree limit. Then, it evaluates its new set of children to select a child to reject. This selection is made in an attempt to maximize the efficiency of the SplitStream forest.

First, the node looks for children to reject in stripes whose stripeIds do not share a prefix with the local node's nodeId. (How the node could have acquired such a child in the first place will become clear in a moment.) If the prospective child is among them, it is selected; otherwise, one is chosen randomly from the set. If no such child exists, the current node is an interior node for only one stripe tree, and it selects the child whose nodeId has the shortest prefix match with that stripeId. If multiple such nodes exist and the prospective child is among them, it is selected; otherwise, one is chosen randomly from the set. The chosen child is then notified that it has been orphaned for a particular stripeId. This is exemplified in Figure 5.

The orphaned child then seeks to locate a new parent in up to two steps. In the first step, the orphaned child examines its former siblings and attempts to attach to a random former sibling that shares a prefix match with the stripeId for which it seeks a parent. The former sibling either adopts or rejects the orphan, using the same criteria as described above. This "push-down" process continues recursively down the tree until the orphan either finds a new parent or no children share a prefix match with the stripeId. If the orphan has not found a parent the second step uses the spare capacity group.

4.4 Spare capacity group

If the orphan has not found a parent, it sends an anycast message to a special Scribe group called the *spare capacity*

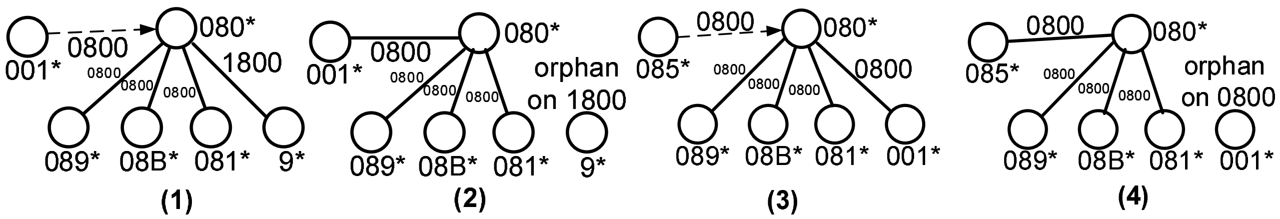


Figure 5: Handling of prospective children by a node that has reached its outdegree limit. Circles represent nodes and the numbers close to them are their nodeIds (* is a wildcard). Solid lines indicate that the bottom node is a child of the top node and the number close to the line is the stripeId. Dashed lines represent requests to join a stripe. The node with id 080* has reached its outdegree limit of 4. (1) Node 001* requests to join stripe 0800. (2) Node 080* takes 001* as a child and drops 9*, which was a child in stripe 1800 that does not share the first digit with 080*. (3) Then node 085* requests to join stripe 0800. (4) Node 080* takes 085* as a child and drops 001*, which has a shorter prefix match with stripe 0800 than other children.

group. All SplitStream nodes that have less children in stripe trees than their forwarding capacity limit are members of this group. Scribe delivers this anycast message to a node in the spare capacity group tree that is near the orphan in the physical network. This node starts a depth-first search (DFS) of the spare capacity group tree by forwarding the message to a child. If the node has no children or they have all been checked, the node checks whether it receives one of the stripes which the orphaned child seeks to receive (in general, this is the set of stripe groups that the orphan has not already joined). If so, it verifies that the orphan is not an ancestor in the corresponding stripe tree, which would create a cycle. To enable this test, each node maintains its path to the root of each stripe that it receives.

If both tests succeed, the node takes on the orphan as a child. If the node reaches its outdegree limit as a result, it leaves the spare capacity group. If one of the tests fails, the node forwards the message to its parent and the DFS of the spare capacity tree continues until an appropriate member is found. This is illustrated in Figure 6.

The properties of Scribe trees and the DFS of the spare capacity tree ensure that the parent is near the orphan in the physical network. This provides low delay and low link stress. However, it is possible for the node to attach to a parent that is already an interior node in another stripe tree. If this parent fails, it may cause the temporary loss of more than one stripe for some nodes. We show in Section 5 that only a small number of nodes and stripes are affected on average.

Anycasting to the spare capacity group may fail to locate an appropriate parent for the orphan even after an appropriate number of retries with sufficient timeouts. If the spare capacity group is empty, the SplitStream forest construction is infeasible because an orphan remains after all forwarding capacity has been exhausted. In this case, the application on the orphaned node is notified that there is no forwarding capacity left in the system.

Anycasting can fail even when there are group members with available forwarding capacity in the desired stripe. This can happen if attaching the orphan to receive the stripe from any of these members causes a cycle because the member is the orphan itself or a descendant of the orphan. We solve this problem as follows. The orphan locates any leaf in the desired stripe tree that is not its descendant. It can do this by anycasting to the stripe tree searching for a leaf that is

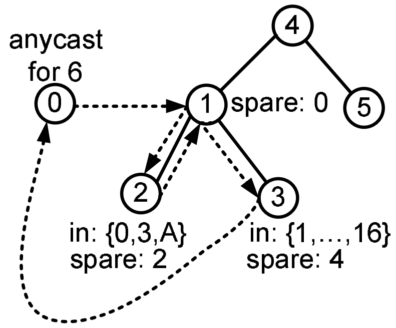


Figure 6: Anycast to the spare capacity group. Node 0 anycasts to the spare capacity group to find a parent for the stripe whose identifier starts with 6. The request is routed by Pastry towards the root of the spare capacity group until it reaches node 1, which is already in the tree. Node 1 forwards the request to node 2, one of its children. Since 2 has no children, it checks if it can satisfy node 0's request. In this case, it cannot because it does not receive stripe 6. Therefore, it sends the request back to node 1 and node 1 sends it down to its other child. Node 3 can satisfy 0's request and replies to 0.

not its descendant. Such a leaf is guaranteed to exist because we require a node to forward the stripe whose stripeId starts with the same digit as its nodeId. The orphan replaces the leaf on the tree and the leaf becomes an orphan. The leaf can attach to the stripe tree using the spare capacity group because this will not cause a cycle.

Finally, anycasting can fail if no member of the spare capacity group provides any of the desired stripes. In this case, we declare failure and notify the application. As we argue next, this is extremely unlikely to happen when the sufficient condition for forest construction (Condition 2) holds.

4.5 Correctness and complexity

Next, we argue informally that SplitStream can build a forest with very high probability provided the set of nodes N satisfies the sufficient condition for feasibility (Condition 2) and there is a modest amount of spare capacity in the system. The analysis assumes that all nodes in N join the forest at the same time and that communication is reliable to ensure that all the spare capacity in the system is available in the spare capacity group. It also assumes that nodes do not leave the system either voluntarily or due to failures. SplitStream includes mechanisms to deal with violations of each of these assumptions but these problems may block some parents that are available to forward stripes to orphans, e.g., they may be unreachable by an orphan due to communication failures. If these problems persist, SplitStream may be unable to ensure feasibility even if Condition 2 holds at every instant. We ignore these issues in this analysis but present simulation results that show SplitStream can cope with them in practice.

The construction respects the bounds on forwarding capacity and indegree because nodes reject children beyond their capacity limit and nodes do not seek to receive more stripes than their desired indegree. Additionally, there are no cycles by construction because an orphan does not attach to a parent whose path to the root includes the orphan. The issue is whether all nodes can receive as many distinct stripes as they desire.

When node i joins the forest, it selects I_i stripes uniformly at random. (The stripe whose stripeId starts with the same digit as i's nodeId is always selected but the nodeId is selected randomly with uniform probability from the id space.) Then i joins the spare capacity tree advertising that it will be able to forward the selected stripes and attempts to join the corresponding stripe trees. We will next estimate the probability that the algorithm leaves an orphan that cannot find a desired stripe.

There are two ways for a node i to acquire a parent for each selected stripe s: (1) joining the stripe tree directly without using the spare capacity group, or (2) anycasting to the spare capacity group. If s is the stripe whose stripeId starts with the same digit as i's nodeId, i is guaranteed to find a parent using (1) after being pushed down zero or more times and this may orphan another node. The algorithm guarantees that i never needs to use (2) to locate a parent on this stripe. The behavior is different when i uses (1) to locate a parent on another stripe; it may fail to find a parent but it will never cause another node to become an orphan.

When a node i first joins a stripe s, it uses (1) to find a parent. If the identifiers of i and s do not share the first digit, i may fail to find a parent for s after being pushed down at most h_s times (where h_s is the height of s's tree) but it does not cause any other node to become an orphan. If the trees are balanced we expect that h_s is $O(log|N|)$.

If the identifiers of i and s share the same digit, i is guaranteed to find a parent using (1) but it may orphan another node j on the same stripe or on a different stripe r. In this case, j attempts to use (1) to acquire a parent on the lost stripe. There are three sub-cases: (a) j looses a stripe r ($r \neq s$), (b) j looses stripe s and the identifiers of j and s do not share the first digit, and (c) j looses stripe s and the identifiers of j and s share the first digit. The algorithm ensures that in case (a), the first digit in j's nodeId does not match the first digit in r's stripeId. This ensures that this node j does not orphan any other node when it uses (1) to obtain a parent for r. Similarly, j will not orphan any other node in case (b). In cases (a) and (b), j either finds a parent or fails to find a parent after being pushed down at most h_r or h_s times. In case (c), we can view j as resuming the walk down s's tree that was started by i.

Therefore, in all cases, i's first join of stripe s results in at most one orphan j (not necessarily $i = j$) that uses anycast to find a parent for a stripe r (not necessarily $r = s$) after $O(log|N|)$ messages. This holds even with concurrent joins.

If an orphan j attempts to locate a parent for stripe r by anycasting to the spare capacity group, it may fail to find a node in the spare capacity group that receives stripe r. We call the probability of this event P_f. It is also possible that all nodes in the spare capacity group that receive r are descendants of j. Our construction ensures that the identifiers of j and r do not share the first digit. Therefore, the expected number of descendants of j for stripe r should be $O(1)$ and small if trees are well balanced. The technique that handles this case succeeds in finding a parent for j with probability one and leaves an orphan on stripe r that has no descendants for stripe r. In either case, we end up with a probability of failure P_f and an expected cost of $O(log|N|)$ messages on success.

We will compute an upper bound on P_f. We start by assuming that we know the set $1, ..., l$ of nodes in the spare capacity group when the anycast is issued and their desired indegrees $I_1, ..., I_l$. We can compute an exact value for P_f with this information:

$$P_f = (1 - \frac{I_1}{k})(1 - \frac{I_2}{k})...(1 - \frac{I_l}{k})$$

If all nodes join at least I_{min} stripes, we can compute an upper bound on P_f that does not require knowledge of the desired indegrees of nodes in the spare capacity group:

$$P_f \leq (1 - \frac{I_{min}}{k})^l$$

We can assume that each node i in the spare capacity group has spare capacity less than k; otherwise, Condition 2 implies that $I_i = k$ and i can satisfy the anycast request. Since the spare capacity in the system $C = \sum_{\forall i \in N} C_i - \sum_{\forall i \in N} I_i$ is the minimum capacity available in the spare capacity group at any time, $l \geq C/(k-1)$ and so

$$P_f \leq (1 - \frac{I_{min}}{k})^{\frac{C}{k-1}}$$

This bound holds even with concurrent anycasts because we use the minimum spare capacity in the system to compute the bound.

There are at most $|N|$ node joins and each node joins at most k stripes. Thus the number of anycasts issued during forest construction that may fail is bounded by $|N| \times k$. Since the probability of A or B occurring is less than or equal to the probability of A plus the probability of B, the following is a rough upper bound on the probability that the algorithm fails to build a feasible forest

$$|N| \times k \times (1 - \frac{I_{min}}{k})^{\frac{C}{k-1}}$$

The probability of failure is very low even with a modest amount of spare capacity in the system. For example, the predicted probability of failure is less than 10^{-11} with $|N| = 1,000,000$, $k = 16$, $I_{min} = 1$, and $C = 0.01 \times |N|$. The probability of failure decreases when I_{min} increases, for example, it is 10^{-13} in the same setting when $I_{min} = 8$ and $C = 0.001 \times |N|$. When the desired indegree of all nodes equals the total number of stripes, the algorithm never fails. In this case, it is guaranteed to build a forest if the sum of the desired indegrees does not exceed the sum of the forwarding capacities even when there is no spare capacity.

Next, we consider the algorithmic complexity of the SplitStream forest construction. The expected amount of state maintained by each node $O(log|N|)$. The expected number of messages to build the forest is $O(|N|log|N|)$ if the trees are well balanced or $O(|N|^2)$ in the worst case. We expect the trees to be well balanced if each node forwards the stripe whose identifier shares the first digit with the node's identifier to at least two other nodes.

5. EXPERIMENTAL EVALUATION

This section presents results of experiments designed to evaluate both the overhead of forest construction in SplitStream and the performance of multicasts using the forest. We ran large-scale experiments on a network simulation environment and live experiments on the PlanetLab Internet testbed [1]. The results support our hypothesis that the overhead of maintaining a forest is low and that multicasts perform well even with high churn in the system.

5.1 Experimental setup

We start by describing the experimental setup.

Network simulation: We used a packet-level, discrete-event network simulator. The simulator models the propagation delay on the physical links but it does not model queuing delay, packet losses, or cross traffic because modeling these would prevent large-scale network simulations. In the simulations we used three different network topology models: *GATech*, *Mercator* and *CorpNet*. Each topology has a set of routers and we ran SplitStream on end nodes that were randomly assigned to routers with uniform probability. Each end node was directly attached by two LAN links (one on each direction) to its assigned router. The delay on LAN links was set to 1ms.

GATech is a transit-stub topology model generated by the Georgia Tech [38] random graph generator. This model has 5050 routers arranged hierarchically. There are 10 transit domains at the top level with an average of 5 routers in each. Each transit router has an average of 10 stub domains attached, and each stub has an average of 10 routers. The link delays between routers are computed by the graph generator and routing is performed using the routing policy weights of the graph generator. We generated 10 different topologies using the same parameters but different random seeds. All *GATech* results are the average of the results obtained by running the experiment in each of these 10 topologies. We did not attach end nodes to transit routers.

Mercator is a topology model with 102,639 routers, obtained from measurements of the Internet using the Mercator system [21]. The authors of [37] used measured data and some simple heuristics to assign an autonomous system to each router. The resulting AS overlay has 2,662 nodes. Routing is performed hierarchically as in the Internet. A route follows the shortest path in the AS overlay between the AS of the source and the AS of the destination. The routes within each AS follow the shortest path to a router in the next AS of the AS overlay path. We use the number of IP hops as a proxy for delay because there is no link delay information.

CorpNet is a topology model with 298 routers and was generated using measurements of the world-wide Microsoft corporate network. Link delays between routers are the minimum delay measured over a one month period.

Due to space constraints, we present results only for the GATech topology for most experiments but we compare these results with those obtained with the other topologies.

SplitStream configuration: The experiments ran Pastry configured with $b = 4$ and a leaf set with size $l = 16$. The number of stripes per SplitStream multicast channel was $k = 2^b = 16$.

We evaluated SplitStream with six different configurations of node degree constraints. The first four are designed to evaluate the impact on overhead and performance of varying the spare capacity in the system. We use the notation $x \times y$ to refer to these configurations where x is the value of the desired indegree for all nodes and y is the forwarding capacity of all nodes. The four configurations are: 16×16, 16×18, 16×32 and $16 \times NB$. The 16×16 configuration has a spare capacity of only 16 because the roots of the stripes do not consume forwarding capacity for the stripes they originate. The 16×18 and 16×32 configurations provide a spare capacity of 12.5% and 100%, respectively. Nodes do not impose any bound on forwarding capacity in the $16 \times NB$ configuration. SplitStream is able to build a forest in any of these configurations with probability 1 (as shown by our analysis).

The last two configurations are designed to evaluate the behavior of SplitStream when nodes have different desired indegrees and forwarding capacities. The configurations are $d \times d$ and *Gnutella*. In $d \times d$, each node has a desired indegree equal to its forwarding capacity and it picks stripes to receive as follows: it picks the stripe whose stripeId has the same first digit as its nodeId with probability 1 and it picks each of the other stripes with probability 0.5. The analysis predicts a low probability of success when building a forest with the $d \times d$ configuration because there is virtually no spare capacity in the system and nodes do not receive all stripes. But SplitStream was able to build a forest successfully in 9 out of 10 runs.

The *Gnutella* configuration is derived from actual measurements of inbound and outbound bandwidth of Gnutella peers [34]. The CDF of these bandwidths is shown in Figure 7. This distribution was sampled to generate pairs of inbound (b_{in}) and outbound (b_{out}) bandwidths for each SplitStream node. Many nodes have asymmetric bandwidth as is often the case with DSL or cable modem connections. We assumed a total bandwidth for a stream of 320Kbps with 20Kbps per stripe. The desired indegree of a node was set to $\max(1, \min(b_{in}/20Kbps, 16))$ and the forwarding capacity to $\min(b_{out}/20Kbps, 32)$. Approximately, 90% of the nodes have a desired indegree of 16. The spare capacity in the system is 6.9 per node. The analysis predicts a success probability of almost 1 when building a forest with this configuration.

SplitStream Implementation: The SplitStream implementation used in the simulation experiments has three optimisations that were omitted in Section 4 for clarity.

The first optimisation reduces the overhead of cycle detection during forest construction. As described in Section 4, each node maintains the path to the root of each stripe that it receives. Maintaining this state incurs a considerable communication overhead because updates to the path are multi-

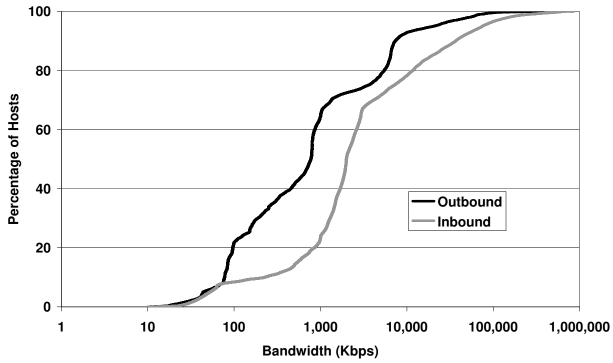


Figure 7: Cumulative distribution of bottleneck bandwidth for both inbound and outbound network links of Gnutella peers.

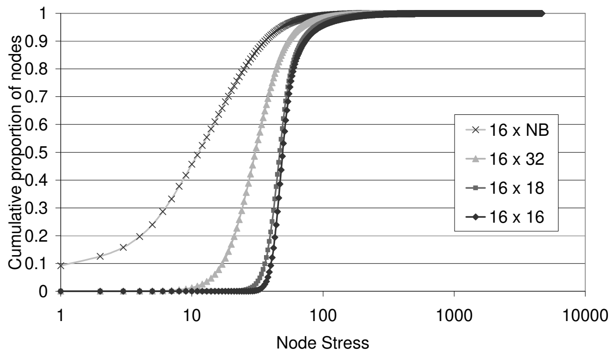


Figure 8: Cumulative distribution of node stress during forest construction with 40,000 nodes on GATech.

cast to all descendants. We can avoid maintaining this state by taking advantage of the prefix routing properties of Pastry. Cycles are possible only if some parent does not share a longer prefix with a stripeId than one of its children for that stripe. Therefore, nodes need to store and update path information only in such cases. This optimisation reduces the overhead of forest construction by up to 40%.

The second optimisation improves anycast performance. When a node joins the SplitStream forest, it may need to perform several anycasts to find a parent for different stripes. Under the optimization, these anycasts are batched; the node uses a single anycast to find parents for multiple stripes. This optimization can reduce the number of anycasts performed during forest construction by up to a factor of eight.

The third optimization improves the DFS traversal of the anycast tree. A parent adds the list of its children to an anycast message before forwarding the message to a child. If the child is unable to satisfy the anycast request, it removes itself from the list and sends the message to one of its siblings (avoiding another visit to the parent).

5.2 Forest construction overhead

The first set of experiments measured the overhead of forest construction without node failures. They started from a Pastry overlay with 40,000 nodes and built a SplitStream forest with all overlay nodes. All the nodes joined the spare capacity and the stripe groups at the same time. The overheads would be lower with less concurrency.

We used two metrics to measure the overhead: *node stress* and *link stress*. Node stress quantifies the load on nodes. A node's stress is equal to the number of messages that it receives. Link stress quantifies the load on the network. The stress of a physical network link is equal to the number of messages sent over the link.

Node stress: Figures 8 and 9 show the cumulative distribution of node stress during forest construction with different configurations on the GATech topology. Figure 8 shows results for the $16 \times y$ configurations and Figure 9 shows results for $d \times d$ and *Gnutella*. A point (x, y) in the graph indicates that a fraction y of all the nodes in the topology has node stress less than or equal to x. Table 1 shows the maximum, mean and median node stress for these distributions. The results were similar on the other topologies.

Figure 8 and Table 1 show that the node stress drops as the spare capacity in the system increases. With more spare capacity, nodes are orphaned less often and so there are less pushdowns and anycasts. The $16 \times NB$ configuration has the lowest node stress because there are no pushdowns or anycasts. The 16×16 configuration has the highest node stress because each node uses an anycast to find a parent for 8 stripes on average. The nodes with the maximum node stress in all configurations (other than $16 \times NB$) are those with nodeIds closest to the identifier of the spare capacity group. Table 1 shows that increasing the spare capacity of the 16×16 configuration by only 12.5% results in a factor of 2.7 decrease in the maximum node stress.

Conf.	16×16	16×18	16×32	$16 \times NB$	$d \times d$	Gnut.
Max	2971	1089	663	472	2532	1054
Mean	57.2	52.6	35.3	16.9	42.1	56.7
Median	49.9	47.4	30.9	12	36.6	54.2

Table 1: Maximum, mean and median node stress during forest construction with 40,000 nodes on GATech.

Figure 9 shows that the node stress is similar with *Gnutella* and 16×16. *Gnutella* has a significant amount spare capacity but not all members of the spare capacity group receive all stripes, which increases the length of the DFS traversals of the anycast tree. $d \times d$ has the same spare capacity as 16×16 but it has lower node stress because nodes join only 9 stripe groups on average instead of 16.

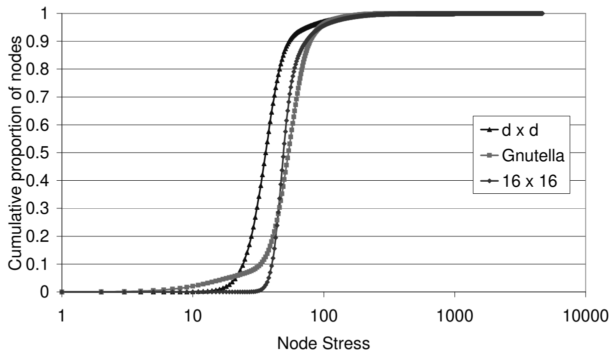


Figure 9: Cumulative distribution of node stress during forest construction with 40,000 nodes on GATech.

We also ran experiments to evaluate the node stress during forest construction for overlays with different sizes. Figure 10 shows the mean node stress with the $16 \times x$ configurations. The maximum and median node stress show similar trends as the number of nodes increases.

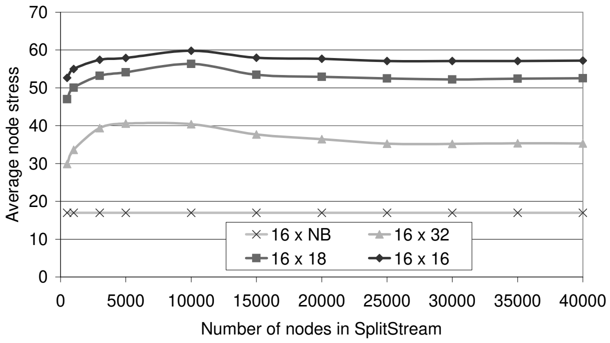


Figure 10: Mean node stress during forest construction with varying number of nodes on GATech.

From the analysis, we would expect the mean node stress to be $O(logn)$, where n is the number of nodes. But there is a second effect that is ignored in the analysis; the average cost of performing an anycast and joining a stripe group decreases with the group size. The first effect dominates when n is small and the second effect compensates the first for large n. This suggests that our complexity analysis is too pessimistic.

The results demonstrate that the node stress during forest construction is low and is largely independent of the number of nodes in the forest. It is interesting to contrast SplitStream with a centralized server distributing an s-byte file to a large number of clients. With 40,000 clients, the server will have a node stress of 40,000 handling requests and will send a total of $s \times 40,000$ bytes. The maximum node stress during SplitStream forest construction with the 16×16 configuration is 13.5 times lower, and the number of bytes sent by each node to multicast the file over the forest is bounded by s.

Link stress: Figure 11 shows the cumulative distribution of link stress during forest construction with different configurations on GATech. A point (x, y) in the graph indicates that a fraction y of all the links in the topology has link stress less than or equal to x. Table 2 shows the maximum, mean and median link stress for links with non-zero link stress. It also shows the fraction of links with non-zero link stress (*links*). Like node stress, the link stress drops as the spare capacity in the system increases and the reason is the same. The links with the highest stress are transit links.

The results were similar on the other topologies: the medians were almost identical and the averages were about 10% lower in CorpNet and 10% higher in Mercator. The difference in the averages is explained by the different ratio of the number of LAN links to the number of router-router links in the topologies. This ratio is highest in CorpNet and it is lowest in Mercator.

We conclude that the link stress induced during forest construction is low on average. The maximum link stress across all configurations is at least 6.8 times lower than the maximum link stress in a centralized system with the same number of nodes.

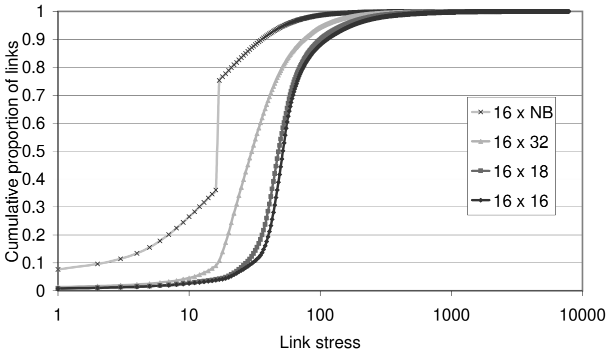


Figure 11: Cumulative distribution of link stress during forest construction with 40,000 nodes on GATech.

Conf.	16×16	16×18	16×32	$16 \times NB$	$d \times d$	Gnut.
Max	5893	4285	2876	1804	5405	5058
Mean	74.1	65.2	43.6	21.2	57.5	70.22
Med.	52.6	48.8	30.8	17	38	53
Links	.99	.99	.99	.94	.99	.99

Table 2: Maximum, mean, and median link stress for used links and fraction of links (*Links*) used during forest construction with 40,000 nodes on GATech.

5.3 Forest multicast performance

We also ran experiments to evaluate the performance of multicast over a SplitStream forest without failures. To put this performance in perspective, we compared SplitStream, IP multicast, and Scribe. IP multicast represents the best that can be achieved with router support and Scribe is a state-of-the-art, application-level multicast system that uses a single tree. We used three metrics to evaluate multicast performance in the comparison: node stress, link stress, and delay penalty relative to IP.

The experiments ran on the 40,000-node Pastry overlay that we described in the previous section. Our implementation of IP multicast used a shortest path tree formed by the merge of the unicast routes from the source to each recipient. This is similar to what could be expected in our experimental setting using protocols like Distance Vector Multicast Routing Protocol (DVMRP) [18]. We created a single group for IP multicast and Scribe that all overlay nodes joined. The tree for the Scribe group was created without the bottleneck remover [13].

To evaluate SplitStream, we multicast a data packet to each of the 16 stripe groups of a forest with 40,000 nodes (built as described in the previous section). For both Scribe and IP multicast, we multicast 16 data packets to the single group.

Node stress: During this multicast experiment, the node stress of each SplitStream node is equal to its desired indegree, for example, 16 in the $16 \times x$ configurations or less in others. Scribe and IP multicast also have a node stress of 16. The number of messages sent by each node in SplitStream is also bounded by the forwarding capacity of each node, for example, it is 16 in the 16×16 configuration. This is important because it enables nodes with different capabilities to participate in the system.

Link stress: We start by presenting results of experiments that measured link stress during multicast with different SplitStream forest sizes. Table 3 shows the results of these experiments with the 16×16 configuration in GATech. When a SplitStream node is added to the system, it uses two additional LAN links (one on each direction) and it induces a link stress of 16 in both. Adding nodes also causes an increase on the link stress of router-router links because the router network remains fixed. Since the majority of links for the larger topologies are LAN links, the median link stress remains constant and the mean link stress grows very slowly. The maximum link stress increases because the link stress in router-router links increases. This problem affects all application-level multicast systems.

Num.	500	1k	3k	10k	20k	30k	40k
Mean	17.6	18.3	17.7	17.9	18.7	20.22	20.4
Med.	16	16	16	16	16	16	16
Max	131	233	509	845	1055	1281	1411
Links	0.13	0.23	0.52	0.89	0.96	0.97	0.98

Table 3: Maximum, mean and median link stress for used links and fraction of links (*Links*) used during multicast with the 16×16 configuration on GATech with varying number of nodes.

The next set of experiments compared the link stress during multicast with different SplitStream configurations and 40,000 nodes on GATech. Table 4 shows the maximum, mean and median link stress for used links, and the fraction of links used in these experiments. The results show that the link stress tends to decrease when the spare capacity increases. However, the absence of bounds on forwarding capacity in $(16 \times NB)$ causes a concentration of stress in a smaller number of links, which results in increased average and maximum stress for used links. The average link stress in $d \times d$ and *Gnutella* is lower because nodes receive less stripes on average.

Conf.	16×16	16×18	16×32	$16 \times NB$	$d \times d$	Gnut.
Max	1411	1124	886	1616	982	1032
Mean	20.5	19	19	20	11.7	18.2
Med.	16	16	16	16	9	16
Links	.98	.98	.97	.94	.97	.97

Table 4: Maximum, mean, and median link stress for used links and fraction of links (*Links*) used during multicast with different SplitStream configurations and 40,000 nodes on GATech.

We picked the SplitStream configuration that performs worst (16×16) and compared its performance with Scribe, IP multicast, and a centralized system using unicast. Figure 12 shows the cumulative distribution of link stress during multicast for the different systems on GATech with 40,000 nodes. A point (x, y) in the graph indicates that a fraction y of all the links in the topology has link stress less than or equal to x. Table 5 shows the maximum, mean and median link stress for used links, and the fraction of links used.

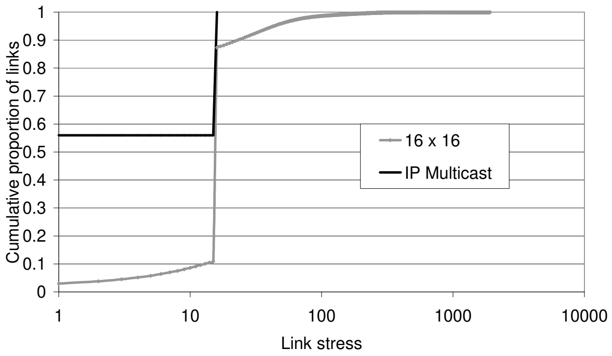
The results show that SplitStream uses a significantly larger fraction of the links in the topology to multicast messages than any of the other systems: SplitStream uses 98% of the links in the topology, IP multicast and the centralized unicast use 43%, and Scribe uses 47%. This is mostly because SplitStream uses both outbound and inbound LAN links for all nodes whereas IP multicast and the centralized unicast only use inbound LAN links. Scribe uses all inbound LAN links but it only uses outbound LAN links for a small fraction of nodes (the interior nodes in the tree). We observed the same behavior on the other topologies.

SplitStream loads the links it uses less than Scribe. Table 5 shows that the average stress in links used by SplitStream is close to the average stress in links used by IP multicast (28% worse). The average link stress in links used by Scribe is 247% worse than that achieved by IP multicast. SplitStream also achieves a maximum link stress that is almost a factor of 3 lower than Scribe's. The comparison would be even more favorable to SplitStream with a configuration with more spare capacity.

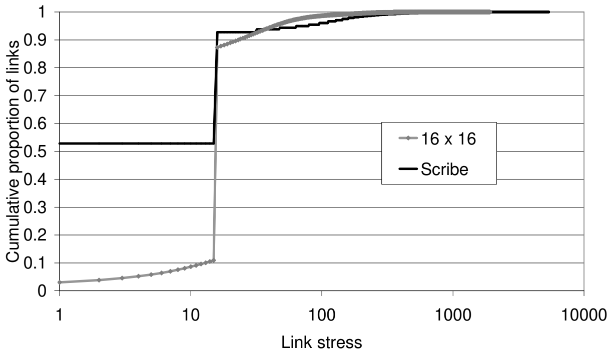
We also observed this behavior in the other topologies. The average stress in links used by SplitStream is at most 13% worse than the average stress in links used by IP mul-

Conf.	centralized	Scribe	IP	16×16
Max	639984	3990	16	1411
Mean	128.9	39.6	16	20.5
Median	16	16	16	16
Links	.43	.47	.43	.98

Table 5: Maximum, mean, and median link stress for used links and fraction of links (*Links*) used by centralized unicast, Scribe, IP, and SplitStream multicast with 40,000 nodes on GATech.



(a) SplitStream vs IP



(b) SplitStream vs Scribe

Figure 12: Cumulative distribution of link stress during multicast with 40,000 nodes on GATech.

ticast on CorpNet and at most 37% worse in Mercator. The average link stress in links used by Scribe is 203% worse than that achieved by IP multicast in CorpNet and 337% worse in Mercator.

We conclude that SplitStream can deliver higher bandwidth than Scribe by using available bandwidth in node access links in both directions. This is a fundamental advantage of SplitStream relative to application level multicast systems that use a single tree.

Delay penalty: Next, we quantify the delay penalty of SplitStream relative to IP multicast. We computed two metrics of delay penalty for each SplitStream stripe: RMD and RAD. To compute these metrics, we measured the distribution of delays to deliver a message to each member of a stripe group using both the stripe tree and IP multicast. RMD is the ratio between the maximum delay using the stripe tree and the maximum delay using IP multicast, and RAD is the ratio between the average delay using the stripe tree and the average delay using IP multicast.

Figures 13 and 14 show RAD results for SplitStream multicast with different configurations with 40,000 nodes. They present the cumulative RAD distribution for the 16 stripes. A point (x, y) in the graph indicates that a fraction y of the 16 stripes has RAD less than or equal to y. The results for RMD were qualitatively similar and the maximum RMD for the configurations in the two figures was 5.4 in GATech and 4.6 in CorpNet.

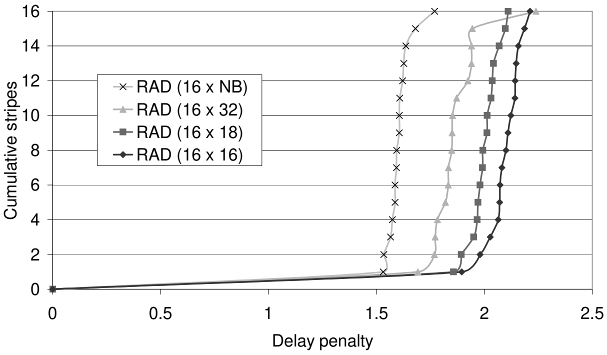
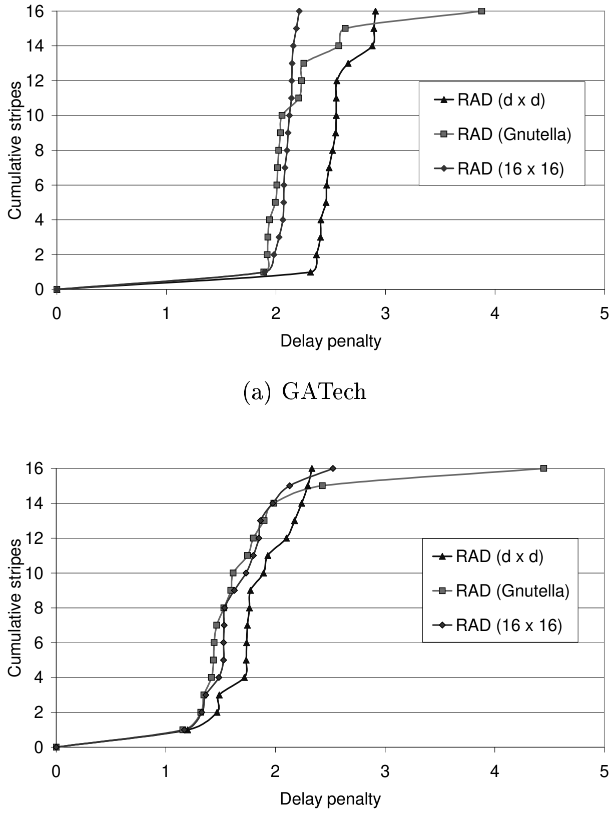


Figure 13: Cumulative distribution of delay penalty with $16 \times x$ configurations on the GATech topology with 40,000 nodes.

Figure 13 shows that the delay penalty increases when the spare capacity in the system decreases. The $16 \times NB$ configuration provides the lowest delay because nodes are never orphaned. The mechanisms to deal with orphaned nodes (push down and anycast) increase the average depth of the stripe trees. Additionally, the bounds on outdegree may prevent children from attaching to the closest prospective parent in the network. To put these results into context, if the stripe trees were built ignoring physical network proximity, the average RAD would be approximately 3.8 for the $16 \times NB$ configuration and significantly higher for the other configurations.

Figure 14 shows that the RAD distribution is qualitatively similar in GATech and CorpNet for three representative configurations. We do not present results for Mercator because we do not have delay values for the links in this topology.

The delay penalty tends to be lower in CorpNet because



(a) GATech

(b) CorpNet

Figure 14: Cumulative distribution of delay penalty on GATech and CorpNet with 40,000 nodes.

Pastry's mechanism to exploit proximity in the network is more effective in CorpNet than in GATech [12]. The figure also shows that the delay penalty with *Gnutella* and with 16×16 is similar for most stripes. However, the variance is higher in *Gnutella* because nodes with a very small forwarding capacity can increase the average depth of a stripe tree significantly if they are close to the top of the tree. The delay penalty is higher with $d \times d$ than with the other two configurations because the average forwarding capacity is lower (only 9), which results in deeper stripe trees.

We conclude that SplitStream achieves delay that is within a small factor of optimal. This performance should be sufficient even for demanding applications that have an interactive component, e.g., large virtual class rooms.

5.4 Resilience to node failures

Previous experiments evaluated the performance of SplitStream without failures. This section presents results to quantify performance and overhead with node failures. The performance metric is the fraction of the desired stripes received by each node, and the overhead metric is the number of control messages sent per second by each node (similar to node stress).

Path diversity: The first set of experiments analyzed the paths from each node i to the root of each stripe that i receives. If all these paths are node-disjoint, a single node failure can deprive i of at most one stripe (until the stripe tree repairs). SplitStream may fail to guarantee node-disjoint paths when nodes use anycast to find a parent but this is not a problem as the next results show.

Conf.	16×16	16×32	$d \times d$	$16 \times NB$	Gnut.
Max	6.8	6.6	5.3	1	7.2
Mean	2.1	1.7	1.5	1	2.4
Median	2	2	1	1	2

Table 6: Worst case maximum, mean, and median number of stripes lost at each node when a single node fails.

In a 40,000-node SplitStream forest on GATech, the mean and median number of lost stripes when a random node fails is 1 for all configurations. However, nodes may loose more than one stripe when some nodes fail. Table 6 shows the max, median, and mean number of stripes lost by a node when its *worst case* ancestor fails. The number of stripes lost is very small for most nodes, even when the worst case ancestor fails. This shows that SplitStream is very robust to node failures.

Catastrophic failures: The next experiment evaluated the resilience of SplitStream to catastrophic failures. We created a 10,000-node SplitStream forest with the 16×16 configuration on GATech and started multicasting data at the rate of one packet per second per stripe. We failed 2,500 nodes 10s into the simulation.

Both Pastry and SplitStream use heartbeats and probes to detect node failures. Pastry used the techniques described in [25] to control probing rates; it was tuned to achieve 1% loss rate with a leaf set probing period of 30s. SplitStream nodes send heartbeats to their children and to their parents. The heartbeats sent to parents allow nodes to detect when a child fails so they can rejoin the spare capacity group. We configured SplitStream nodes to send these hearbeats every 30s. In both Pastry and SplitStream, heartbeats and probes are suppressed by other traffic.

Figure 15 shows the maximum, average, and minimum number of stripes received by each node during the experiment. Nodes loose a large number of stripes with the large scale failure but SplitStream and Pastry recover quickly. Most nodes receive packets on at least 14 stripes after 30s (one failure detection period) and they receive all stripes after 60s. Furthermore, all nodes receive all stripes after less than 3 minutes.

Figure 16 breaks down the average number of messages sent per second per node during the experiment. The line labeled *Pastry* shows the Pastry overhead and the line labeled *Pastry+SplitStream* represents the total overhead, i.e., the average number of Pastry and SplitStream control messages. The figure also shows the total number of messages including data packets (labeled *Pastry+SplitStream+Data*).

The results show that nodes send only 1.6 control messages per second before the failure. This overhead is very low.

The overhead increases after the failure while Pastry and SplitStream repair. Even after repair, the overhead is higher than before the failure because Pastry uses a self-tuning mechanism to adapt routing table probing rates [25]. This mechanism detects a large number of failures in a short period and increases the probing rate (to the maximum in this case). Towards the end of the trace, the overhead starts to dip because the self-tuning mechanism forgets the failures it registered.

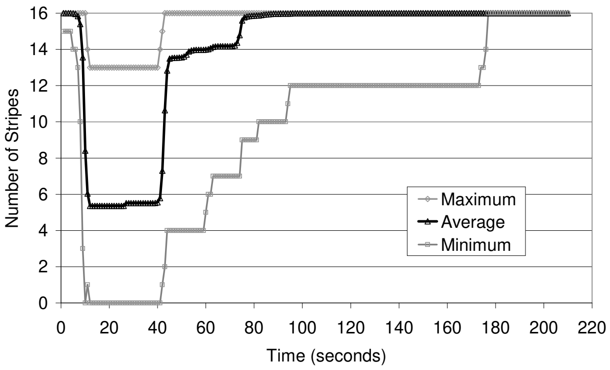


Figure 15: Maximum, average, and minimum number of stripes received when 25% out of 10,000 nodes fail on GATech.

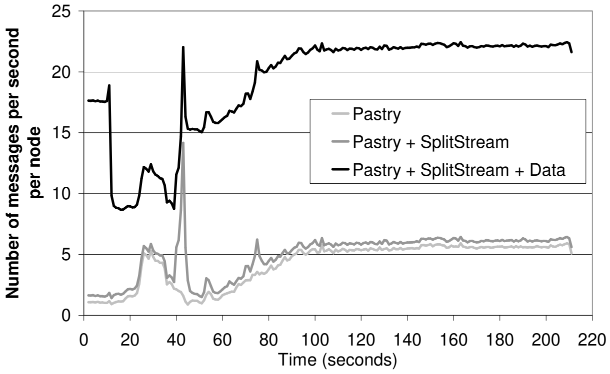


Figure 16: Number of messages per second per node when 25% out of 10,000 nodes fail on GATech.

High churn: The final simulation experiment evaluated the performance and overhead of SplitStream with high churn in the overlay. We used a real trace of node arrivals and departures from a study of Gnutella [34]. The study monitored 17,000 unique nodes in the Gnutella overlay over a period of 60 hours. It probed each node every seven minutes to check if it was still part of the Gnutella overlay. The average session time over the trace was approximately 2.3 hours and the number of active nodes in the overlay varied between 1300 and 2700. Both the arrival and departure rate exhibit large daily variations.

We ran the experiment on the GATech topology. Nodes joined a Pastry overlay and failed according to the trace. Twenty minutes into the trace, we created a SplitStream forest with all the nodes in the overlay at that time. We used the 16×20 configuration. From that time on, new nodes joined the Pastry overlay and then joined the SplitStream forest. We sent a data packet to each stripe group every 10 seconds. Figure 17 shows the average and 0.5th percentile of the number of stripes received by each node over the trace.

The results show that SplitStream performs very well even under high churn: 99.5% of the nodes receive at least 75% of the stripes (12) almost all the time. The average number

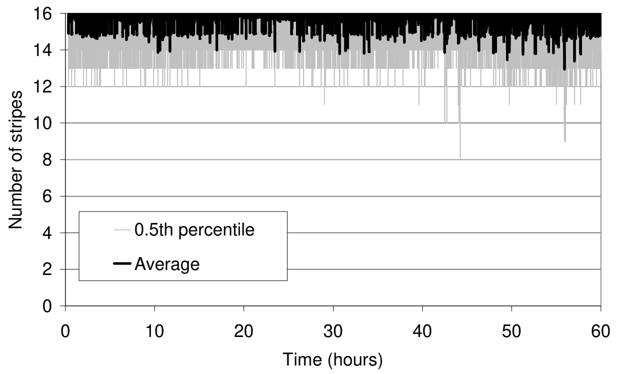


Figure 17: Number of stripes received with the Gnutella trace on GATech.

of stripes received by each node is between 15 and 16 most of the time.

Figure 18 shows the average number of messages per second per node in the Gnutella trace. The results show that nodes send less than 1.6 control messages per second most of the time. The amount of control traffic varies mostly because Pastry's self-tuning mechanism adjusts the probing rate to match the observed failure rate. The number of control messages never exceeds 4.5 per second per node in the trace. Additionally, the vast majority of these messages are probes or heartbeats, which are small (50B on the wire). Therefore, the overhead is very low, for example, control traffic never consumes more than 0.17% of the total bandwidth used with a 1Mb/s data stream.

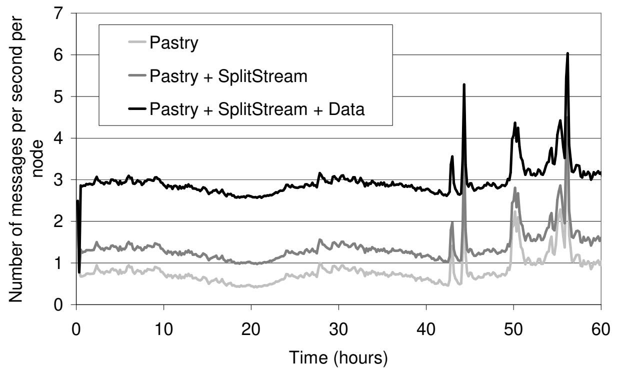


Figure 18: Number of messages per second per node with the Gnutella trace on GATech.

5.5 PlanetLab results

Finally, we present results of live experiments that evaluated SplitStream in the PlanetLab Internet testbed [1]. The experiments ran on 50 hosts at different PlanetLab sites throughout the US and Europe. Each host ran a single SplitStream node, which resulted in a total of 50 nodes in the SplitStream forest. We used the 16×16 configuration with a data stream of 2.5Mbits/sec. A 160Kbit packet with a sequence number was multicast every second on each stripe. Between sequence numbers 279 and 284, five hosts were randomly selected and the SplitStream nodes running on them were killed. This caused a number of SplitStream nodes to loose packets from one or more stripes while the affected stripe trees were repaired.

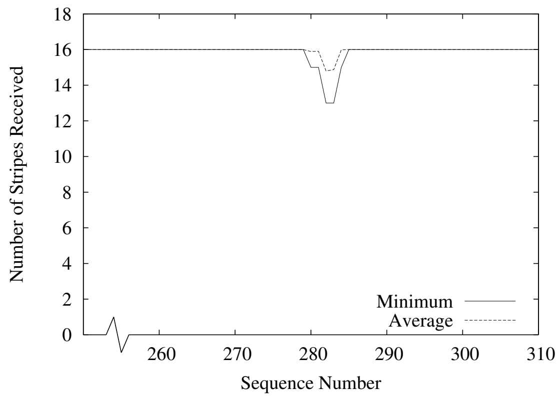


Figure 19: Number of stripes received versus sequence number.

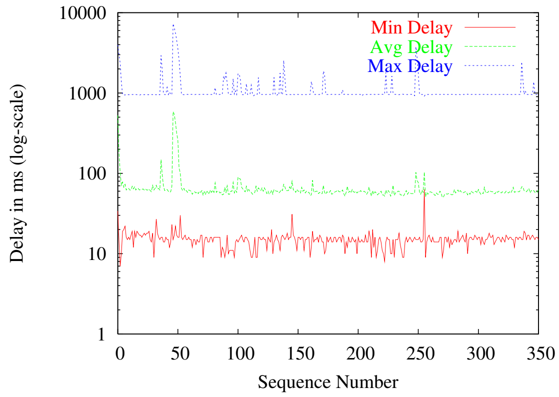


Figure 20: Packet delays versus sequence number.

Figure 19 shows the average and minimum number of stripe packets received by any live SplitStream node with a particular sequence number. The effect of the failed nodes can be clearly seen. Since nodes use TCP as the transport protocol, nodes immediately know if the connected node was killed as TCP stream is broken. As can be seen in Figure 19, once a failure has been detected SplitStream is able to find a new parent rapidly, i.e. within 2-3 seconds.

Figure 20 shows the minimum, average and maximum delay for packets received with each sequence number at all SplitStream nodes. The delay includes the time taken to send the packet from a source to the root of each stripe through Pastry. Finally, Figure 21 shows the CDF of the delays experienced by all packets received independent of sequence number and stripe.

The delay spikes in Figure 20 are caused by congestion losses in the Internet[1]. However, Figure 21 shows that 98% of packets experience a delay of under 1 second. Our results show that SplitStream works as expected in an actual deployment, and that it recovers from failures gracefully.

[1] TCP is used as the transport protocol; therefore, lost packets cause retransmission delays. A transport protocol that does not attempt to recover packet losses could avoid these delays.

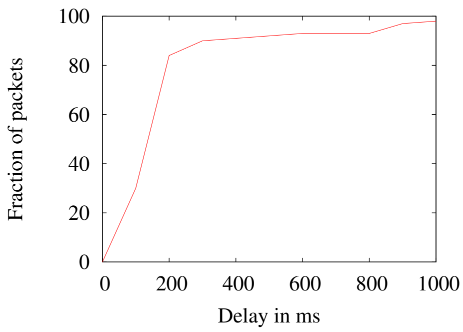


Figure 21: Cumulative distribution of packet delays. 2% of the packets experience a delay of greater then 1 second and the maximum observed delay is 7.3 seconds.

6. RELATED WORK

Many application-level multicast systems have been proposed [16, 22, 32, 40, 13, 6, 23]. All are based on a single multicast tree per sender. Several systems use end-system multicast for video distribution, notably Overcast [22] and SpreadIt [7]. SplitStream differs in that it distributes forwarding load over all participants using multiple multicast trees, thereby reducing the bandwidth demands on individual peers.

Overcast organizes dedicated servers into a source-rooted multicast tree using bandwidth estimation measurements to optimize bandwidth usage across the tree. The main differences between Overcast and SplitStream are *(i)* that Overcast uses dedicated servers while SplitStream utilises clients; *(ii)* Overcast creates a single bandwidth optimised multicast tree, whereas SplitStream assumes that the network bandwidth available between peers is limited by their connections to their ISP rather than the network backbone. This scenario is typical because most backbone links are over provisioned.

Nguyen and Zakhor [29] propose streaming video from multiple sources concurrently, thereby exploiting path diversity and increasing tolerance to packet loss. They subsequently extend the work in [29] to use Forward Error Correction [9] encodings. The work assumes that the client is aware of the set of servers from which to receive the video. SplitStream constructs multiple multicast trees in a decentralized fashion and is therefore more scalable.

Apostolopoulos [4, 5] originally proposed utilising striped video and MDC to exploit path diversity for increased robustness to packet loss. They propose building an overlay composed of relays, and having each stripe delivered to the client using a different source. The work examines the performance of the MDC, but does not describe an infrastructure to actually forward the stripes to the clients. CoopNet [30] implements such a hybrid system, which utilises multiple trees and striped video using MDC. When the video server is overloaded, clients are redirected to other clients, thereby creating a distribution tree routed at the server. There are two fundamental differences between CoopNet and SplitStream: *(i)* CoopNet uses a centralised algorithm (running on the server) to build the trees while SplitStream is completely decentralised; and *(ii)* CoopNet does not attempt to manage the bandwidth contribution of individual nodes. However, it is possible to add this capability to CoopNet.

FCast [20] is a reliable file transfer protocol based on IP multicast. It combines a Forward Error Correction [9] encoding and a data carousel mechanism. Instead of relying on IP multicast, FCast could be easily built upon SplitStream, for example, to provide software updates cooperatively.

In [10, 26], algorithms and content encodings are described that enable parallel downloads and increase packet loss resilience in richly connected, collaborative overlay networks by exploiting downloads from multiple peers.

7. CONCLUSIONS

We presented the design and evaluation of SplitStream, an application-level multicast system for high-bandwidth data dissemination in cooperative environments. The system stripes the data across a forest of multicast trees to balance forwarding load and to tolerate faults. It is able to distribute the forwarding load among participating nodes while respecting individual node bandwidth constraints. When combined with redundant content encoding, SplitStream offers resilience to node failures and unannounced departures, even while the affected multicast tree is repaired. The overhead of forest construction and maintenance is modest and well balanced across nodes and network links, even with high churn. Multicasts using the forest do not load nodes beyond their bandwidth constraints and they distribute the load more evenly over the network links than application-level multicast systems that use a single tree.

The simulator and the versions of Pastry and SplitStream that ran on top of it are available upon request from Microsoft Research. The version of Pastry and SplitStream that ran on PlanetLab is available for download from: `http://www.cs.rice.edu/CS/Systems/Pastry/FreePastry`

Acknowledgements

We thank Manuel Costa, Alan Mislove and Ansley Post for useful discussions and help with implementation and experimentation; Steven Gribble, Krishna Gummadi and Stefan Saroiu for the data from [34]; and Hongsuda Tangmunarunkit, Ramesh Govindan and Scott Shenker for the Mercator topology ([37]). We would also like to thank Mayank Bawa, Amin Vahdat (our shepherd) and the anonymous reviewers for their comments on earlier drafts.

8. REFERENCES

[1] Planetlab. http://www.planet-lab.org.

[2] E. Adar and B. Huberman. Free riding on Gnutella. *First Monday*, 5(10), Oct. 2000. `http://firstmonday.org/issues/issue5_10/adar/index.html`.

[3] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient overlay networks. In *SOSP'01*, Banff, Canada, Dec. 2001.

[4] J. G. Apostolopoulos. Reliable video communication over lossy packet networks using multiple state encoding and path diversity. In *Visual Communications and Image Processing*, Jan. 2001.

[5] J. G. Apostolopoulos and S. J. Wee. Unbalanced multiple description video communication using path diversity. In *IEEE International Conference on Image Processing*, Oct. 2001.

[6] S. Banerjee, B. Bhattacharjee, and C. Kommareddy. Scalable application layer multicast. In *Proceedings of ACM SIGCOMM*, Aug. 2002.

[7] M. Bawa, H. Deshpande, and H. Garcia-Molina. Transience of peers and streaming media. In *HotNets-I*, New Jersey, USA, Oct. 2002.

[8] K. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Transactions on Computer Systems*, 17(2):41–88, May 1999.

[9] R. Blahut. *Theory and Practice of Error Control Codes.* Addison Wesley, MA, 1994.

[10] J. Byers, J. Considine, M. Mitzenmacher, and S. Rost. Informed content delivery across adaptive overlay networks. In *SIGCOMM'2002*, Pittsburgh, PA, USA, Aug. 2002.

[11] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron. Exploiting network proximity in peer-to-peer overlay networks. Technical Report MSR-TR-2002-82, Microsoft Research, 2002.

[12] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron. Proximity neighbor selection in tree-based structured peer-to-peer overlays. Technical Report MSR-TR-2003-52, Microsoft Research, Aug. 2003.

[13] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE JSAC*, 20(8), Oct. 2002.

[14] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. Scalable application-level anycast for highly dynamic groups. In *Networked Group Communications*, Oct. 2003.

[15] M. Castro, M. Jones, A.-M. Kermarrec, A. Rowstron, M. Theimer, H. Wang, and A. Wolman. An evaluation of scalable application-level multicast built using peer-to-peer overlay networks. In *INFOCOM'03*, 2003.

[16] Y. Chu, S. Rao, and H. Zhang. A case for end system multicast. In *Proc. of ACM Sigmetrics*, pages 1–12, June 2000.

[17] Y. K. Dalal and R. Metcalfe. Reverse path forwarding of broadcast packets. *Communications of the ACM*, 21(12):1040–1048, 1978.

[18] S. Deering and D. Cheriton. Multicast routing in datagram internetworks and extended LANs. *ACM Transactions on Computer Systems*, 8(2), May 1990.

[19] P. Eugster, S. Handurukande, R. Guerraoui, A.-M. Kermarrec, and P. Kouznetsov. Lightweight probabilistic broadcast. In *Proceedings of The International Conference on Dependable Systems and Networks (DSN 2001)*, July 2001.

[20] J. Gemmell, E. Schooler, and J. Gray. Fcast multicast file distribution. *IEEE Network*, 14(1):58–68, Jan 2000.

[21] R. Govindan and H. Tangmunarunkit. Heuristics for internet map discovery. In *Proc. 19th IEEE INFOCOM*, pages 1371–1380, Tel Aviv, Israel, March 2000. IEEE.

[22] J. Jannotti, D. Gifford, K. Johnson, M. Kaashoek, and J. O'Toole. Overcast: Reliable multicasting with an overlay network. In *Proc. OSDI 2000*, San Diego, CA, 2000.

[23] D. Kostic, A. Rodriguez, J. Albrecht, A. Bhirud, and A. Vahdat. Using random subsets to build scalable network services. In *USITS'03*, Mar. 2003.

[24] M. Luby. LT Codes. In *FOCS 2002*, Nov. 2002.

[25] R. Mahajan, M. Castro, and A. Rowstron. Controlling the cost of reliability in peer-to-peer overlays. In *IPTPS'03*, Feb. 2003.

[26] P. Maymounkov and D. Mazières. Rateless Codes and Big Downloads. In *IPTPS'03*, Feb. 2003.

[27] A. Mohr, E. Riskin, and R. Ladner. Unequal loss protection: Graceful degredation of image quality over packet erasure channels through forward error correction. *IEEE JSAC*, 18(6):819–828, June 2000.

[28] T. Ngan, P. Druschel, and D. S. Wallach. Enforcing fair sharing of peer-to-peer resources. In *IPTPS '03*, Berkeley, CA, Feb. 2003.

[29] T. Nguyen and A. Zakhor. Distributed video streaming with forward error correction. In *Packet Video Workshop*, Pittsburgh, USA., 2002.

[30] V. Padmanabhan, H. Wang, P. Chou, and K. Sripanidkulchai. Distributing streaming media content using cooperative networking. In *The 12th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV '02)*, Miami Beach, FL, USA, May 2002.

[31] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. ACM SIGCOMM'01*, San Diego, CA, Aug. 2001.

[32] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Application-level multicast using content-addressable networks. In *NGC'2001*, Nov. 2001.

[33] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. IFIP/ACM Middleware 2001*, Heidelberg, Germany, Nov. 2001.

[34] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of the Multimedia Computing and Networking (MMCN)*, San Jose, CA, Jan. 2002.

[35] A. Snoeren, K. Conley, and D. Gifford. Mesh-based content routing using XML. In *SOSP'01*, Banff, Canada, Dec. 2001.

[36] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. ACM SIGCOMM'01*, San Diego, CA, Aug. 2001.

[37] H. Tangmunarunkit, R. Govindan, D. Estrin, and S. Shenker. The impact of routing policy on internet paths. In *Proc. 20th IEEE INFOCOM*, Alaska, USA, Apr. 2001.

[38] E. Zegura, K. Calvert, and S. Bhattacharjee. How to model an internetwork. In *INFOCOM96*, San Francisco, CA, 1996.

[39] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-resilient wide-area location and routing. Technical Report UCB//CSD-01-1141, U. C. Berkeley, April 2001.

[40] S. Zhuang, B. Zhao, A. Joseph, R. Katz, and J. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *NOSSDAV'2001*, June 2001.