Chainsaw: Eliminating Trees from Overlay Multicast

Vinay Pai, Kapil Kumar, Karthik Tamilmani, Vinay Sambamurthy, Alexander E. Mohr
Department of Computer Science
Stony Brook University
{vinay,kkumar,tamilman,vsmurthy,amohr}@cs.stonybrook.edu

Abstract

In this paper, we present Chainsaw, a p2p overlay multicast system that completely eliminates trees. Peers are notified of new packets by their neighbors and must explicitly request a packet from a neighbor in order to receive it. This way, duplicate data can be eliminated and a peer can ensure it receives all packets. We show with simulations that Chainsaw has a short startup time, good resilience to catastrophic failure and essentially no packet loss. We support this argument with real-world experiments on Planetlab and compare Chainsaw to Bullet and Splitstream using MACEDON.

1 Introduction

A common approach taken by peer-to-peer (p2p) multicast networks is to build a routing tree rooted at the sender. The advantage of a tree-based topology is that once the tree is built, routing decisions are simple and predictable—a node receives data from its parent and forwards it to its children. This tends to minimize both delay and jitter (variation in delay).

However, there are disadvantages to a tree-based approach. Since nodes depend on their parent to deliver data to them, any data loss near the root node affects every node below it. Moreover, whenever a node other than a leaf node leaves the system, the tree must be quickly repaired to prevent disruption. Another disadvantage of a tree is that interior nodes are responsible for fanning out data to all of their children, while the leaf nodes do not upload at all.

Another common feature of p2p multicast systems is that they are *push-based*, i.e. they forward data based on some routing algorithm without explicit requests from the recipient. A purely push-based system can't recover from lost transmissions easily. Moreover, if there are multiple senders to a given node, there is a chance that the node will receive duplicate data, resulting in wasted bandwidth.

In a *pull-based* system, data is sent to nodes only in response to a request for that packet. As a result, a node can easily recover from packet loss by re-requesting lost packets. Moreover, there is no need for global routing algorithms, as nodes only need to be aware of what packets their neighbors have.

We designed Chainsaw, a pull-based system that does not rely on a rigid network structure. In our experiments we used a randomly constructed graph with a fixed minimum node degree. Data is divided into finite packets and disseminated using a simple request-response protocol. In our simulations we were able to stream 100kB/sec of data to 10,000 nodes. Our system also withstood the simultaneous failure of half the nodes in the system with 99.6% of the remaining nodes suffering no packet loss at all. Moreover, we observed that new nodes joining the system could start playback within a third of a second without suffering any packet loss. To validate our simulation results, we implemented our protocol in Macedon [13] and ran experiments on PlanetLab [6], and obtained comparable results. We also compared the performance of our system to Bullet [11] and SplitStream [3].

In Section 2 we outline work related to ours. In Section 3 we describe the our system architecture. In Section 4 we present our experimental results. In Section 5 we outline some future work and finally, we conclude.

2 Background

Chu et al. [5] argue that IP is not the correct layer to implement multicast. They proposed *Narada*, a self-organizing application-layer overlay network. Since then many overlay networks [3–5, 9, 11, 12] have been proposed, providing different characteristics. We give a brief overview of SplitStream, Bullet and Gossip-style protocols. We also give an overview of BitTorrent, because it is similar in spirit to our system even though it is not a multicast system, but a file-transfer protocol.

2.1 SplitStream

SplitStream [3] is a tree-based streaming system that is built on top of the Scribe [4] overlay network, which in turn is built on top of the Pastry [14] structured routing protocol. In SplitStream, the data is divided into several disjoint sections called *stripes*, and one tree is built per stripe. In order to receive the complete stream, a node must join every tree. To ensure that a node does not have to upload more data than it receives, the trees are built such that every node is an interior node in precisely one tree.

In addition to improving fairness, ensuring that a node is a leaf node in all but one of the trees improves robustness. A node is only responsible for data forwarding on one of the stripes, so if a node suddenly leaves the system, at most one stripe is affected. However, SplitStream does not have any mechanism for recovering from packet loss, and any loss near the root of a tree will affect every node downstream from it.

2.2 Bullet

Bullet [11] is another high-bandwidth data dissemination method. It aims to provide nodes with a steady flow of data at a high rate. A Bullet network consists of a tree with a mesh overlaid on top of it.

The data stream is divided into blocks which are further divided into packets. Nodes transmit a disjoint subset of the packets to each of their children. An algorithm called RanSub [10] distributes random, orthogonal subsets of nodes every epoch to each node participating in the overlay. Nodes receive a subset of the data from their parents and recover the remaining by locating a set of disjoint peers using these random subsets.

2.3 Gossip-based Broadcast

Gossip protocols provide a scalable option for large scale information dissemination. Pcast [2] is a two phase protocol in which the exchange of periodic digests takes place independent of the data dissemination. Lpbcast [8] extends pcast in that it requires nodes to have only partial membership information.

2.4 BitTorrent

The BitTorrent [7] file sharing protocol creates an unstructured overlay mesh to distribute a file. Files are divided into discrete *pieces*. Peers that have a complete copy of the file are called *seeds*. Interested peers join this overlay to download pieces of the file. It is pull-based in that peers must request a piece in order to download it. Peers may obtain pieces either directly from the seed or exchange pieces with other peers.

3 System Description

We built a request-response based high-bandwidth data dissemination protocol drawing upon gossip-based protocols and BitTorrent. The source node, called a *seed*, generates a series of new packets with monotonically increasing sequence numbers. If desired, one could easily have multiple seeds scattered throughout the network. In this paper we assume that there is only one seed in the system. We could also support many-to-many multicast applications by replacing the sequence number with a $(\text{stream-id}, \text{sequence \#})$ tuple. However, for the applications we describe in this paper, a single sender and an integer sequence number suffice.

Every peer connects to a set of nodes that we call its *neighbors*. Peers only maintain state about their neighbors. The main piece of information they maintain is a list of packets that each neighbor has. When a peer receives a packet it sends a NOTIFY message to its neighbors. The seed obviously does not download packets, but it sends out NOTIFY messages whenever it generates new packets.

Every peer maintains a *window of interest*, which is the range of sequence numbers that the peer is interested in acquiring at the current time. It also maintains and informs its neighbors about a *window of availability*, which is the range of packets that it is willing to upload to its neighbors. The window of availability will typically be larger than the window of interest.

For every neighbor, a peer creates a list of *desired packets*, i.e. a list of packets that the peer wants, and is in the neighbor's window of availability. It will then apply some strategy to pick one or more packets from the list and request them via a REQUEST message. Currently, we simply pick packets at random, but more intelligent strategies may yield enhanced improvements (see Section 5.2).

A peer keeps track of what packets it has requested from every neighbor and ensures that it does not request the same packet from multiple neighbors. It also limits the number of outstanding requests with a given neighbor, to ensure that requests are spread out over all neighbors. Nodes keep track of requests from their neighbors and send the corresponding packets as bandwidth allows.

The algorithms that nodes use to manipulate their windows and to decide when to pass data up to the application layer are determined by the specific requirements of the end application. For example, if the application does not require strict ordering, data may be passed up as soon as it is received. On the other hand, if order must be preserved, data would be passed up as soon as a *contiguous* block is available.

For the experiments outlined in this paper, we built our graph by having every node repeatedly connect to a randomly picked node, from the set of known hosts, until it was connected to a specified minimum number of neighbors. Our system does not rely on any specific topology, however we could use other membership protocols like in BitTorrent [7] or Gnutella [1]

For the remainder of this paper, we assume that the application is similar to live streaming. The seed generates new packets at a constant rate that we refer to as the *stream rate*. Nodes maintain a window of interest of a constant size and slide it forward at a rate equal to the stream rate. If a packet has not been received by the time it "falls off" the trailing edge of the window, the node will consider that packet lost and will no longer try to acquire it.

During our initial investigations, we observed that some packets were never requested from the seed until several seconds after they were generated. As a result, those packets wouldn't propagate to all the nodes in time, resulting in packet loss. This is an artifact of picking pieces to request at random and independently from each neighbor, resulting in some pieces not being requested when that neighbor is the seed.

We fixed this problem with an algorithm called *Request Overriding*. The seed maintains a list of packets that have never been uploaded before. If the list is not empty and the seed receives a request for a packet that is not on the list, the seed ignores the sequence number requested, sends the oldest packet on the list instead, and deletes that packet from the list. This algorithm ensures that at least one copy of every packet is uploaded quickly, and the seed will not spend its upload bandwidth on uploading packets that could be obtained from other peers unless it has spare bandwidth available.

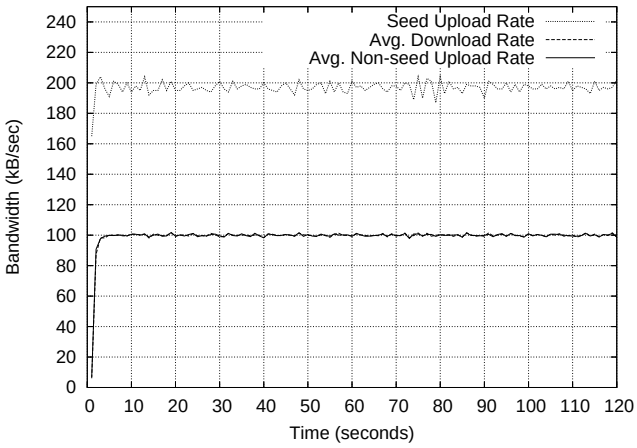

Figure 1: The seed's upload rate and the average upload and download rate for all other nodes.

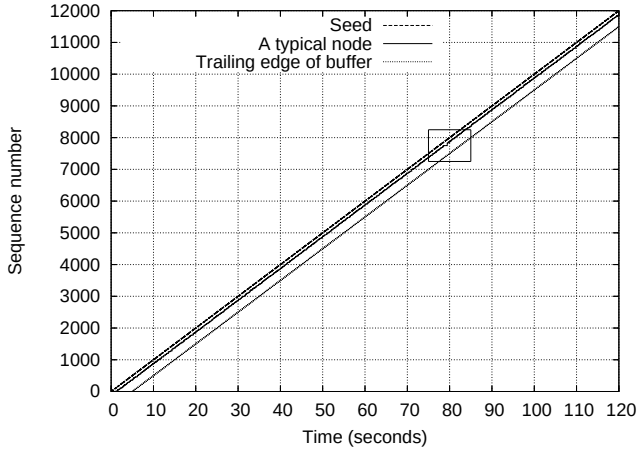

Figure 2: A plot of the highest sequence number of contiguous data downloaded by a typical node as a function of time. The diagonal line on top (dashed) represents the new pieces generated by the seed, while the bottom line (dotted) represents the trailing edge of the node's buffer.

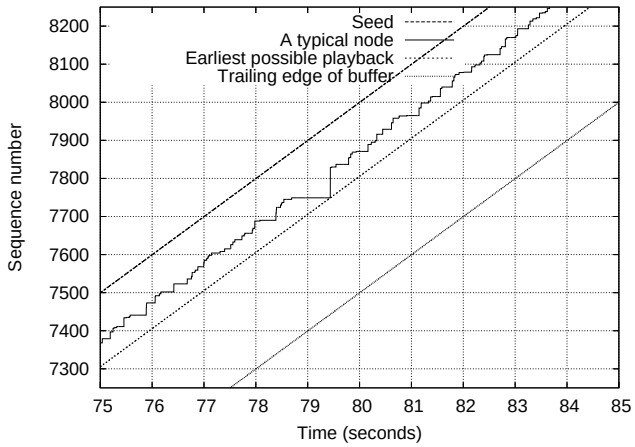

Figure 3: A zoomed in view of the highlighted portion of Figure 2. The line grazing the stepped solid line represents the minimum buffering delay that avoids packet loss.

In most cases, it is better to have the seed push out new packets quickly, but there are situations when Request Overriding is undesirable. For example, a packet may be very old and in danger of being lost. Therefore, REQUEST packets could have a bit that tells the seed to disable Request Overriding. We have not yet implemented this bit in our simulator or prototype.

4 Experimental Results

We built a discrete-time simulator to evaluate our system and run experiments on large networks. Using it, we were able to simulate 10,000 node networks. We also built a prototype implementation and compared it to Bullet [11] and SplitStream [3].

4.1 No Loss Under Normal Operation

In order to show that our system supports high-bandwidth streaming to a large number of nodes, we simulated a 10,000 node network and attempted to stream 100 kB/sec over it. The seed had an upload capacity of 200 kB/sec, while all other nodes had upload and download capacities of 120 kB/sec and maintained 5 second buffers. The end-to-end round-trip latency between all pairs of nodes was 50 ms.

Figure 1 shows the upload bandwidth of the seed and the average upload and download speeds of the non-seed nodes as a function of time. It took less than three seconds for nodes to reach the target download rate of 100 kB/sec. Once attained, their bandwidth remained steady at that rate through the end of the experiment. On average, the non-seed nodes uploaded at close to 100 kB/sec (well short of their 120 kB/sec capacity), while the seed saturated its upload capacity of 200 kB/sec.

Figure 2 shows another view of the the same experiment. The solid line represents the highest sequence number of contiguous data downloaded by a node, as a function of time. The time by which this line lags behind the dashed line representing the seed is the buffering delay for that node. The dotted diagonal line below the progress line represents the trailing edge of the node's buffer. If the progress line were to touch the line representing the trailing edge, that would imply an empty buffer and possible packet loss.

To make it easier to read, we zoom in on a portion of the graph in Figure 3. We also add a third diagonal line that just grazes the node's progress line. The time by which this line lags behind the seed line is the minimum buffering delay required to avoid all packet loss. For this node (which is, in fact, the worst of all nodes) the delay is 1.94 seconds. The remaining nodes had delays between 1.49 and 1.85 seconds.

4.2 Quick Startup Time

When a new node joins the system, it can shorten its playback time by taking advantage of the fact that its neighbors already have several seconds worth of contiguous data in their buffers. Rather than requesting the newest packets generated by the seed, the node can start requesting packets that are several seconds old. It can quickly fill up its buffer with contiguous data by requesting packets sequen-

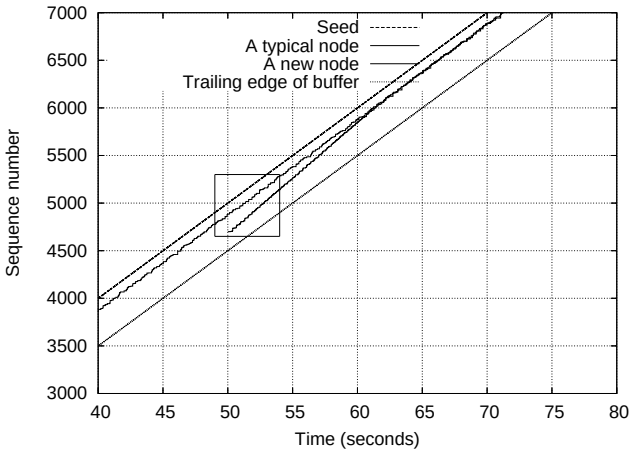

Figure 4: The bold line shows the behavior of a new node joining at 50 sec contrasted with a node that has been in the system since the start of the experiment.

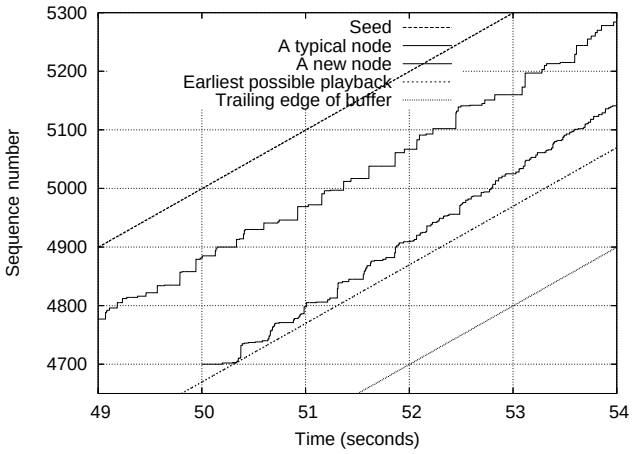

Figure 5: A zoomed in view highlighting the behavior during the first few seconds of the node joining. The dotted line grazing the bold line shows that the node could have started playback within 330 ms without suffering packet loss.

tially rather than at random.

One of the nodes in the experiment described in Section 4.1 joined the system 50 seconds later than rest. Since other nodes lagged behind the seed by less than 2 seconds, this node started by requesting packets that were 3 seconds old. Figure 4 shows the behavior of this node contrasted with the behavior of an old node. Since the node's download capacity is 20kB/sec higher than the stream rate, it is able to download faster than the stream rate and fill its buffer. In less than 15 seconds, its buffer had filled up to the same level as the older nodes. From this point on, the behavior of the new node was indistinguishable from the remaining nodes.

From the zoomed in view in Figure 5, we observe that the earliest possible playback line for the new node is 3.33 seconds behind the seed, or 330ms behind the point where the node joined. This means the node could have started playback within a third of a second of joining and not have suffered any packet loss.

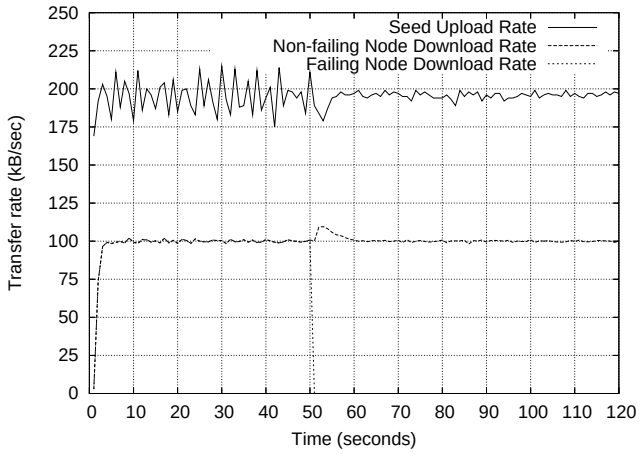

Figure 6: Observed bandwidth trends when 50% of the nodes are simultaneously failed at 50 seconds.

4.3 Resilience to Catastrophic Failure

We believe that Chainsaw is resilient to node failure because all a node has to do to recover from the failure of its neighbor is to redirect packet requests from that neighbor to a different one. We simulated a catastrophic event by killing off half the non-seed nodes simultaneously.

On average, nodes would be left with half the neighbors they had before the event, but it is likely that some unlucky nodes end up with much fewer. Therefore, we started with a minimum node degree of 40 instead of 30 to minimize the chance of a node ending up with too few neighbors. We used a 10 second buffer instead of a 5 second buffer to prevent momentary disruptions in bandwidth from causing packet loss.

Figure 6 shows the average download rate achieved by the non-failed nodes. Contrary to what one might expect, the average bandwidth briefly *increased* following the node failures! The progress line in Figure 7 helps explain this counter-intuitive behavior. Initially, nodes lagged 1.6 seconds behind the seed. Following the node failures, the lag briefly increased to 5.2 seconds, but then dropped to 0.8 seconds, because with fewer neighbors making demands on their bandwidth, nodes were able to upload and download pieces more quickly than before. The brief spurt in download rate was caused by buffers filling to a higher level than before.

The brief increase in lag was not because of reduced bandwidth, but due to "holes" in the received packets. Some of the failed nodes had received new packets from the seed and not yet uploaded them to any other node. However, since the seed only uploaded duplicate copies of those packets after at least one copy of newer packets had been uploaded, there was a delay in filling in those holes.

Of the 4999 non-seed nodes that did not fail, 4981 nodes (99.6%) suffered no packet loss at all. The remaining 18 nodes had packet loss rates ranging from 0.1% to 17.5% with a mean of 3.74%. These nodes were left with between 9 and 13 neighbors—significantly below the average 20 neighbors. In practice, every node would keep a

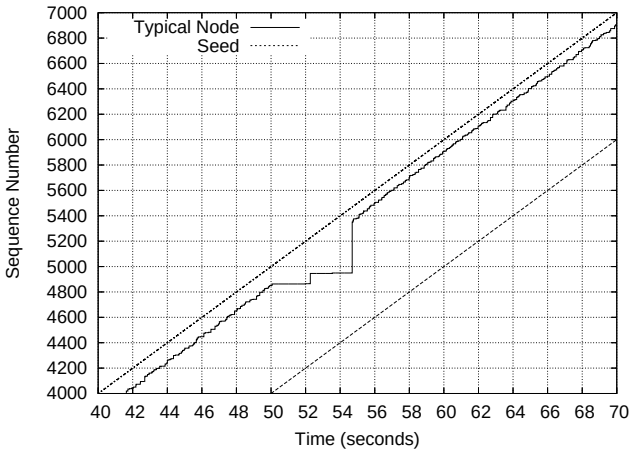

Figure 7: Progress of a non-failing node when 50% of the nodes in the network simultaneously fail at 50 seconds. All effects of the catastrophic event are eliminated within 5 seconds.

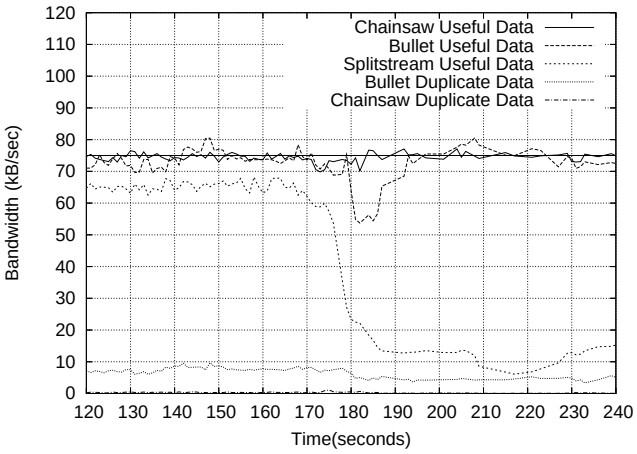

Figure 8: Useful and duplicate data rates for Chainsaw, Bullet and SplitStream as a function of time from our PlanetLab experiment. 50% of the nodes in the system were killed at the 180 second mark.

list of known peers in addition to a neighbor list. When a neighbor disappears, the node picks a neighbor randomly from the known peers list and repeats this process until it has a sufficient number of neighbors. We expect such a mechanism to be robust, even with high rates of churn.

4.4 PlanetLab: Bullet and SplitStream

In order to compare Chainsaw against Bullet [11] and SplitStream [3], we used the Macedon [13] prototyping tool, developed by the authors of Bullet. Macedon allows one to specify the high-level behavior of a system, while letting it take care of the implementation details. The Macedon distribution already includes implementations of Bullet and SplitStream, so we implemented our protocol in their framework to allow a fair comparison between these systems.

We conducted our experiments on the PlanetLab [6] test-bed, using 174 nodes with good connectivity and a large memory capacity. For each of the three protocols, we deployed the application, allowed time for it to build the network and then streamed 600 kbits/sec (75 kB/sec) over it for 360 sec. Half way into the streaming, at the 180 second mark, we killed off half the nodes to simulate catastrophic failure.

Figure 8 shows the average download rate achieved by the non-failing nodes before and after the event. Initially both Chainsaw and Bullet achieved the target bandwidth of 75 kB/sec. However, after the nodes failed, Bullet's bandwidth dropped by 30% to 53 kB/sec and it took 14 seconds to recover, while Chainsaw continued to deliver data at 75 kB/sec with no interruption. SplitStream delivered 65 kB/sec initially, but the bandwidth dropped to 13 kB/sec after the failure event.

In SplitStream, every node is an interior node in one of the trees, so its possible for a node with insufficient upload bandwidth to become a bottleneck. When a large number of nodes fail, every tree is is likely to lose a number of interior nodes, resulting in a severe reduction in bandwidth. Macedon is still a work in progress and its authors have not fully implemented SplitStream's recovery mechanisms. Once implemented, we expect SplitStream's bandwidth to return to its original level in a few seconds, once the trees are repaired. Therefore, we ignore SplitStream's packet loss and focus on comparing Chainsaw to Bullet for now.

The packet loss rates for both Chainsaw and Bullet were unaffected by the catastrophic failure. With Chainsaw 73 of the 76 non-failing nodes had no packet loss at all. One of the nodes had an a consistent loss rate of nearly 60% throughout the experiment, whereas two others had brief bursts of packet loss over intervals spanning a few seconds. With Bullet, every node consistently suffered some packet loss rates. The overall packet loss for various nodes varied from 0.88% to 3.64% with a mean of 1.30%.

With Chainsaw, nodes did receive a small number of duplicate packets due to spurious timeouts. However, the duplicate data rate rarely exceeded 1%. With Bullet, on the other hand, nodes consistently received 5-10% duplicate data, resulting in wasted bandwidth.

We think that the improved behavior that Chainsaw exhibits is primarily due to its design assumption that in the common case most of a peer's neighbors will eventually receive most packets. When combined with the direct exchange of "have" information, Chainsaw is able to locate and request packets that it does not yet have within a few RTTs, whereas Bullet's propagation of such information is divided into epochs spanning multiple seconds and is dependent on few assumptions to the RanSub tree remaining relatively intact. As a result Chainsaw has near-zero packet loss, minimal duplicates and low delay.

5 Future Work

In our experiments we have used symmetric links so that aggregate upload bandwidth was sufficient for every node to receive the broadcast at the streaming rate. If large numbers of nodes have upload capacities less than the streaming rate, as might be the case with ADSL or cable modem users, users might experience packet loss. Further work is needed to allocate bandwidth when insufficient capac-

ity exists. Also we have not demonstrated experimentally that Chainsaw performs well under high rates of churn, although we expect that with its pure mesh architecture, churn will not be a significant problem.

5.1 Incentives

So far, we have assumed that nodes are cooperative, in that they willingly satisfy their neighbor's requests. However, studies [1, 15] have shown that large fractions of nodes in peer-to-peer networks can be *leeches*, i.e. they try to benefit from the system without contributing. Chainsaw is very similar in design to our unstructured file-transfer system SWIFT [16]. Therefore, we believe that we can adapt SWIFT's pairwise currency system to ensure that nodes that do not contribute are the ones penalized when the total demand for bandwidth exceeds the total supply.

5.2 Packet Picking Strategy

Currently, nodes use a purely random strategy to decide what packets to request from their neighbors. We find that this strategy works well in general, but there are pathological cases where problems occur. For example, a node will give the same importance to a packet that is in danger of being delayed beyond the deadline as one that has just entered its window of interest. As a result it may pick the new packet instead of the old one, resulting in packet loss.

We may be able to eliminate these pathological cases and improve system performance by picking packets to request more intelligently. Possibilities include taking into account the rarity of a packet in the system, the age of the packet, and its importance to the application. Some applications may assign greater importance to some parts of the stream than others. For example, lost metadata packets may be far more difficult to recover from than lost data packets.

6 Conclusion

We built a pull-based peer-to-peer streaming network on top of an unstructured topology. Through simulations, we demonstrated that our system was capable of disseminating data at a high rate to a large number of peers with no packet loss and extremely low duplicate data rates. We also showed that a new node could start downloading and begin play back within a fraction of a second after joining the network, making it highly suitable to applications like on-demand media streaming. Finally, we showed that our system is robust to catastrophic failure. A vast majority of the nodes were able to download data with no packet loss even when half the nodes in the system failed simultaneously.

So far we have only investigated behavior in a *cooperative* environment. However, Chainsaw is very similar in its design to the SWIFT [16] incentive-based file-trading network. Therefore, we believe that we will be able to adapt SWIFT's economic incentive model to streaming, allowing our system to work well in non-cooperative environments.

Acknowledgements

We would like to thank Dejan Kostić and Charles Killian for helping us out with MACEDON.

References

[1] E. Adar and B. A. Huberman. Free Riding on Gnutella. *First Monday*, 5(10), Oct 2000.

[2] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Trans. Comput. Syst.*, 1999.

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

[4] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE JSAC*, 2002.

[5] Y. Chu, S. G. Rao, and H. Zhang. A case for end system multicast. In *Measurement and Modeling of Computer Systems*, 2000.

[6] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. Planetlab: an overlay testbed for broad-coverage services. *SIGCOMM Computer Communication Review*, 2003.

[7] B. Cohen. BitTorrent, 2001. http://www.bitconjurer.org/BitTorrent/.

[8] P. Eugster, R. Guerraoui, S. B. Handurukande, P. Kouznetsov, and A. Kermarrec. Lightweight probabilistic broadcast. *ACM Trans. Comput. Syst.*, 2003.

[9] J. Jannotti, D. K. Gifford, K. L. Johnson, M. Frans Kaashoek, and J. O'Toole, Jr. Overcast: Reliable multicasting with an overlay network. In *OSDI*, 2000.

[10] D. Kostić, A. Rodriguez, J. Albrecht, A. Bhirud, and A. Vahdat. Using random subsets to build scalable network services. In *USENIX USITS*, 2003.

[11] D. Kostić, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: high bandwidth data dissemination using an overlay mesh. In *SOSP*, 2003.

[12] S. Ratnasamy, M. Handley, R. M. Karp, and S. Shenker. Application-level multicast using content-addressable networks. In *Workshop on Networked Group Communication*, 2001.

[13] A. Rodriguez, C. Killian, S. Bhat, D. Kostić, and A. Vahadat. Macedon: Methodology for Automtically Creating, Evaluating, and Designing Overlay Networks. In *NSDI*, 2004.

[14] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms*, 2001.

[15] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. *Proceedings of Multimedia Computing and Networking*, 2002.

[16] 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.