Bullet: High Bandwidth Data Dissemination Using an Overlay Mesh

Dejan Kostić, Adolfo Rodriguez, Jeannie Albrecht, and Amin Vahdat*
Department of Computer Science
Duke University
{dkostic,razor,albrecht,vahdat}@cs.duke.edu

ABSTRACT

In recent years, overlay networks have become an effective alternative to IP multicast for efficient point to multipoint communication across the Internet. Typically, nodes self-organize with the goal of forming an efficient overlay tree, one that meets performance targets without placing undue burden on the underlying network. In this paper, we target high-bandwidth data distribution from a single source to a large number of receivers. Applications include large-file transfers and real-time multimedia streaming. For these applications, we argue that an overlay *mesh*, rather than a tree, can deliver fundamentally higher bandwidth and reliability relative to typical tree structures. This paper presents *Bullet*, a scalable and distributed algorithm that enables nodes spread across the Internet to self-organize into a high bandwidth overlay mesh. We construct Bullet around the insight that data should be distributed in a disjoint manner to strategic points in the network. Individual Bullet receivers are then responsible for locating and retrieving the data from multiple points in parallel.

Key contributions of this work include: i) an algorithm that sends data to different points in the overlay such that any data object is equally likely to appear at any node, ii) a scalable and decentralized algorithm that allows nodes to locate and recover missing data items, and iii) a complete implementation and evaluation of Bullet running across the Internet and in a large-scale emulation environment reveals up to a factor two bandwidth improvements under a variety of circumstances. In addition, we find that, relative to tree-based solutions, Bullet reduces the need to perform expensive bandwidth probing. In a tree, it is critical that a node's parent delivers a high rate of application data to each child. In Bullet however, nodes simultaneously receive data from multiple sources in parallel, making it less important to locate any single source capable of sustaining a high transmission rate.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems; H.4.3 [**Information Systems Applications**]: Communications Applications

General Terms

Experimentation, Management, Performance

Keywords

Bandwidth, Overlays, Peer-to-peer

*This research is supported in part by the National Science Foundation (EIA-99772879, ITR-0082912), Hewlett Packard, IBM, Intel, and Microsoft. In addition, Albrecht is supported by an NSF fellowship and Vahdat is supported by an NSF CAREER award (CCR-9984328).

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

1. INTRODUCTION

In this paper, we consider the following general problem. Given a sender and a large set of interested receivers spread across the Internet, how can we maximize the amount of bandwidth delivered to receivers? Our problem domain includes software or video distribution and real-time multimedia streaming. Traditionally, native IP multicast has been the preferred method for delivering content to a set of receivers in a scalable fashion. However, a number of considerations, including scale, reliability, and congestion control, have limited the wide-scale deployment of IP multicast. Even if all these problems were to be addressed, IP multicast does not consider bandwidth when constructing its distribution tree. More recently, overlays have emerged as a promising alternative to multicast for network-efficient point to multipoint data delivery.

Typical overlay structures attempt to mimic the structure of multicast routing trees. In network-layer multicast however, interior nodes consist of high speed routers with limited processing power and extensibility. Overlays, on the other hand, use programmable (and hence extensible) end hosts as interior nodes in the overlay tree, with these hosts acting as repeaters to multiple children down the tree. Overlays have shown tremendous promise for multicast-style applications. However, we argue that a tree structure has fundamental limitations both for high bandwidth multicast and for high reliability. One difficulty with trees is that bandwidth is guaranteed to be monotonically decreasing moving down the tree. Any loss high up the tree will reduce the bandwidth available to receivers lower down the tree. A number of techniques have been proposed to recover from losses and hence improve the available bandwidth in an overlay tree [2, 6]. However, fundamentally, the bandwidth available to any host is limited by the bandwidth available from that node's single parent in the tree.

Thus, our work operates on the premise that the model for high-bandwidth multicast data dissemination should be re-examined. Rather than sending identical copies of the same data stream to all nodes in a tree and designing a scalable mechanism for recovering from loss, we propose that participants in a multicast overlay cooperate to strategically transmit *disjoint* data sets to various points in the network. Here, the sender splits data into sequential blocks. Blocks are further subdivided into individual objects which are in turn transmitted to different points in the network. Nodes still receive a set of objects from their parents, but they are then responsible for locating peers that hold missing data objects. We use a distributed algorithm that aims to make the availability of data items uniformly spread across all overlay participants. In this way, we avoid the problem of locating the "last object", which may only be available at a few nodes. One hypothesis of this work is that, relative to a tree, this model will result in higher bandwidth—leveraging the bandwidth from simultaneous parallel downloads from multiple sources rather than a single parent—and higher reliability—retrieving data from multiple peers reduces the potential damage from a single node failure.

To illustrate Bullet's behavior, consider a simple three node overlay with a root R and two children A and B. R has 1 Mbps of available (TCP-friendly) bandwidth to each of A and B. However, there is also 1 Mbps of available bandwidth between A and B. In this example, Bullet would transmit a disjoint set of data at 1 Mbps to each of A and B. A and B would then each independently discover the availability of disjoint data at the remote peer and begin streaming data to one another, effectively achieving a retrieval rate of 2 Mbps. On the other hand, any overlay tree is restricted to delivering at most 1 Mbps even with a scalable technique for recovering lost data.

Any solution for achieving the above model must maintain a number of properties. First, it must be TCP friendly [15]. No flow should consume more than its fair share of the bottleneck bandwidth and each flow must respond to congestion signals (losses) by reducing its transmission rate. Second, it must impose low control overhead. There are many possible sources of such overhead, including probing for available bandwidth between nodes, locating appropriate nodes to "peer" with for data retrieval and redundantly receiving the same data objects from multiple sources. Third, the algorithm should be decentralized and scalable to thousands of participants. No node should be required to learn or maintain global knowledge, for instance global group membership or the set of data objects currently available at all nodes. Finally, the approach must be robust to individual failures. For example, the failure of a single node should result only in a temporary reduction in the bandwidth delivered to a small subset of participants; no single failure should result in the complete loss of data for any significant fraction of nodes, as might be the case for a single node failure "high up" in a multicast overlay tree.

In this context, this paper presents the design and evaluation of Bullet, an algorithm for constructing an overlay mesh that attempts to maintain the above properties. Bullet nodes begin by self-organizing into an overlay tree, which can be constructed by any of a number of existing techniques [1, 18, 21, 24, 34]. Each Bullet node, starting with the root of the underlying tree, then transmits a *disjoint* set of data to each of its children, with the goal of maintaining uniform representativeness of each data item across all participants. The level of disjointness is determined by the bandwidth available to each of its children. Bullet then employs a scalable and efficient algorithm to enable nodes to quickly locate multiple peers capable of transmitting missing data items to the node. Thus, Bullet layers a high-bandwidth mesh on top of an arbitrary overlay tree. Depending on the type of data being transmitted, Bullet can optionally employ a variety of encoding schemes, for instance Erasure codes [7, 26, 25] or Multiple Description Coding (MDC) [17], to efficiently disseminate data, adapt to variable bandwidth, and recover from losses. Finally, we use TFRC [15] to transfer data both down the overlay tree and among peers. This ensures that the entire overlay behaves in a congestion-friendly manner, adjusting its transmission rate on a per-connection basis based on prevailing network conditions.

One important benefit of our approach is that the bandwidth delivered by the Bullet mesh is somewhat independent of the bandwidth available through the underlying overlay tree. One significant limitation to building high bandwidth overlay trees is the overhead associated with the tree construction protocol. In these trees, it is critical that each participant locates a parent via probing with a high level of available bandwidth because it receives data from only a single source (its parent). Thus, even once the tree is constructed, nodes must continue their probing to adapt to dynamically changing network conditions. While bandwidth probing is an active area of research [20, 35], accurate results generally require the transfer of a large amount of data to gain confidence in the results. Our approach with Bullet allows receivers to obtain high bandwidth in *aggregate* using individual transfers from peers spread across the system. Thus, in Bullet, the bandwidth available from any individual peer is much less important than in any bandwidth-optimized tree. Further, all the bandwidth that would normally be consumed probing for bandwidth can be reallocated to streaming data across the Bullet mesh.

We have completed a prototype of Bullet running on top of a number of overlay trees. Our evaluation of a 1000-node overlay running across a wide variety of emulated 20,000 node network topologies shows that Bullet can deliver up to twice the bandwidth of a bandwidth-optimized tree (using an offline algorithm and global network topology information), all while remaining TCP friendly. We also deployed our prototype across the PlanetLab [31] wide-area testbed. For these live Internet runs, we find that Bullet can deliver comparable bandwidth performance improvements. In both cases, the overhead of maintaining the Bullet mesh and locating the appropriate disjoint data is limited to 30 Kbps per node, acceptable for our target high-bandwidth, large-scale scenarios.

The remainder of this paper is organized as follows. Section 2 presents Bullet's system components including RanSub, informed content delivery, and TFRC. Section 3 then details Bullet, an efficient data distribution system for bandwidth intensive applications. Section 4 evaluates Bullet's performance for a variety of network topologies, and compares it to existing multicast techniques. Section 5 places our work in the context of related efforts and Section 6 presents our conclusions.

2. SYSTEM COMPONENTS

Our approach to high bandwidth data dissemination centers around the techniques depicted in Figure 1. First, we split the target data stream into blocks which are further subdivided into individual (typically packet-sized) objects. Depending on the requirements of the target applications, objects may be encoded [17, 26] to make data recovery more efficient. Next, we purposefully disseminate disjoint objects

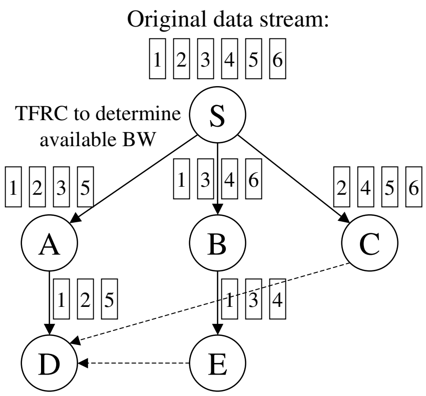

Figure 1: High-level view of Bullet's operation.

to different clients at a rate determined by the available bandwidth to each client. We use the equation-based TFRC protocol to communicate among all nodes in the overlay in a congestion responsive and TCP friendly manner.

Given the above techniques, data is spread across the overlay tree at a rate commensurate with the available bandwidth in the overlay tree. Our overall goal however is to deliver more bandwidth than would otherwise be available through any tree. Thus, at this point, nodes require a scalable technique for locating and retrieving disjoint data from their peers. In essence, these perpendicular links across the overlay form a mesh to augment the bandwidth available through the tree. In Figure 1, node D only has sufficient bandwidth to receive 3 objects per time unit from its parent. However, it is able to locate two peers, C and E, who are able to transmit "missing" data objects, in this example increasing delivered bandwidth from 3 objects per time unit to 6 data objects per time unit. Locating appropriate remote peers cannot require global state or global communication. Thus, we propose the periodic dissemination of changing, uniformly random subsets of global state to each overlay node once per configurable time period. This random subset contains summary tickets of the objects available at a subset of the nodes in the system. Each node uses this information to request data objects from remote nodes that have significant divergence in object membership. It then attempts to establish a number of these peering relationships with the goals of minimizing overlap in the objects received from each peer and maximizing the total useful bandwidth delivered to it.

In the remainder of this section, we provide brief background on each of the techniques that we employ as fundamental building blocks for our work. Section 3 then presents the details of the entire Bullet architecture.

2.1 Data Encoding

Depending on the type of data being distributed through the system, a number of data encoding schemes can improve system efficiency. For instance, if multimedia data is being distributed to a set of heterogeneous receivers with variable bandwidth, MDC [17] allows receivers obtaining different subsets of the data to still maintain a usable multimedia stream. For dissemination of a large file among a set of receivers, Erasure codes enable receivers not to focus on retrieving *every* transmitted data packet. Rather, after obtaining a threshold minimum number of packets, receivers are able to decode the original data stream. Of course, Bullet is amenable to a variety of other encoding schemes or even the "null" encoding scheme, where the original data stream is transmitted best-effort through the system.

In this paper, we focus on the benefits of a special class of erasure-correcting codes used to implement the "digital fountain" [7] approach. Redundant Tornado [26] codes are created by performing XOR operations on a selected number of original data packets, and then transmitted along with the original data packets. Tornado codes require *any* $(1+\epsilon)k$ correctly received packets to reconstruct the original k data packets, with the typically low *reception overhead* (ϵ) of $0.03 - 0.05$. In return, they provide significantly faster encoding and decoding times. Additionally, the decoding algorithm can run in real-time, and the reconstruction process can start as soon as sufficiently many packets have arrived. Tornado codes require a predetermined *stretch* factor (n/k, where n is the total number of encoded packets), and their encoding time is proportional to n. LT codes [25] remove these two limitations, while maintaining a low reception overhead of 0.05.

2.2 RanSub

To address the challenge of locating disjoint content within the system, we use RanSub [24], a scalable approach to distributing changing, uniform random subsets of global state to all nodes of an overlay tree. RanSub assumes the presence of some scalable mechanism for efficiently building and maintaining the underlying tree. A number of such techniques are described in [1, 18, 21, 24, 34].

RanSub distributes random subsets of participating nodes throughout the tree using *collect* and *distribute* messages. Collect messages start at the leaves and propagate up the tree, leaving state at each node along the path to the root. Distribute messages start at the root and travel down the tree, using the information left at the nodes during the previous collect round to distribute uniformly random subsets to all participants. Using the collect and distribute messages, RanSub distributes a random subset of participants to each node once per *epoch*. The lower bound on the length of an epoch is determined by the time it takes to propagate data up then back down the tree, or roughly twice the height of the tree. For appropriately constructed trees, the minimum epoch length will grow with the logarithm of the number of participants, though this is not required for correctness.

As part of the distribute message, each participant sends a uniformly random subset of remote nodes, called a *distribute set*, down to its children. The contents of the distribute set are constructed using the *collect set* gathered during the previous *collect phase*. During this phase, each participant sends a collect set consisting of a random subset of its descendant nodes up the tree to the root along with an estimate of its total number of descendants. After the root receives all collect sets and the collect phase completes, the distribute phase begins again in a new epoch.

One of the key features of RanSub is the *Compact* operation. This is the process used to ensure that membership in a collect set propagated by a node to its parent is both random and uniformly representative of all members of the sub-tree rooted at that node. Compact takes multiple fixed-size subsets and the total population represented by each subset as input, and generates a new fixed-size subset. The

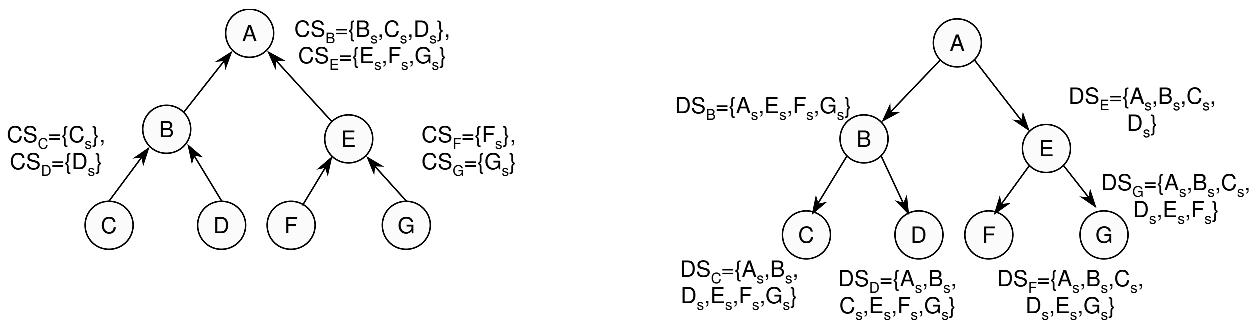

Figure 2: This example shows the two phases of the RanSub protocol that occur in one epoch. The collect phase is shown on the left, where the collect sets are traveling up the overlay to the root. The distribute phase on the right shows the distribute sets traveling down the overlay to the leaf nodes.

members of the resulting set are uniformly random representatives of the input subset members.

RanSub offers several ways of constructing distribute sets. For our system, we choose the *RanSub-nondescendants* option. In this case, each node receives a random subset consisting of all nodes excluding its descendants. This is appropriate for our download structure where descendants are expected to have less content than an ancestor node in most cases.

A parent creates *RanSub-nondescendants* distribute sets for each child by compacting collect sets from that child's siblings and its own distribute set. The result is a distribute set that contains a random subset representing all nodes in the tree except for those rooted at that particular child. We depict an example of RanSub's collect-distribute process in Figure 2. In the figure, A_S stands for node A's state.

2.3 Informed Content Delivery Techniques

Assuming we can enable a node to locate a peer with disjoint content using RanSub, we need a method for reconciling the differences in the data. Additionally, we require a bandwidth-efficient method with low computational overhead. We chose to implement the approximate reconciliation techniques proposed in [6] for these tasks in Bullet.

To describe the content, nodes maintain *working sets*. The working set contains sequence numbers of packets that have been successfully received by each node over some period of time. We need the ability to quickly discern the resemblance between working sets from two nodes and decide whether a fine-grained reconciliation is beneficial. Summary tickets, or *min-wise sketches* [5], serve this purpose. The main idea is to create a summary ticket that is an unbiased random sample of the working set. A summary ticket is a small fixed-size array. Each entry in this array is maintained by a specific permutation function. The goal is to have each entry populated by the element with the smallest permuted value. To insert a new element into the summary ticket, we apply the permutation functions in order and update array values as appropriate.

The permutation function can be thought of as a specialized hash function. The choice of permutation functions is important as the quality of the summary ticket depends directly on the randomness properties of the permutation functions. Since we require them to have a low computational overhead, we use simple permutation functions, such as $P_j(x) = (ax+b)mod|U|$, where U is the universe size (dependant on the data encoding scheme). To compute the resemblance between two working sets, we compute the number of summary ticket entries that have the same value, and divide it by the total number of entries in the summary tickets. Figure 3 shows the way the permutation functions are used to populate the summary ticket.

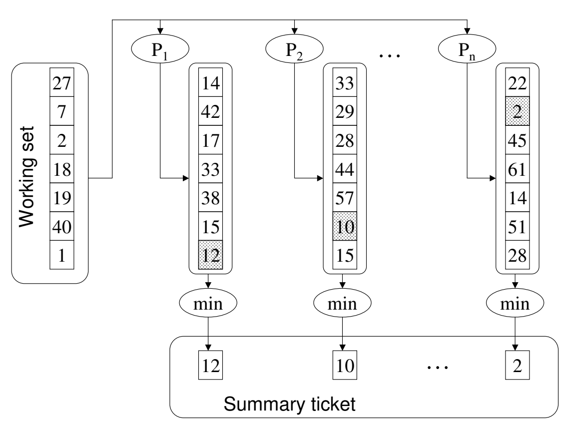

Figure 3: Example showing a sample summary ticket being constructed from the working set.

To perform approximate fine-grain reconciliation, a peer A sends its digest to peer B and expects to receive packets not described in the digest. For this purpose, we use a Bloom filter [4], a bit array of size m with k independent associated hash functions. An element s from the set of received keys $S = \{s_o, s_2, \ldots, s_{n-1}\}$ is inserted into the filter by computing the hash values $h_0, h_1, \ldots, h_{k-1}$ of s and setting the bits in the array that correspond to the hashed values. To check whether an element x is in the Bloom filter, we hash it using the hash functions and check whether all positions in the bit array are set. If at least one is not set, we know that the Bloom filter does not contain x.

When using Bloom filters, the insertion of different elements might cause all the positions in the bit array corresponding to an element that is not in the set to be nonzero. In this case, we have a false positive. Therefore, it is possible that peer B will not send a packet to peer A even though A is missing it. On the other hand, a node will never send a packet that is described in the Bloom filter, i.e. there are no false negatives. The probability of getting a false positive p_f on the membership query can be expressed as a function of the ratio $\frac{m}{n}$ and the number of hash functions k: $p_f = (1 - e^{-kn/m})^k$. We can therefore choose the size of the Bloom filter and the number of hash functions that will yield a desired false positive ratio.

2.4 TCP Friendly Rate Control

Although most traffic in the Internet today is best served by TCP, applications that require a smooth sending rate and that have a higher tolerance for loss often find TCP's reaction to a single dropped packet to be unnecessarily severe. TCP Friendly Rate Control, or TFRC, targets unicast streaming multimedia applications with a need for less drastic responses to single packet losses [15]. TCP halves the sending rate as soon as one packet loss is detected. Alternatively, TFRC is an equation-based congestion control protocol that is based on loss events, which consist of multiple packets being dropped within one round-trip time. Unlike TCP, the goal of TFRC is not to find and use all available bandwidth, but instead to maintain a relatively steady sending rate while still being responsive to congestion.

To guarantee fairness with TCP, TFRC uses the response function that describes the steady-state sending rate of TCP to determine the transmission rate in TFRC. The formula of the TCP response function [27] used in TFRC to describe the sending rate is:

$$T = \frac{s}{R\sqrt{\frac{2p}{3}} + t_{RTO}(3\sqrt{\frac{3p}{8}})p(1+32p^2)}$$

This is the expression for the sending rate T in bytes/second, as a function of the round-trip time R in seconds, loss event rate p, packet size s in bytes, and TCP retransmit value t_{RTO} in seconds.

TFRC senders and receivers must cooperate to achieve a smooth transmission rate. The sender is responsible for computing the weighted round-trip time estimate R between sender and receiver, as well as determining a reasonable retransmit timeout value t_{RTO}. In most cases, using the simple formula $t_{RTO} = 4R$ provides the necessary fairness with TCP. The sender is also responsible for adjusting the sending rate T in response to new values of the loss event rate p reported by the receiver. The sender obtains a new measure for the loss event rate each time a feedback packet is received from the receiver. Until the first loss is reported, the sender doubles its transmission rate each time it receives feedback just as TCP does during slow-start.

The main role of the receiver is to send feedback to the sender once per round-trip time and to calculate the loss event rate included in the feedback packets. To obtain the loss event rate, the receiver maintains a loss interval array that contains values for the last eight loss intervals. A loss interval is defined as the number of packets received correctly between two loss events. The array is continually updated as losses are detected. A weighted average is computed based on the sum of the loss interval values, and the inverse of the sum is the reported loss event rate, p.

When implementing Bullet, we used an unreliable version of TFRC. We wanted a transport protocol that was congestion aware and TCP friendly. Lost packets were more easily recovered from other sources rather than waiting for a retransmission from the initial sender. Hence, we eliminate retransmissions from TFRC. Further, TFRC does not aggressively seek newly available bandwidth like TCP, a desirable trait in an overlay tree where there might be multiple competing flows sharing the same links. For example, if a leaf node in the tree tried to aggressively seek out new bandwidth, it could create congestion all the way up to the root of the tree. By using TFRC we were able to avoid these scenarios.

3. BULLET

Bullet is an efficient data distribution system for bandwidth intensive applications. While many current overlay network distribution algorithms use a distribution tree to deliver data from the tree's root to all other nodes, Bullet layers a mesh on top of an original overlay tree to increase overall bandwidth to all nodes in the tree. Hence, each node receives a *parent stream* from its parent in the tree and some number of *perpendicular streams* from chosen peers in the overlay. This has significant bandwidth impact when a single node in the overlay is unable to deliver adequate bandwidth to a receiving node.

Bullet requires an underlying overlay tree for RanSub to deliver random subsets of participants's state to nodes in the overlay, informing them of a set of nodes that may be good candidates for retrieving data not available from any of the node's current peers and parent. While we also use the underlying tree for baseline streaming, this is not critical to Bullet's ability to efficiently deliver data to nodes in the overlay. As a result, Bullet is capable of functioning on top of essentially any overlay tree. In our experiments, we have run Bullet over random and bandwidth-optimized trees created offline (with global topological knowledge). Bullet registers itself with the underlying overlay tree so that it is informed when the overlay changes as nodes come and go or make performance transformations in the overlay.

As with streaming overlays trees, Bullet can use standard transports such as TCP and UDP as well as our implementation of TFRC. For the remainder of this paper, we assume the use of TFRC since we primarily target streaming high-bandwidth content and we do not require reliable or in-order delivery. For simplicity, we assume that packets originate at the root of the tree and are tagged with increasing sequence numbers. Each node receiving a packet will optionally forward it to each of its children, depending on a number of factors relating to the child's bandwidth and its relative position in the tree.

3.1 Finding Overlay Peers

RanSub periodically delivers subsets of uniformly random selected nodes to each participant in the overlay. Bullet receivers use these lists to locate remote peers able to transmit missing data items with good bandwidth. RanSub messages contain a set of summary tickets that include a small (120 bytes) summary of the data that each node contains. RanSub delivers subsets of these summary tickets to nodes every configurable *epoch* (5 seconds by default). Each node in the tree maintains a working set of the packets it has received thus far, indexed by sequence numbers. Nodes associate each working set with a Bloom filter that maintains a summary of the packets received thus far. Since the Bloom filter does not exceed a specific size (m) and we would like to limit the rate of false positives, Bullet periodically cleans up the Bloom filter by removing lower sequence numbers from it. This allows us to keep the Bloom filter population n from growing at an unbounded rate. The net effect is that a node will attempt to recover packets for a finite amount of time depending on the packet arrival rate. Similarly, Bullet removes older items that are not needed for data reconstruction from its working set and summary ticket.

We use the collect and distribute phases of RanSub to carry Bullet summary tickets up and down the tree. In our current implementation, we use a set size of 10 summary tickets, allowing each collect and distribute to fit well within the size of a non-fragmented IP packet. Though Bullet supports larger set sizes, we expect this parameter to be tunable to specific applications' needs. In practice, our default size of 10 yields favorable results for a variety of overlays and network topologies. In essence, during an epoch a node receives a summarized partial view of the system's state at that time. Upon receiving a random subset each epoch, a Bullet node may choose to peer with the node having the lowest similarity ratio when compared to its own summary ticket. This is done only when the node has sufficient space in its sender list to accept another sender (senders with lackluster performance are removed from the current sender list as described in section 3.4). Once a node has chosen the best node it sends it a peering request containing the requesting node's Bloom filter. Such a request is accepted by the potential sender if it has sufficient space in its receiver list for the incoming receiver. Otherwise, the send request is rejected (space is periodically created in the receiver lists as further described in section 3.4).

3.2 Recovering Data From Peers

Assuming it has space for the new peer, a recipient of the peering request installs the received Bloom filter and will periodically transmit keys not present in the Bloom filter to the requesting node. The requesting node will refresh its installed Bloom filters at each of its sending peers periodically. Along with the fresh filter, a receiving node will also assign a portion of the sequence space to each of its senders. In this way, a node is able the reduce the likelihood that two peers simultaneously transmit the same key to it, wasting network resources. A node divides the sequence space in its current working set among each of its senders uniformly.

As illustrated in Figure 4, a Bullet receiver views the data space as a matrix of packet sequences containing s rows, where s is its current number of sending peers. A receiver periodically (every 5 seconds by default) updates each sender with its current Bloom filter and the range of sequences covered in its Bloom filter. This identifies the range of packets that the receiver is currently interested in recovering. Over time, this range shifts as depicted in Figure 4-b). In addition, the receiving node assigns to each sender a row from the matrix, labeled *mod*. A sender will forward packets to

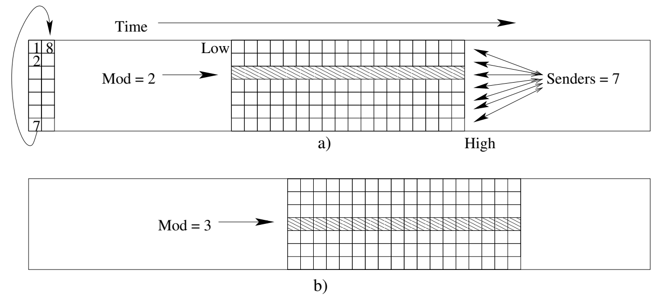

Figure 4: A Bullet receiver views data as a matrix of sequenced packets with rows equal to the number of peer senders it currently has. It requests data within the range (Low, High) of sequence numbers based on what it has received. a) The receiver requests a specific row in the sequence matrix from each sender. b) As it receives more data, the range of sequences advances and the receiver requests different rows from senders.

the receiver that have a sequence number x such that x modulo s equals the *mod* number. In this fashion, receivers register to receive *disjoint* data from their sending peers.

By specifying ranges and matrix rows, a receiver is unlikely to receive duplicate data items, which would result in wasted bandwidth. A duplicate packet, however, may be received when a parent recovers a packet from one of its peers and relays the packet to its children (and descendants). In this case, a descendant would receive the packet out of order and may have already recovered it from one of its peers. In practice, this wasteful reception of duplicate packets is tolerable; less than 10% of all received packets are duplicates in our experiments.

3.3 Making Data Disjoint

We now provide details of Bullet's mechanisms to increase the ease by which nodes can find disjoint data not provided by parents. We operate on the premise that the main challenge in recovering lost data packets transmitted over an overlay distribution tree lies in finding the peer node housing the data to recover. Many systems take a hierarchical approach to this problem, propagating repair requests up the distribution tree until the request can be satisfied. This ultimately leads to scalability issues at higher levels in the hierarchy particularly when overlay links are bandwidth-constrained.

On the other hand, Bullet attempts to recover lost data from any non-descendant node, not just ancestors, thereby increasing overall system scalability. In traditional overlay distribution trees, packets are lost by the transmission transport and/or the network. Nodes attempt to stream data as fast as possible to each child and have essentially no control over which portions of the data stream are dropped by the transport or network. As a result, the streaming subsystem has no control over how many nodes in the system will ultimately receive a particular portion of the data. If few nodes receive a particular range of packets, recovering these pieces of data becomes more difficult, requiring increased communication costs, and leading to scalability problems.

In contrast, Bullet nodes are aware of the bandwidth achievable to each of its children using the underlying transport. If a child is unable to receive the streaming rate that the parent receives, the parent consciously decides which portion of the data stream to forward to the constrained child. In addition, because nodes recover data from participants chosen uniformly at random from the set of non-descendants, it is advantageous to make each transmitted packet recoverable from approximately the same number of participant nodes. That is, given a randomly chosen subset of peer nodes, it is with the same probability that each node has a particular data packet. While not explicitly proven here, we believe that this approach maximizes the probability that a lost data packet can be recovered, regardless of which packet is lost. To this end, Bullet distributes incoming packets among one or more children in hopes that the expected number of nodes receiving each packet is approximately the same.

A node p maintains for each child, i, a limiting and sending factor, lf_i and sf_i. These factors determine the proportion of p's received data rate that it will forward to each child. The sending factor sf_i is the portion of the parent stream (rate) that each child should "own" based on the number of descendants the child has. The more descendants a child has, the larger the portion of received data it should own. The limiting factor lf_i represents the proportion of the parent rate beyond the sending factor that each child can handle. For example, a child with one descendant, but high bandwidth would have a low sending factor, but a very high limiting factor. Though the child is responsible for owning a small portion of the received data, it actually can receive a large portion of it.

Because RanSub collects descendant counts d_i for each child i, Bullet simply makes a call into RanSub when sending data to determine the current sending factors of its children. For each child i out of k total, we set the sending factor to be:

$$sf_i = \frac{d_i}{\sum_{j=1}^{k} d_j}.$$

In addition, a node tracks the data successfully transmitted via the transport. That is, Bullet data transport sockets are non-blocking; successful transmissions are send attempts that are accepted by the non-blocking transport. If the transport would block on a send (i.e., transmission of the packet would exceed the TCP-friendly fair share of network resources), the send fails and is counted as an unsuccessful send attempt. When a data packet is received by a parent, it calculates the proportion of the total data stream that has been sent to each child, thus far, in this epoch. It then assigns ownership of the current packet to the child with sending proportion farthest away from its sf_i as illustrated in Figure 5.

Having chosen the target of a particular packet, the parent attempts to forward the packet to the child. If the send is not successful, the node must find an alternate child to own the packet. This occurs when a child's bandwidth is not adequate to fulfill its responsibilities based on its descendants (sf_i). To compensate, the node attempts to deterministically find a child that can own the packet (as evidenced by its transport accepting the packet). The net result is that children with more than adequate bandwidth will own more of their share of packets than those with inadequate bandwidth. In the event that no child can accept a packet, it must be dropped, corresponding to the case where the sum of all children bandwidths is inadequate to serve the received

```
foreach child in children {
  if ( (child->sent / total_sent)
        < child->sending_factor)
    target_child = child;
}

if (!senddata( target_child->addr,
               msg, size, key)) {
  // send succeeded
  target_child->sent++;
  target_child->child_filter.insert(got_key);
  sent_packet = 1;
}

foreach child in children {
  should_send = 0;
  if (!sent_packet)   // transfer ownership
    should_send = 1;
  else    //  test for available bandwidth
    if ( key % (1.0/child->limiting_factor) == 0 )
      should_send = 1;
  if (should_send) {
    if (!senddata( child->addr,
                   msg, size, key)) {
      if (!sent_packet)   // i received ownership
        child->sent++;
      else
        increase(child->limiting_factor);
      child->child_filter.insert(got_key);
      sent_packet = 1;
    }
    else   // send failed
      if (sent_packet)    // was for extra bw
        decrease(child->limiting_factor);
  }
}
```

Figure 5: Pseudo code for Bullet's disjoint data send routine

stream. While making data more difficult to recover, Bullet still allows for recovery of such data to its children. The sending node will cache the data packet and serve it to its requesting peers. This process allows its children to potentially recover the packet from one of their own peers, to whom additional bandwidth may be available.

Once a packet has been successfully sent to the owning child, the node attempts to send the packet to all other children depending on the limiting factors lf_i. For each child i, a node attempts to forward the packet deterministically if the packet's sequence modulo $1/lf_i$ is zero. Essentially, this identifies which lf_i fraction of packets of the received data stream should be forwarded to each child to make use of the available bandwidth to each. If the packet transmission is successful, lf_i is increased such that one more packet is to be sent per epoch. If the transmission fails, lf_i is decreased by the same amount. This allows children limiting factors to be continuously adjusted in response to changing network conditions.

It is important to realize that by maintaining limiting factors, we are essentially using feedback from children (by observing transport behavior) to determine the best data to stop sending during times when a child cannot handle the entire parent stream. In one extreme, if the sum of children bandwidths is not enough to receive the entire parent stream, each child will receive a completely disjoint data stream of packets it owns. In the other extreme, if each child has ample bandwidth, it will receive the entire parent stream as each lf_i would settle on 1.0. In the general case, our owning strategy attempts to make data disjoint among children subtrees with the guiding premise that, as much as possible, the expected number of nodes receiving a packet is the same across all packets.

3.4 Improving the Bullet Mesh

Bullet allows a maximum number of peering relationships. That is, a node can have up to a certain number of receivers and a certain number of senders (each defaults to 10 in our implementation). A number of considerations can make the current peering relationships sub-optimal at any given time: i) the probabilistic nature of RanSub means that a node may not have been exposed to a sufficiently appropriate peer, ii) receivers greedily choose peers, and iii) network conditions are constantly changing. For example, a sender node may wind up being unable to provide a node with very much useful (non-duplicate) data. In such a case, it would be advantageous to remove that sender as a peer and find some other peer that offers better utility.

Each node periodically (every few RanSub epochs) evaluates the bandwidth performance it is receiving from its sending peers. A node will drop a peer if it is sending too many duplicate packets when compared to the total number of packets received. This threshold is set to 50% by default. If no such wasteful sender is found, a node will drop the sender that is delivering the least amount of useful data to it. It will replace this sender with some other sending peer candidate, essentially reserving a *trial* slot in its sender list. In this way, we are assured of keeping the best senders seen so far and will eliminate senders whose performance deteriorates with changing network conditions.

Likewise, a Bullet sender will periodically evaluate its receivers. Each receiver updates senders of the total received bandwidth. The sender, knowing the amount of data it has sent to each receiver, can determine which receiver is benefiting the least by peering with this sender. This corresponds to the one receiver acquiring the least portion of its bandwidth through this sender. The sender drops this receiver, creating an empty slot for some other trial receiver. This is similar to the concept of *weans* presented in [24].

4. EVALUATION

We have evaluated Bullet's performance in real Internet environments as well as the ModelNet [37] IP emulation framework. While the bulk of our experiments use ModelNet, we also report on our experience with Bullet on the PlanetLab Internet testbed [31]. In addition, we have implemented a number of underlying overlay network trees upon which Bullet can execute. Because Bullet performs well over a randomly created overlay tree, we present results with Bullet running over such a tree compared against an offline greedy bottleneck bandwidth tree algorithm using global topological information described in Section 4.1.

All of our implementations leverage a common development infrastructure called MACEDON [33] that allows for the specification of overlay algorithms in a simple domain-specific language. It enables the reuse of the majority of common functionality in these distributed systems, including probing infrastructures, thread management, message passing, and debugging environment. As a result, we believe that our comparisons qualitatively show algorithmic differences rather than implementation intricacies. Our implementation of the core Bullet logic is under 1000 lines of code in this infrastructure.

Our ModelNet experiments make use of 50 2Ghz Pentium-4's running Linux 2.4.20 and interconnected with 100 Mbps and 1 Gbps Ethernet switches. For the majority of these experiments, we multiplex one thousand instances (overlay participants) of our overlay applications across the 50 Linux nodes (20 per machine). In ModelNet, packet transmissions are routed through *emulators* responsible for accurately emulating the hop-by-hop delay, bandwidth, and congestion of a network topology. In our evaluations, we used four 1.4Ghz Pentium III's running FreeBSD-4.7 as emulators. This platform supports approximately 2-3 Gbps of aggregate simultaneous communication among end hosts. For most of our ModelNet experiments, we use 20,000-node INET-generated topologies [10]. We randomly assign our participant nodes to act as clients connected to one-degree stub nodes in the topology. We randomly select one of these participants to act as the source of the data stream.

Propagation delays in the network topology are calculated based on the relative placement of the network nodes in the plane by INET. Based on the classification in [8], we classify network links as being Client-Stub, Stub-Stub, Transit-Stub, and Transit-Transit depending on their location in the network. We restrict topological bandwidth by setting the bandwidth for each link depending on its type. Each type of link has an associated bandwidth range from which the bandwidth is chosen uniformly at random. By changing these ranges, we vary bandwidth constraints in our topologies. For our experiments, we created three different ranges corresponding to *low*, *medium*, and *high* bandwidths relative to our typical streaming rates of 600-1000 Kbps as specified in Table 1. While the presented ModelNet results are restricted to two topologies with varying bandwidth constraints, the results of experiments with additional topologies all show qualitatively similar behavior.

We do not implement any particular coding scheme for our experiments. Rather, we assume that either each sequence number directly specifies a particular data block and the block offset for each packet, or we are distributing data within the same block for LT Codes, e.g., when distributing a file.

4.1 Offline Bottleneck Bandwidth Tree

One of our goals is to determine Bullet's performance relative to the best possible bandwidth-optimized tree for a given network topology. This allows us to quantify the possible improvements of an overlay mesh constructed using Bullet relative to the best possible tree. While we have not yet proven this, we believe that this problem is NP-hard. Thus, in this section we present a simple greedy offline algorithm to determine the connectivity of a tree likely to deliver a high level of bandwidth. In practice, we are not aware of any scalable online algorithms that are able to deliver the bandwidth of an offline algorithm. At the same time, trees constructed by our algorithm tend to be "long and skinny" making them less resilient to failures and inappropriate for delay sensitive applications (such as multimedia streaming). In addition to any performance comparisons, a Bullet mesh has much lower depth than the bottleneck tree and is more resilient to failure, as discussed in Section 4.6.

Topology classification	Client-Stub	Stub-Stub	Transit-Stub	Transit-Transit
Low bandwidth	300-600	500-1000	1000-2000	2000-4000
Medium bandwidth	800-2800	1000-4000	1000-4000	5000-10000
High bandwidth	1600-5600	2000-8000	2000-8000	10000-20000

Table 1: Bandwidth ranges for link types used in our topologies expressed in Kbps.

Specifically, we consider the following problem: given complete knowledge of the topology (individual link latencies, bandwidth, and packet loss rates), what is the overlay tree that will deliver the highest bandwidth to a set of predetermined overlay nodes? We assume that the throughput of the slowest overlay link (the bottleneck link) determines the throughput of the entire tree. We are, therefore, trying to find the directed overlay tree with the maximum bottleneck link. Accordingly, we refer to this problem as the overlay maximum bottleneck tree (OMBT). In a simplified case, assuming that congestion only exists on access links and there are no lossy links, there exists an optimal algorithm [23]. In the more general case of contention on any physical link, and when the system is allowed to choose the routing path between the two endpoints, this problem is known to be NP-hard [12], even in the absence of link losses. For the purposes of this paper, our goal is to determine a "good" overlay streaming tree that provides each overlay participant with substantial bandwidth, while avoiding overlay links with high end-to-end loss rates.

We make the following assumptions:

1. The routing path between any two overlay participants is fixed. This closely models the existing overlay network model with IP for unicast routing.

2. The overlay tree will use TCP-friendly unicast connections to transfer data point-to-point.

3. In the absence of other flows, we can estimate the throughput of a TCP-friendly flow using a steady-state formula [27].

4. When several (n) flows share the same bottleneck link, each flow can achieve throughput of at most $\frac{c}{n}$, where c is the physical capacity of the link.

Given these assumptions, we concentrate on estimating the throughput available between two participants in the overlay. We start by calculating the throughput using the steady-state formula. We then "route" the flow in the network, and consider the physical links one at a time. On each physical link, we compute the fair-share for each of the competing flows. The throughput of an overlay link is then approximated by the minimum of the fair-shares along the routing path, and the formula rate. If some flow does not require the same share of the bottleneck link as other competing flows (i.e., its throughput might be limited by losses elsewhere in the network), then the other flows might end up with a greater share than the one we compute. We do not account for this, as the major goal of this estimate is simply to avoid lossy and highly congested physical links.

More formally, we define the problem as follows:
Overlay Maximum Bottleneck Tree (OMBT).
Given a physical network represented as a graph $G = (V, E)$, set of overlay participants $P \subset V$, source node ($s \in P$), bandwidth $B : E \rightarrow R^+$, loss rate $L : E \rightarrow [0, 1]$, propagation delay $D : E \rightarrow R^+$ of each link, set of possible overlay links $O = \{(v, w) \mid v, w \in P, v \neq w\}$, routing table $RT : O \times E \rightarrow \{0, 1\}$, find the overlay tree $T = \{o \mid o \in O\}$ ($|T| = |P| - 1$, $\forall v \in P$ there exists a path $o^v = s \rightsquigarrow v$) that maximizes

$$\min_{o|o\in T}(\min(f(o), \min_{e|e\in o} \frac{b(e)}{|\{p \mid p \in T, e \in p\}|}))$$

where $f(o)$ is the TCP steady-state sending rate, computed from round-trip time $d(o) = \sum_{e\in o} d(e) + \sum_{e\in o'} d(e)$ (given overlay link $o = (v, w), o' = (w, v)$), and loss rate $l(o) = 1 - \prod_{e\in o}(1 - l(e))$. We write $e \in o$ to express that link e is included in the o's routing path ($RT(o, e) = 1$).

Assuming that we can estimate the throughput of a flow, we proceed to formulate a greedy OMBT algorithm. This algorithm is non-optimal, but a similar approach was found to perform well [12].

Our algorithm is similar to the Widest Path Heuristic (WPH) [12], and more generally to Prim's MST algorithm [32]. During its execution, we maintain the set of nodes already in the tree, and the set of remaining nodes. To grow the tree, we consider all the overlay links leading from the nodes in the tree to the remaining nodes. We greedily pick the node with the highest throughput overlay link. Using this overlay link might cause us to route traffic over physical links traversed by some other tree flows. Since we do not re-examine the throughput of nodes that are already in the tree, they might end up being connected to the tree with slower overlay links than initially estimated. However, by attaching the node with the highest residual bandwidth at every step, we hope to lessen the effects of after-the-fact physical link sharing. With the synthetic topologies we use for our emulation environment, we have not found this inaccuracy to severely impact the quality of the tree.

4.2 Bullet vs. Streaming

We have implemented a simple streaming application that is capable of streaming data over any specified tree. In our implementation, we are able to stream data through overlay trees using UDP, TFRC, or TCP. Figure 6 shows average bandwidth that each of 1000 nodes receives via this streaming as time progresses on the x-axis. In this example, we use TFRC to stream 600 Kbps over our offline bottleneck bandwidth tree and a random tree (other random trees exhibit qualitatively similar behavior). In these experiments, streaming begins 100 seconds into each run. While the random tree delivers an achieved bandwidth of under 100 Kbps, our offline algorithm overlay delivers approximately 400 Kbps of data. For this experiment, bandwidths were set to the medium range from Table 1. We believe that any degree-constrained online bandwidth overlay tree algorithm would exhibit similar (or lower) behavior to our bandwidth-

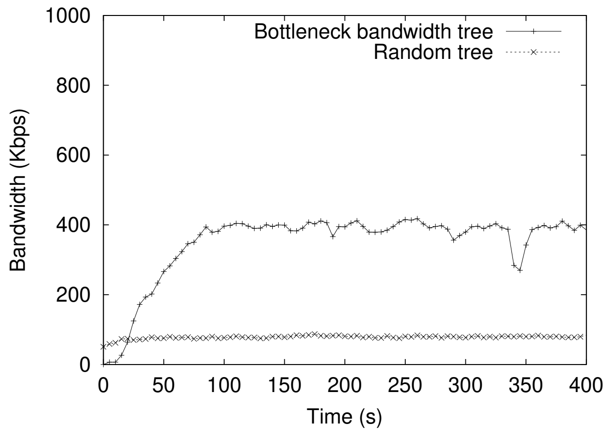

Figure 6: Achieved bandwidth over time for TFRC streaming over the bottleneck bandwidth tree and a random tree.

optimized overlay. Hence, Bullet's goal is to overcome this bandwidth limit by allowing for the perpendicular reception of data and by utilizing disjoint data flows in an attempt to match or exceed the performance of our offline algorithm.

To evaluate Bullet's ability to exceed the bandwidth achievable via tree distribution overlays, we compare Bullet running over a random overlay tree to the streaming behavior shown in Figure 6. Figure 7 shows the average bandwidth received by each node (labeled Useful total) with standard deviation. The graph also plots the total amount of data received and the amount of data a node receives from its parent. For this topology and bandwidth setting, Bullet was able to achieve an average bandwidth of 500 Kbps, fives times that achieved by the random tree and more than 25% higher than the offline bottleneck bandwidth algorithm. Further, the total bandwidth (including redundant data) received by each node is only slightly higher than the useful content, meaning that Bullet is able to achieve high bandwidth while wasting little network resources. Bullet's use of TFRC in this example ensures that the overlay is TCP friendly throughout. The average per-node control overhead is approximately 30 Kbps. By tracing certain packets as they move through the system, we are able to acquire link stress estimates of our system. Though the link stress can be different for each packet since each can take a different path through the overlay mesh, we average link stress due to each traced packet. For this experiment, Bullet has an average link stress of approximately 1.5 with an absolute maximum link stress of 22.

The standard deviation in most of our runs is fairly high because of the limited bandwidth randomly assigned to some Client-Stub and Stub-Stub links. We feel that this is consistent with real Internet behavior where clients have widely varying network connectivity. A time slice is shown in Figure 8 that plots the CDF of instantaneous bandwidths that each node receives. The graph shows that few client nodes receive inadequate bandwidth even though they are bandwidth constrained. The distribution rises sharply starting at approximately 500 Kbps. The vast majority of nodes receive a stream of 500-600 Kbps.

We have evaluated Bullet under a number of bandwidth constraints to determine how Bullet performs relative to the

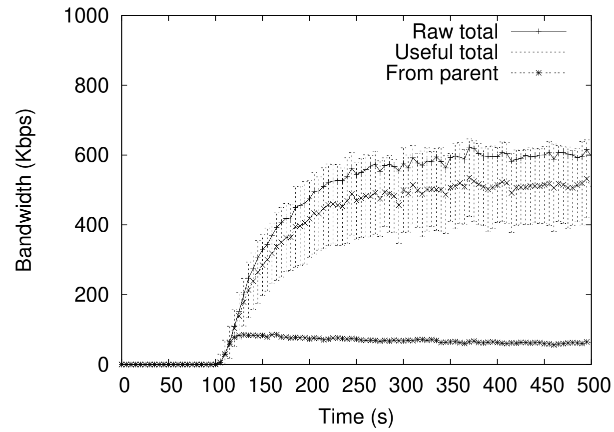

Figure 7: Achieved bandwidth over time for Bullet over a random tree.

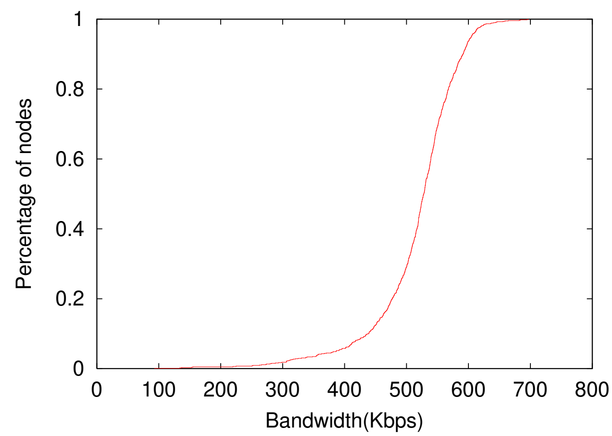

Figure 8: CDF of instantaneous achieved bandwidth at time 430 seconds.

available bandwidth of the underlying topology. Table 1 describes representative bandwidth settings for our streaming rate of 600 Kbps. The intent of these settings is to show a scenario where more than enough bandwidth is available to achieve a target rate even with traditional tree streaming, an example of where it is slightly not sufficient, and one in which the available bandwidth is quite restricted. Figure 9 shows achieved bandwidths for Bullet and the bottleneck bandwidth tree over time generated from topologies with bandwidths in each range.

In all of our experiments, Bullet outperforms the bottleneck bandwidth tree by a factor of up to 100%, depending on how much bandwidth is constrained in the underlying topology. In one extreme, having more than ample bandwidth, Bullet and the bottleneck bandwidth tree are both able to stream at the requested rate (600 Kbps in our example). In the other extreme, heavily constrained topologies allow Bullet to achieve twice the bandwidth achievable via the bottleneck bandwidth tree. For all other topologies, Bullet's benefits are somewhere in between. In our example, Bullet running over our medium-constrained bandwidth topology is able to outperform the bottleneck bandwidth tree by a factor of 25%. Further, we stress that we believe it would

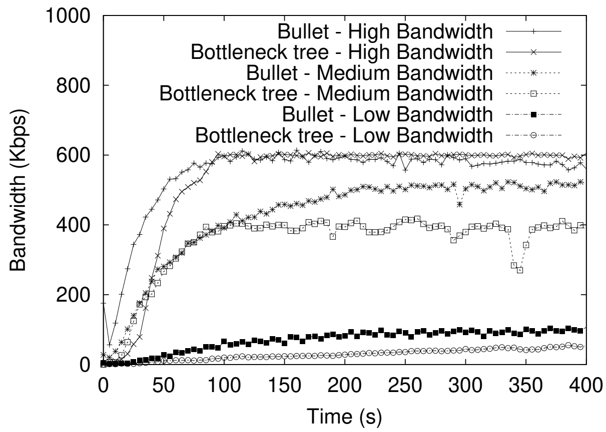

Figure 9: Achieved bandwidth for Bullet and bottleneck tree over time for high, medium, and low bandwidth topologies.

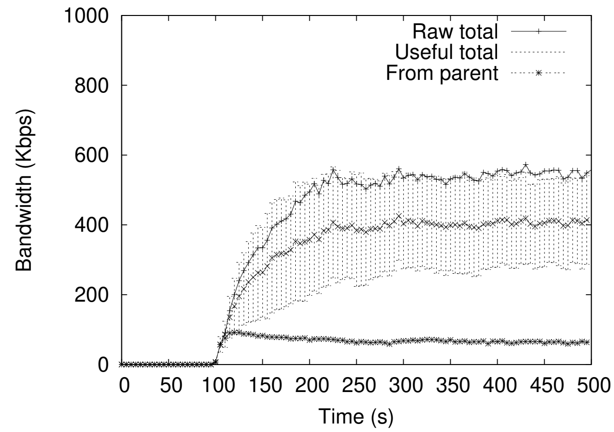

Figure 10: Achieved bandwidth over time using non-disjoint data transmission.

be extremely difficult for any online tree-based algorithm to exceed the bandwidth achievable by our offline bottleneck algorithm that makes use of global topological information. For instance, we built a simple bandwidth optimizing overlay tree construction based on Overcast [21]. The resulting dynamically constructed trees never achieved more than 75% of the bandwidth of our own offline algorithm.

4.3 Creating Disjoint Data

Bullet's ability to deliver high bandwidth levels to nodes depends on its disjoint transmission strategy. That is, when bandwidth to a child is limited, Bullet attempts to send the "correct" portions of data so that recovery of the lost data is facilitated. A Bullet parent sends different data to its children in hopes that each data item will be readily available to nodes spread throughout its subtree. It does so by assigning ownership of data objects to children in a manner that makes the expected number of nodes holding a particular data object equal for all data objects it transmits. Figure 10 shows the resulting bandwidth over time for the non-disjoint strategy in which a node (and more importantly, the root of the tree) attempts to send all data to each of its children (subject to independent losses at individual child links). Because the children transports throttle the sending rate at each parent, some data is inherently sent disjointly (by chance). By not explicitly choosing which data to send its child, this approach deprives Bullet of 25% of its bandwidth capability, when compared to the case when our disjoint strategy is enabled in Figure 7.

4.4 Epidemic Approaches

In this section, we explore how Bullet compares to data dissemination approaches that use some form of epidemic routing. We implemented a form of "gossiping", where a node forwards non-duplicate packets to a randomly chosen number of nodes in its local view. This technique does not use a tree for dissemination, and is similar to lpbcast [14] (recently improved to incorporate retrieval of data objects [13]). We do not disseminate packets every T seconds; instead we forward them as soon as they arrive.

We also implemented a pbcast-like [2] approach for retrieving data missing from a data distribution tree. The idea here is that nodes are expected to obtain most of their data from their parent. Nodes then attempt to retrieve any missing data items through gossiping with random peers. Instead of using gossiping with a fixed number of rounds for each packet, we use anti-entropy with a FIFO Bloom filter to attempt to locate peers that hold any locally missing data items.

To make our evaluation conservative, we assume that nodes employing gossip and anti-entropy recovery are able to maintain full group membership. While this might be difficult in practice, we assume that RanSub [24] could also be applied to these ideas, specifically in the case of anti-entropy recovery that employs an underlying tree. Further, we also allow both techniques to reuse other aspects of our implementation: Bloom filters, TFRC transport, etc. To reduce the number of duplicate packets, we use less peers in each round (5) than Bullet (10). For our configuration, we experimentally found that 5 peers results in the best performance with the lowest overhead. In our experiments, increasing the number of peers did not improve the average bandwidth achieved throughout the system. To allow TFRC enough time to ramp up to the appropriate TCP-friendly sending rate, we set the epoch length for anti-entropy recovery to 20 seconds.

For these experiments, we use a 5000-node INET topology with no explicit physical link losses. We set link bandwidths according to the medium range from Table 1, and randomly assign 100 overlay participants. The randomly chosen root either streams at 900 Kbps (over a random tree for Bullet and greedy bottleneck tree for anti-entropy recovery), or sends packets at that rate to randomly chosen nodes for gossiping. Figure 11 shows the resulting bandwidth over time achieved by Bullet and the two epidemic approaches. As expected, Bullet comes close to providing the target bandwidth to all participants, achieving approximately 60 percent more then gossiping and streaming with anti-entropy. The two epidemic techniques send an excessive number of duplicates, effectively reducing the useful bandwidth provided to each node. More importantly, both approaches assign equal significance to other peers, regardless of the available band-

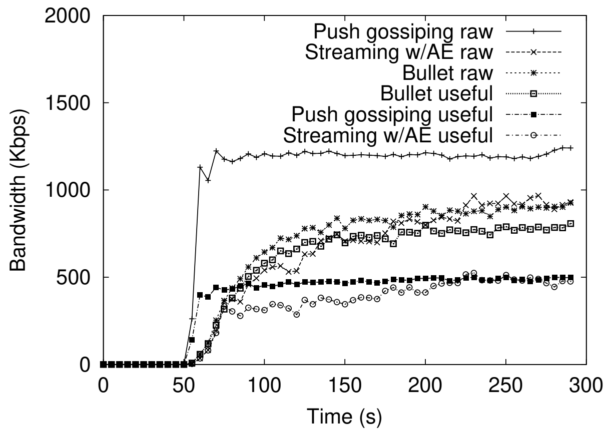

Figure 11: Achieved bandwidth over time for Bullet and epidemic approaches.

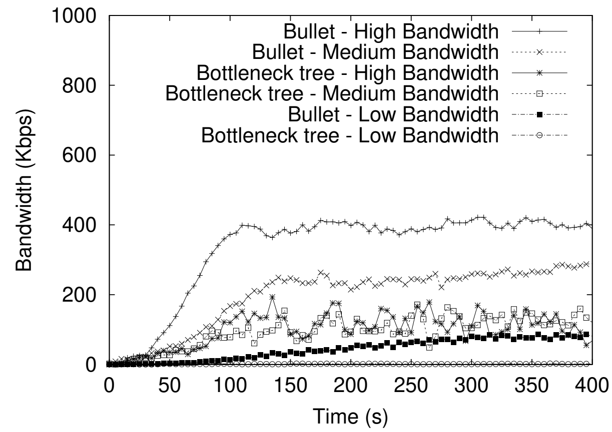

Figure 12: Achieved bandwidths for Bullet and bottleneck bandwidth tree over a lossy network topology.

width and the similarity ratio. Bullet, on the other hand, establishes long-term connections with peers that provide good bandwidth and disjoint content, and avoids most of the duplicates by requesting disjoint data from each node's peers.

4.5 Bullet on a Lossy Network

To evaluate Bullet's performance under more lossy network conditions, we have modified our 20,000-node topologies used in our previous experiments to include random packet losses. ModelNet allows the specification of a *packet loss rate* in the description of a network link. Our goal by modifying these loss rates is to simulate queuing behavior when the network is under load due to background network traffic.

To effect this behavior, we first modify all non-transit links in each topology to have a packet loss rate chosen uniformly random from $[0, 0.003]$ resulting in a maximum loss rate of 0.3%. Transit links are likewise modified, but with a maximum loss rate of 0.1%. Similar to the approach in [28], we randomly designated 5% of the links in the topologies as overloaded and set their loss rates uniformly random from $[0.05, 0.1]$ resulting in a maximum packet loss rate of 10%. Figure 12 shows achieved bandwidths for streaming over Bullet and using our greedy offline bottleneck bandwidth tree. Because losses adversely affect the bandwidth achievable over TCP-friendly transport and since bandwidths are strictly monotonically decreasing over a streaming tree, tree-based algorithms perform considerably worse than Bullet when used on a lossy network. In all cases, Bullet delivers at least twice as much bandwidth than the bottleneck bandwidth tree. Additionally, losses in the low bandwidth topology essentially keep the bottleneck bandwidth tree from delivering any data, an artifact that is avoided by Bullet.

4.6 Performance Under Failure

In this section, we discuss Bullet's behavior in the face of node failure. In contrast to streaming distribution trees that must quickly detect and make tree transformations to overcome failure, Bullet's failure resilience rests on its ability to maintain a higher level of achieved bandwidth by virtue of perpendicular (peer) streaming. While all nodes under a failed node in a distribution tree will experience a temporary disruption in service, Bullet nodes are able compensate for this by receiving data from peers throughout the outage.

Because Bullet, and, more importantly, RanSub makes use of an underlying tree overlay, part of Bullet's failure recovery properties will depend on the failure recovery behavior of the underlying tree. For the purposes of this discussion, we simply assume the worst-case scenario where an underlying tree has no failure recovery. In our failure experiments, we fail one of root's children (with 110 of the total 1000 nodes as descendants) 250 seconds after data streaming is started. By failing one of root's children, we are able to show Bullet's worst-case performance under a single node failure.

In our first scenario, we disable failure detection in RanSub so that after a failure occurs, Bullet nodes request data only from their current peers. That is, at this point, RanSub stops functioning and no new peer relationships are created for the remainder of the run. Figure 13 shows Bullet's achieved bandwidth over time for this case. While the average achieved rate drops from 500 Kbps to 350 Kbps, most nodes (including the descendants of the failed root child) are able to recover a large portion of the data rate.

Next, we enable RanSub failure detection that recognizes a node's failure when a RanSub epoch has lasted longer than the predetermined maximum (5 seconds for this test). In this case, the root simply initiates the next distribute phase upon RanSub timeout. The net result is that nodes that are not descendants of the failed node will continue to receive updated random subsets allowing them to peer with appropriate nodes reflecting the new network conditions. As shown in Figure 14, the failure causes a negligible disruption in performance. With RanSub failure detection enabled, nodes quickly learn of other nodes from which to receive data. Once such recovery completes, the descendants of the failed node use their already established peer relationships to compensate for their ancestor's failure. Hence, because Bullet is an overlay mesh, its reliability characteristics far exceed that of typical overlay distribution trees.

4.7 PlanetLab

This section contains results from the deployment of Bullet over the PlanetLab [31] wide-area network testbed. For

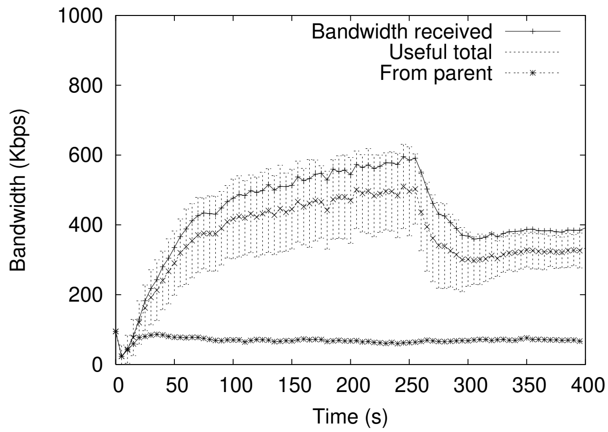

Figure 13: Bandwidth over time with a worst-case node failure and no RanSub recovery.

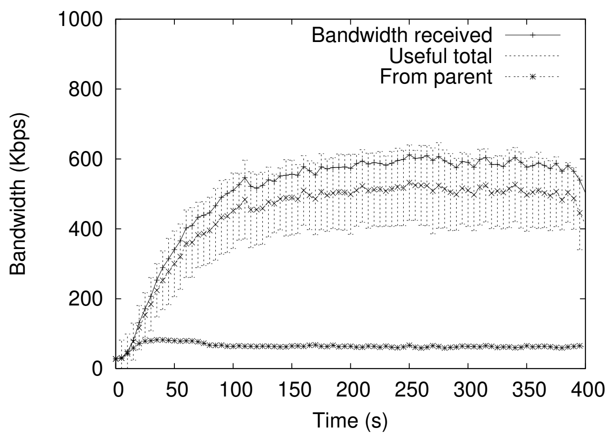

Figure 14: Bandwidth over time with a worst-case node failure and RanSub recovery enabled.

our first experiment, we chose 47 nodes for our deployment, with no two machines being deployed at the same site. Since there is currently ample bandwidth available throughout the PlanetLab overlay (a characteristic not necessarily representative of the Internet at large), we designed this experiment to show that Bullet can achieve higher bandwidth than an overlay tree when the source is constrained, for instance in cases of congestion on its outbound access link, or of overload by a flash-crowd.

We did this by choosing a root in Europe connected to PlanetLab with fairly low bandwidth. The node we selected was in Italy (`cs.unibo.it`) and we had 10 other overlay nodes in Europe. Without global knowledge of the topology in PlanetLab (and the Internet), we are, of course, unable to produce our greedy bottleneck bandwidth tree for comparison. We ran Bullet over a random overlay tree for 300 seconds while attempting to stream at a rate of 1.5 Mbps. We waited 50 seconds before starting to stream data to allow nodes to successfully join the tree. We compare the performance of Bullet to data streaming over multiple handcrafted trees. Figure 15 shows our results for two such trees. The "good" tree has all nodes in Europe located high in the tree, close to the root. We used pathload [20] to measure the

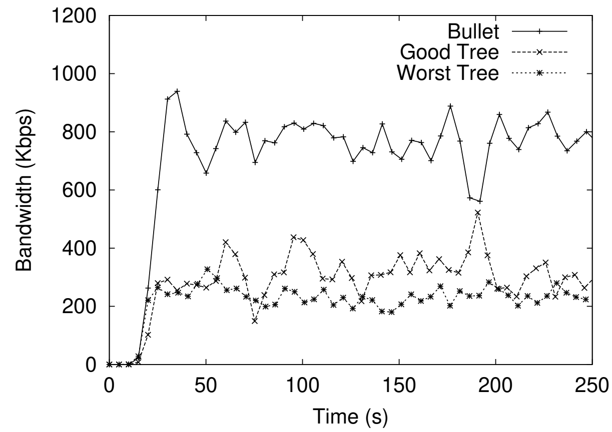

Figure 15: Achieved bandwidth over time for Bullet and TFRC streaming over different trees on PlanetLab with a root in Europe.

available bandwidth between the root and all other nodes. Nodes with high bandwidth measurements were placed close to the root. In this case, we are able to achieve a bandwidth of approximately 300 Kbps. The "worst" tree was created by setting the root's children to be the three nodes with the worst bandwidth characteristics from the root as measured by pathload. All subsequent levels in the tree were set in this fashion.

For comparison, we replaced all nodes in Europe from our topology with nodes in the US, creating a topology that only included US nodes with high bandwidth characteristics. As expected, Bullet was able to achieve the full 1.5 Mbps rate in this case. A well constructed tree over this high-bandwidth topology yielded slightly lower than 1.5 Mbps, verifying that our approach does not sacrifice performance under high bandwidth conditions and improves performance under constrained bandwidth scenarios.

5. RELATED WORK

Snoeren et al. [36] use an overlay mesh to achieve reliable and timely delivery of mission-critical data. In this system, every node chooses n "parents" from which to receive duplicate packet streams. Since its foremost emphasis is reliability, the system does not attempt to improve the bandwidth delivered to the overlay participants by sending disjoint data at each level. Further, during recovery from parent failure, it limits an overlay router's choice of parents to nodes with a level number that is less than its own level number.

The power of "perpendicular" downloads is perhaps best illustrated by Kazaa [22], the popular peer-to-peer file swapping network. Kazaa nodes are organized into a scalable, hierarchical structure. Individual users search for desired content in the structure and proceed to simultaneously download potentially disjoint pieces from nodes that already have it. Since Kazaa does not address the multicast communication model, a large fraction of users downloading the same file would consume more bandwidth than nodes organized into the Bullet overlay structure. Kazaa does not use erasure coding; therefore it may take considerable time to locate "the last few bytes."

BitTorrent [3] is another example of a file distribution system currently deployed on the Internet. It utilizes *trackers* that direct downloaders to random subsets of machines that already have portions of the file. The tracker poses a scalability limit, as it continuously updates the systemwide distribution of the file. Lowering the tracker communication rate could hurt the overall system performance, as information might be out of date. Further, BitTorrent does not employ any strategy to disseminate data to different regions of the network, potentially making it more difficult to recover data depending on client access patterns. Similar to Bullet, BitTorrent incorporates the notion of "choking" at each node with the goal of identifying receivers that benefit the most by downloading from that particular source.

FastReplica [11] addresses the problem of reliable and efficient file distribution in content distribution networks (CDNs). In the basic algorithm, nodes are organized into groups of fixed size (n), with full group membership information at each node. To distribute the file, a node splits it into n equal-sized portions, sends the portions to other group members, and instructs them to download the missing pieces in parallel from other group members. Since only a fixed portion of the file is transmitted along each of the overlay links, the impact of congestion is smaller than in the case of tree distribution. However, since it treats all paths equally, FastReplica does not take full advantage of high-bandwidth overlay links in the system. Since it requires file store-and-forward logic at each level of the hierarchy necessary for scaling the system, it may not be applicable to high-bandwidth streaming.

There are numerous protocols that aim to add reliability to IP multicast. In Scalable Reliable Multicast (SRM) [16], nodes multicast retransmission requests for missed packets. Two techniques attempt to improve the scalability of this approach: probabilistic choice of retransmission timeouts, and organization of receivers into hierarchical local recovery groups. However, it is difficult to find appropriate timer values and local scoping settings (via the TTL field) for a wide range of topologies, number of receivers, etc. even when adaptive techniques are used. One recent study [2] shows that SRM may have significant overhead due to retransmission requests.

Bullet is closely related to efforts that use epidemic data propagation techniques to recover from losses in the non-reliable IP-multicast tree. In pbcast [2], a node has global group membership, and periodically chooses a random subset of peers to send a digest of its received packets. A node that receives the digest responds to the sender with the missing packets in a last-in, first-out fashion. Lbpcast [14] addresses pbcast's scalability issues (associated with global knowledge) by constructing, in a decentralized fashion, a partial group membership view at each node. The average size of the views is engineered to allow a message to reach all participants with high probability. Since lbpcast does not require an underlying tree for data distribution and relies on the push-gossiping model, its network overhead can be quite high.

Compared to the reliable multicast efforts, Bullet behaves favorably in terms of the network overhead because nodes do not "blindly" request retransmissions from their peers. Instead, Bullet uses the summary views it obtains through RanSub to guide its actions toward nodes with disjoint content. Further, a Bullet node splits the retransmission load between all of its peers. We note that pbcast nodes contain a mechanism to rate-limit retransmitted packets and to send different packets in response to the same digest. However, this does not guarantee that packets received in parallel from multiple peers will not be duplicates. More importantly, the multicast recovery methods are limited by the bandwidth through the tree, while Bullet strives to provide more bandwidth to all receivers by making data deliberately disjoint throughout the tree.

Narada [19] builds a delay-optimized mesh interconnecting all participating nodes and actively measures the available bandwidth on overlay links. It then runs a standard routing protocol on top of the overlay mesh to construct forwarding trees using each node as a possible source. Narada nodes maintain global knowledge about all group participants, limiting system scalability to several tens of nodes. Further, the bandwidth available through a Narada tree is still limited to the bandwidth available from each parent. On the other hand, the fundamental goal of Bullet is to increase bandwidth through download of disjoint data from multiple peers.

Overcast [21] is an example of a bandwidth-efficient overlay tree construction algorithm. In this system, all nodes join at the root and migrate down to the point in the tree where they are still able to maintain some minimum level of bandwidth. Bullet is expected to be more resilient to node departures than any tree, including Overcast. Instead of a node waiting to get the data it missed from a new parent, a node can start getting data from its perpendicular peers. This transition is seamless, as the node that is disconnected from its parent will start demanding more missing packets from its peers during the standard round of refreshing its filters. Overcast convergence time is limited by probes to immediate siblings and ancestors. Bullet is able to provide approximately a target bandwidth without having a fully converged tree.

In parallel to our own work, SplitStream [9] also has the goal of achieving high bandwidth data dissemination. It operates by splitting the multicast stream into k stripes, transmitting each stripe along a separate multicast tree built using Scribe [34]. The key design goal of the tree construction mechanism is to have each node be an intermediate node in at most one tree (while observing both inbound and outbound node bandwidth constraints), thereby reducing the impact of a single node's sudden departure on the rest of the system. The join procedure can potentially sacrifice the interior-node-disjointness achieved by Scribe. Perhaps more importantly, SplitStream assumes that there is enough available bandwidth to carry each stripe on every link of the tree, including the links between the data source and the roots of individual stripe trees independently chosen by Scribe. To some extent, Bullet and SplitStream are complementary. For instance, Bullet could run on each of the stripes to maximize the bandwidth delivered to each node along each stripe.

CoopNet [29] considers live content streaming in a peer-to-peer environment, subject to high node churn. Consequently, the system favors resilience over network efficiency. It uses a centralized approach for constructing either random or deterministic node-disjoint (similar to SplitStream) trees, and it includes an MDC [17] adaptation framework based on scalable receiver feedback that attempts to maximize the signal-to-noise ratio perceived by receivers. In the case of on-demand streaming, CoopNet [30] addresses the flash-crowd problem at the central server by redirecting incoming clients to a fixed number of nodes that have previously retrieved portions of the same content. Compared to CoopNet, Bullet provides nodes with a uniformly random subset of the system-wide distribution of the file.

6. CONCLUSIONS

Typically, high bandwidth overlay data streaming takes place over a distribution tree. In this paper, we argue that, in fact, an overlay mesh is able to deliver fundamentally higher bandwidth. Of course, a number of difficult challenges must be overcome to ensure that nodes in the mesh do not repeatedly receive the same data from peers. This paper presents the design and implementation of Bullet, a scalable and efficient overlay construction algorithm that overcomes this challenge to deliver significant bandwidth improvements relative to traditional tree structures. Specifically, this paper makes the following contributions:

- We present the design and analysis of Bullet, an overlay construction algorithm that creates a mesh over any distribution tree and allows overlay participants to achieve a higher bandwidth throughput than traditional data streaming. As a related benefit, we eliminate the overhead required to probe for available bandwidth in traditional distributed tree construction techniques.

- We provide a technique for recovering missing data from peers in a scalable and efficient manner. RanSub periodically disseminates summaries of data sets received by a changing, uniformly random subset of global participants.

- We propose a mechanism for making data disjoint and then distributing it in a uniform way that makes the probability of finding a peer containing missing data equal for all nodes.

- A large-scale evaluation of 1000 overlay participants running in an emulated 20,000 node network topology, as well as experimentation on top of the PlanetLab Internet testbed, shows that Bullet running over a random tree can achieve twice the throughput of streaming over a traditional bandwidth tree.

Acknowledgments

We would like to thank David Becker for his invaluable help with our ModelNet experiments and Ken Yocum for his help with ModelNet emulation optimizations. In addition, we thank our shepherd Barbara Liskov and our anonymous reviewers who provided excellent feedback.

7. REFERENCES

[1] Suman Banerjee, Bobby Bhattacharjee, and Christopher Kommareddy. Scalable Application Layer Multicast. In *Proceedings of ACM SIGCOMM*, August 2002.

[2] Kenneth Birman, Mark Hayden, Oznur Ozkasap, Zhen Xiao, Mihai Budiu, and Yaron Minsky. Bimodal Multicast. *ACM Transaction on Computer Systems*, 17(2), May 1999.

[3] Bittorrent. http://bitconjurer.org/BitTorrent.

[4] Burton Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communication of ACM*, 13(7):422–426, July 1970.

[5] Andrei Broder. On the Resemblance and Containment of Documents. In *Proceedings of Compression and Complexity of Sequences (SEQUENCES'97)*, 1997.

[6] John W. Byers, Jeffrey Considine, Michael Mitzenmacher, and Stanislav Rost. Informed Content Delivery Across Adaptive Overlay Networks. In *Proceedings of ACM SIGCOMM*, August 2002.

[7] John W. Byers, Michael Luby, Michael Mitzenmacher, and Ashutosh Rege. A Digital Fountain Approach to Reliable Distribution of Bulk Data. In *SIGCOMM*, pages 56–67, 1998.

[8] Ken Calvert, Matt Doar, and Ellen W. Zegura. Modeling Internet Topology. *IEEE Communications Magazine*, June 1997.

[9] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. Splitstream: High-bandwidth Content Distribution in Cooperative Environments. In *Proceedings of the 19th ACM Symposium on Operating System Principles*, October 2003.

[10] Hyunseok Chang, Ramesh Govindan, Sugih Jamin, Scott Shenker, and Walter Willinger. Towards Capturing Representative AS-Level Internet Topologies. In *Proceedings of ACM SIGMETRICS*, June 2002.

[11] Ludmila Cherkasova and Jangwon Lee. FastReplica: Efficient Large File Distribution within Content Delivery Networks. In *4th USENIX Symposium on Internet Technologies and Systems*, March 2003.

[12] Reuven Cohen and Gideon Kaempfer. A Unicast-based Approach for Streaming Multicast. In *INFOCOM*, pages 440–448, 2001.

[13] Patrick Eugster, Sidath Handurukande, Rachid Guerraoui, Anne-Marie Kermarrec, and Petr Kouznetsov. Lightweight Probabilistic Broadcast. *To appear in ACM Transactions on Computer Systems*.

[14] Patrick Eugster, Sidath Handurukande, Rachid Guerraoui, Anne-Marie Kermarrec, and Petr Kouznetsov. Lightweight Probabilistic Broadcast. In *Proceedings of The International Conference on Dependable Systems and Networks (DSN)*, 2001.

[15] Sally Floyd, Mark Handley, Jitendra Padhye, and Jorg Widmer. Equation-based congestion control for unicast applications. In *SIGCOMM 2000*, pages 43–56, Stockholm, Sweden, August 2000.

[16] Sally Floyd, Van Jacobson, Ching-Gung Liu, Steven McCanne, and Lixia Zhang. A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing. *IEEE/ACM Transactions on Networking*, 5(6):784–803, 1997.

[17] Vivek K Goyal. Multiple Description Coding: Compression Meets the Network. *IEEE Signal Processing Mag.*, pages 74–93, May 2001.

[18] Yang hua Chu, Sanjay Rao, and Hui Zhang. A Case For End System Multicast. In *Proceedings of the ACM Sigmetrics 2000 International Conference on Measurement and Modeling of Computer Systems*, June 2000.

[19] Yang hua Chu, Sanjay G. Rao, Srinivasan Seshan, and Hui Zhang. Enabling Conferencing Applications on the Internet using an Overlay Multicast Architecture. In *Proceedings of ACM SIGCOMM*, August 2001.

[20] Manish Jain and Constantinos Dovrolis. End-to-end Available Bandwidth: Measurement Methodology, Dynamics, and Relation with TCP Throughput. In *Proceedings of SIGCOMM 2002*, New York, August 19–23 2002.

[21] John Jannotti, David K. Gifford, Kirk L. Johnson, M. Frans Kaashoek, and Jr. James W. O'Toole. Overcast: Reliable Multicasting with an Overlay Network. In *Proceedings of Operating Systems Design and Implementation (OSDI)*, October 2000.

[22] Kazaa media desktop. http://www.kazaa.com.

[23] Min Sik Kim, Simon S. Lam, and Dong-Young Lee.

Optimal Distribution Tree for Internet Streaming Media. Technical Report TR-02-48, Department of Computer Sciences, University of Texas at Austin, September 2002.

[24] Dejan Kostić, Adolfo Rodriguez, Jeannie Albrecht, Abhijeet Bhirud, and Amin Vahdat. Using Random Subsets to Build Scalable Network Services. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, March 2003.

[25] Michael Luby. LT Codes. In *In The 43rd Annual IEEE Symposium on Foundations of Computer Science*, 2002.

[26] Michael G. Luby, Michael Mitzenmacher, M. Amin Shokrollahi, Daniel A. Spielman, and Volker Stemann. Practical Loss-Resilient Codes. In *Proceedings of the 29th Annual ACM Symposium on the Theory of Computing (STOC '97)*, pages 150–159, New York, May 1997. Association for Computing Machinery.

[27] Jitedra Padhye, Victor Firoiu, Don Towsley, and Jim Krusoe. Modeling TCP Throughput: A Simple Model and its Empirical Validation. In *ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 303–314, Vancouver, CA, 1998.

[28] Venkata N. Padmanabhan, Lili Qiu, and Helen J. Wang. Server-based Inference of Internet Link Lossiness. In *Proceedings of the IEEE Infocom*, San Francisco, CA, USA, 2003.

[29] Venkata N. Padmanabhan, Helen J. Wang, and Philip A. Chou. Resilient Peer-to-Peer Streaming. In *Proceedings of the 11th ICNP*, Atlanta, Georgia, USA, 2003.

[30] Venkata N. Padmanabhan, Helen J. Wang, Philip A. Chou, and Kunwadee Sripanidkulchai. Distributing Streaming Media Content Using Cooperative Networking. In *ACM/IEEE NOSSDAV*, 2002.

[31] Larry Peterson, Tom Anderson, David Culler, and Timothy Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proceedings of ACM HotNets-I*, October 2002.

[32] R. C. Prim. Shortest Connection Networks and Some Generalizations. In *Bell Systems Technical Journal*, pages 1389–1401, November 1957.

[33] Adolfo Rodriguez, Sooraj Bhat, Charles Killian, Dejan Kostić, and Amin Vahdat. MACEDON: Methodology for Automatically Creating, Evaluating, and Designing Overlay Networks. Technical Report CS-2003-09, Duke University, July 2003.

[34] Antony Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. SCRIBE: The Design of a Large-scale Event Notification Infrastructure. In *Third International Workshop on Networked Group Communication*, November 2001.

[35] Stefan Savage. Sting: A TCP-based Network Measurement Tool. In *Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems (USITS-99)*, pages 71–80, Berkeley, CA, October 11–14 1999. USENIX Association.

[36] Alex C. Snoeren, Kenneth Conley, and David K. Gifford. Mesh-Based Content Routing Using XML. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, October 2001.

[37] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostić, Jeff Chase, and David Becker. Scalability and Accuracy in a Large-Scale Network Emulator. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.