High-Bandwidth Data Dissemination for Large-Scale Distributed Systems

DEJAN KOSTIĆ
Ecole Polytechnique Fédérale de Lausanne
ALEX C. SNOEREN, AMIN VAHDAT, RYAN BRAUD, CHARLES KILLIAN, and JAMES W. ANDERSON
University of California, San Diego
JEANNIE ALBRECHT
Williams College
and
ADOLFO RODRIGUEZ and ERIK VANDEKIEFT
IBM

This article focuses on the multireceiver data dissemination problem. Initially, IP multicast formed the basis for efficiently supporting such distribution. More recently, overlay networks have emerged to support point-to-multipoint communication. Both techniques focus on constructing trees rooted at the source to distribute content among all interested receivers. We argue, however, that trees have two fundamental limitations for data dissemination. First, since all data comes from a single parent, participants must often continuously probe in search of a parent with an acceptable level of bandwidth. Second, due to packet losses and failures, available bandwidth is monotonically decreasing down the tree.

To address these limitations, we present Bullet, a data dissemination mesh that takes advantage of the computational and storage capabilities of end hosts to create a distribution structure where a node receives data in parallel from multiple peers. For the mesh to deliver improved bandwidth and reliability, we need to solve several key problems: (i) disseminating disjoint data over the mesh, (ii) locating missing content, (iii) finding who to peer with (peering strategy), (iv) retrieving data at the

Authors' current addresses: D. Kostić, Ecole Polytechnique Fédérale de Lausanne (EPFL), School of Computer & Communication Sciences, Station 14, CH-1015 Lausanne, Switzerland; email: Dejan.Kostic@epfl.ch; A. C. Snoeren, A. Vahdat, R. Braud, C. Killian, and J. W. Anderson, Department of Computer Science and Engineering, University of California, San Diego, 9500 Gilman Drive, M/C 0404, La Jolla, CA 92093; email: {snoeren,vahdat,rbraud, ckillian,jwanderson}@cs.ucsd.edu; J. Albrecht, Department of Computer Science, Williams College, 47 Lab Campus Dr., Williamstown, MA 01267; email: jeannie@cs.williams.edu; A. Rodriguez, WebSphere Technology Institute, IBM, Research Triangle Park, North Carolina 27709; email: adolfo@us.ibm.com; E. Vandekieft, NEC, 10850 Gold Center Dr., Ste. 200, Rancho Cordova, CA 95670; email: erik.vandekieft@necam.com.

© 2008 ACM 0734-2071/2008/02-ART3 $5.00 DOI 10.1145/1328671.1328674 http://doi.acm.org/10.1145/1328671.1328674

right rate from all peers (flow control), and (v) recovering from failures and adapting to dynamically changing network conditions. Additionally, the system should be self-adjusting and should have few user-adjustable parameter settings. We describe our approach to addressing all of these problems in a working, deployed system across the Internet. Bullet outperforms state-of-the-art systems, including BitTorrent, by 25-70% and exhibits strong performance and reliability in a range of deployment settings. In addition, we find that, relative to tree-based solutions, Bullet reduces the need to perform expensive bandwidth probing.

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

Additional Key Words and Phrases: Bandwidth, overlays, peer-to-peer

ACM Reference Format:

Kostić, D., Snoeren, A. C., Vahdat, A., Braud, R., Killian, C., Anderson, J. W., Albrecht, J., Rodriguez, A., and Vandekieft, E. 2008. High-bandwidth data dissemination for large-scale distributed systems. ACM Trans. Comput. Syst. 26, 1, Article 3 (February 2008), 61 pages. DOI = 10.1145/1328671.1328674 http://doi.acm.org/10.1145/1328671.1328674

1. INTRODUCTION

Single-source high-bandwidth data dissemination is fundamental to a wide range of distributed applications. Here, some large content (e.g., a file) produced at a central location needs to be quickly and efficiently transmitted to a set of interested users spread across a wide-area network. Streaming is a related and emerging application area. Streaming is similar to file distribution, except that the source produces new content at a fixed rate and the system participants are "playing the stream" while receiving the content. Here, the goal is to deliver content to all receivers at a relatively steady rate as measured over relatively small time scales. To demonstrate the importance of high-bandwidth data dissemination, we describe how this model benefits four important distributed applications.

—*Software and virus signature updates.* Recently, it has become crucial to disseminate security-related software updates as quickly as possible to hundreds of millions of end hosts to prevent malicious users from gaining access to PCs. Software vendors face significant problems while attempting to meet this goal. For example, the security-laden Microsoft Windows XP Service Pack 2 size is 260 MB in its full form, and it took more than 2 months to distribute it to approximately 90 million users [InformationWeek 2004]. During that time, the users' machines remained exposed to malicious attacks, despite the fact that fixes had already been developed. Moreover, these downloads accounted for only less than half of the total 200 million users, leaving the remainder vulnerable. While not all of the delay can be attributed to performance limitations, certainly quickly and reliably distributing hundreds of megabytes to 100 million users is a challenging task. By timely disseminating security updates, we can thwart data loss, identity theft, and potential DDoS attacks that cause millions of dollars in damage.

—*Video distribution and multimedia streaming.* Today's end-user PCs come with powerful CPUs, large disks, and broadband Internet connections. These factors have enabled new multimedia applications, such as movie downloads, real-time streaming, and video-on-demand. Multimedia traffic has recently surpassed Web traffic [Saroiu et al. 2002], and there are often *flash crowds* for new content when it appears [Padmanabhan et al. 2003b]. As a result, media servers often become overloaded during popular events and deny service to a large fraction of potential viewers.

—*Content distribution within a CDN.* Content distribution networks (CDNs) appeared as a response to performance and reliability issues with Web content delivery. CDNs deploy large numbers of topologically dispersed machines. However, CDNs do not solve all Internet content delivery problems. For example, when the CDN server does not have the requested object cached, it must fetch it from the CDN customer's Web site. This model is problematic because simultaneous requests from CDN servers can overwhelm the origin Web server. Using an efficient data distribution mechanism among the CDN servers will not overwhelm the origin server and will make the new content available more efficiently to end users.

—*Running jobs on planetary testbeds.* Experiments on planetary scale testbeds, such as PlanetLab and the Grid, cannot run until the executable and the associated data sets are copied to potentially hundreds or thousands of machines. Since these environments often do not offer a high-performance globally visible data repository, users currently resort to ad hoc file distribution techniques that can overwhelm individual servers. Experimentation with a new distributed system often requires several tens of runs to debug the software. Setting up the required experimental infrastructure and copying the executable as quickly as possible each time increases productivity and reduces the overall time to develop a new distributed system or complete a scientific computation.

There are a number of requirements for any system aiming to realize this vision of an efficient and reliable content distribution infrastructure. First, since the system must support millions of receivers, it should be *scalable*. In addition to avoiding global operations, scalability requires that the system not place an arbitrary number of participant identities in its messages. Second, no human administrator can be expected to administer millions of end hosts; hence the system should be *self-organizing*. Third, we do not want a single point of failure; therefore we seek a *decentralized* solution. Further, for maximum performance, the system should be *efficient*. This requirement translates into low levels of control traffic and duplicate data transmission. An additional requirement in the case of file distribution is *reliability*; a file is practically useless unless it is received in its entirety.

From the networking standpoint, this large-scale system ought to be *congestion-friendly*. This means that all data flows should obey the congestion signals sent by the network and should not consume more than their fair share of capacity of any physical link. The system should *adapt* to the changes in receiver membership and the characteristics of the wide- area network. In addition, we also want to minimize the number of "magic constants" (user-adjustable parameter settings) because in many cases it is impossible to pick constant values that perform well across a range of conditions.

In this article, we present the design and the analysis of Bullet [Kostić et al. 2003b], an algorithm for constructing an overlay mesh for high-bandwidth data dissemination. In a mesh, a node receives data in parallel from multiple peers for high throughput and good resilience to network dynamics. At a high level, we use the following approaches in Bullet. First, we use a source sending strategy that promotes data diversity. Instead of having a source attempt to send the same data to each of its children, the source deliberately sends disjoint data to each of the receivers connected to it. Moreover, for streaming at a fixed rate, each node uses extra available bandwidth to distribute data in a uniform way that makes the probability of finding a peer containing missing data equal for all nodes. Second, to address the issue of finding a set of peers and locating missing content, we employ a scalable and decentralized mechanism for *sampling global state*. Third, to pick an optimal set of peers under changing network conditions, we augment the "first-level" node sampling mechanism with sampling at the network level. Specifically, nodes maintain trial slots to try out peers that seem to have high level of disjoint data and were picked after the first sampling phase. Finally, we *implicitly* probe network characteristics as part of active data transfers rather than explicitly probing for network bandwidth. This feature is in contrast with other systems and overlay tree mechanisms.

We implemented and evaluated a 1000-node Bullet system in a live emulated environment, under realistic network conditions, and ran Bullet across the PlanetLab wide-area testbed. In the presence of congestion induced by cross-traffic, Bullet can achieve twice the performance of a bandwidth-optimized tree computed offline with full network information. As an added benefit, Bullet eliminates the need for probing for available bandwidth in traditional overlay tree construction techniques. Further, Bullet outperforms epidemic approaches that use gossiping or antientropy to propagate data.

Contemporaneous to our work on Bullet, several other mesh-based data dissemination protocols were proposed, including BitTorrent [Cohen 2003] and SplitStream [Castro et al. 2003]. To understand some of the tradeoffs associated with the different design decisions made in these systems, we set out to systematically identify and evaluate the effects of various architectural decisions on the performance of mesh-based data dissemination. Throughout this article, we weave the description of our experiences with Bullet, and its evolution to a new system called Bullet′ [Kostić et al. 2005] (Bullet prime). Bullet′ uses adaptive mechanisms for maintaining a high-download rate under dynamic network conditions to match the underlying network topology. We conduct a detailed evaluation of design space parameters for mesh-based distribution systems and performance evaluation of a number of competing systems running in both controlled emulation environments and live across the Internet. Bullet′ outperforms state-of-the-art systems, including BitTorrent, by 25–70% and exhibits strong performance and reliability in a range of deployment settings. This level of performance does not come at the network's expense; both Bullet and Bullet′ have low control overhead (approximately 30 kb/s) and are congestion-friendly.

The remainder of this article is organized as follows. Section 2 describes the design space for all high-bandwidth data dissemination systems, Section 3

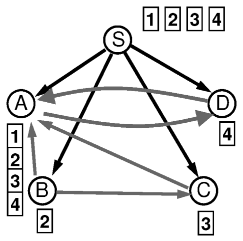

Fig. 1. High-level view of a single-source high-bandwidth data dissemination system.

describes the architectural overview of Bullet and Bullet′ and the system components of our systems. We describe the specific implementation details of Bullet and Bullet′, as well as discuss particular strategies, in Section 4. Section 5 evaluates Bullet's performance for a variety of network topologies, compares it to existing multicast techniques, and presents testing our experimental strategies and comparisons with other file distribution systems. We place our work in the context of the related work in Section 6. Finally, Section 7 presents our conclusions.

2. DESIGN PRINCIPLES AND PATTERNS

In this section we examine the design principles for high-bandwidth data dissemination, exemplified by the following motivating large-scale distributed applications: (i) multimedia streaming and (ii) file (content) distribution. When necessary, we make the distinction between these two scenarios.

Throughout the article, we assume that the source transmits the data as a sequence of data objects (or simply *blocks* when distributing a file) that serve as the smallest transfer unit. Otherwise, peers would not be able to help each other until they had the entire content downloaded. In addition, we concentrate on the case when the source of the content is the only node that has the content initially, and wishes to disseminate it to a large group of receivers as quickly as possible. This usage scenario corresponds to a flash-crowd retrieving popular content over the Internet. We use Figure 1 to demonstrate the general behavior of high-bandwidth data dissemination systems. The source is marked S, and it has content comprising four data objects that it wishes to disseminate (1–4). It sends these data objects in parallel to the four nodes, A, B, C, and D. These nodes establish peering relationships, and in this example a receiver A has three senders (B, C, and D) sending data to it, apart from the source. A and D have a symmetric peering relationship; A and B do not.

The design of any large-scale high-bandwidth data dissemination system can be decomposed into a set of decisions regarding the fundamental tenets of peer-to-peer applications and data transfer. As we see them, these tenets are push versus pull, encoding data, finding the right peers, methods for requesting data from those peers, and serving data to others in an effective manner. Additionally, an important design consideration is whether the system should be fair to participants or focus on being fast. In all cases, however, the underlying goal is to *always keep your incoming pipe full of new data*. This means that nodes must prevent duplicate data transmission, restrict control overhead in favor of distributing data, and adapt to changing network conditions. In the following sections, we enumerate these fundamental decisions in more detail.

2.1 Push or Pull

Options for distributing content can be categorized by whether data is pushed from sources to destinations, pulled from sources by destinations, or a hybrid of the two. Traditional streaming applications typically choose the push methodology because data has low latency and is coming at a constant rate, and all participants are supposed to get all the data at the same time. To increase capacity, nodes may simultaneously receive data from multiple senders, and in a push system, they must then devote overhead to keeping their senders informed about what they have to prevent receipt of duplicates. Alternately, systems can use a pull mechanism where receivers must first learn of what data exists and which nodes have it, and then request it. This has the advantage that receivers, not senders, are in control of the size and specification of the data that is outstanding to them, which allows them to control and adapt to their environments more easily. However, this two-step process of discovery followed by requesting means additional delay before receiving data and extra control messages in some cases.

2.2 Encoded or Unencoded

The simplest way of sending content involves sending the original, or *unencoded*, data objects into the overlay. An advantage of this approach for large file distribution is that receivers typically do not have to fit the entire file into physical memory to sustain high performance. Incoming data objects can be cached in memory, and later written to disk. Data objects only have to be read when the peers request them. Even if the data objects are not in memory and have to be fetched from the disk, pipelining techniques can be used to absorb the cost of disk reads. As a downside, sending the unencoded file might expose the "last block" problem of some file distribution mechanisms, when it becomes difficult for a node to locate and retrieve the last few data objects of the file.

Recently, a number of erasure-correcting codes that implement the "digital fountain" [Byers et al. 1998] approach were suggested by researchers [Luby 2002; Luby et al. 1997; Maymounkov and Mazieres 2003]. When the source encodes the file with these codes, *any* $(1+\epsilon)n$ correctly received *encoded* data objects are sufficient to reconstruct the original n data objects, with the typically low *reception overhead* (ϵ) of 0.03–0.05. These codes hold the potential of removing the "last block" problem, because there is no need for a receiver to acquire any particular data object, as long as it recovers a sufficient number of distinct data objects. We assume that only the source is capable of encoding the file, and do not consider the potential benefits of *network coding* [Ahlswede et al. 2000], where intermediate nodes can produce encoded packets from the ones they have received thus far.

Based on the publicly available specification, we implemented *rateless* erasure codes [Maymounkov and Mazieres 2003]. Although it is straightforward to implement these codes with a low CPU encoding and decoding overhead, they exhibit some performance artifacts that are relevant from a systems perspective. First, the reconstruction of the file cannot make significant progress until a significant number of the encoded data objects is successfully received. Even with n received data objects (i.e., corresponding to the original file size), only 30% of the file content can be reconstructed [Krohn et al. 2004]. Further, since the encoded data objects are computed by XOR-ing random sets of original data objects, the decoding stage requires random access to all of the reconstructed file data objects. This pattern of access, coupled with the bursty nature of the decoding process, causes increased decoding time due to disk swapping if all of the decoded file data objects cannot fit into physical memory.[1] Consequently, the source is forced to transmit the file as a series of *segments* that can fit into physical memory of the receivers. Even if all receivers have homogeneous memory size, this arrangement presents a few problems. First, the source needs to decide when to start transmitting encoded data objects that belong to the next segment. If the file distribution mechanism exhibits considerable latency between the time a data object is first generated and the time when nodes receive it, the source might send too many unnecessary data objects into the overlay. Second, receivers need to simultaneously locate and retrieve data belonging to multiple segments. Opening too many TCP connections can also affect overall performance. Therefore, the receivers have to locate enough senders hosting the segments they are interested in, while still being able to fill their incoming pipe.

We have observed a four percent overhead when encoding and decoding files of tens of megabytes in size. Although mathematically possible, it is difficult to make this overhead significantly smaller via parameter settings or a large number of data objects. Since the data objects should be sufficiently large to overcome the fixed overhead due to per-data object headers, we cannot use an excessive number of data objects. Similarly, since a file consisting of a large number of data objects may not fit into physical memory, a segment may not have enough data objects to reduce the decoding overhead. Finally, the decoding process is sensitive to the number of recovered degree-one (unencoded) data objects that are generated with relatively low probability (e.g., 0.01). These data objects are necessary to start the decoding process and without a sufficient number of these data objects the decoding process cannot complete.

Another class of “digital fountain” codes, called *Raptor codes* [Shokrollahi 2003], have recently been developed. These codes are *systematic*, meaning that a system using these codes first transmits the unencoded blocks, and only then starts transmitting the encoded blocks. The presence of unencoded blocks reduces the memory footprint required for decoding a file. These blocks make the scheme more desirable for streaming because a segment of original data that cannot be reconstructed is useless, if we assume that a stream is broken into segments. The *reception overhead* of Raptor codes is even lower than for the rateless and LT codes, often as low as 2%.

[1]We are assuming a memory-efficient implementation of the codes that releases the memory occupied by the encoded data object when all the original data objects that were used in in its construction are available. Performance can be even worse if the encoded data objects are kept until the file is reconstructed in full.

Other encoding schemes might be used to improve efficiency of real-time streaming. If multimedia data is being streamed to a set of heterogeneous receivers with variable bandwidth, MDC [Goyal 2001] allows receivers obtaining different subsets of the data to still maintain a usable multimedia stream. We quantify the potential benefits of using encoding at the source in Section 5.12.

2.3 Peering Strategy

In both the streaming and the file distribution cases, a node receives data by peering with neighbors and receiving data objects from them. To enable simultaneous data retrieval from multiple peers, each node requires techniques to learn about suitable remote peers, selecting those that have useful and sufficient levels of available bandwidth, and determining the ideal set of peers which can optimize the incoming bandwidth of useful data. Ideally, a node would have perfect and timely information about the distribution of data objects throughout the system and would be able to download any data object from any other peer, but any such approach requiring global knowledge cannot scale. Instead, the system must approximate the peering decisions. It can do this by using a centralized coordination point, though constantly updating that point with updates of data objects received would also not scale, while also adding a single point of failure. Alternatively, a node could simply maintain a fixed set of peers, though this approach would suffer from changing network and node conditions or an initially poor selection of peers. One might also imagine using a DHT to coordinate location of nodes with given data objects. While we have not tested this approach, we reason that it would not perform well due to the extra overhead required for locating nodes with each data object. Overall, a good approach to picking peers would be one which neither causes nodes to maintain global knowledge, nor to communicate with a large number of nodes, but manages to locate peers which have a lot of data to give it. A good peering strategy will also allow the node to maintain a set of peers small enough to minimize control overhead, but large enough to keep the pipe full in the face of changing network conditions.

2.4 Request Strategy

In either a push- or pull-based system, there has to be a decision made about which data objects should be queued to send to which peers. In a pull-based system, this is a request for data object. Therefore we call this the *request strategy*. For the request strategy, we need to answer several important questions.

First, what is the best order for requesting data objects? For example, if all nodes make requests for the data objects in the same order, the senders in peering relationships will be sending the same data to all the peers, and there would be very little opportunity for nodes to help each other to speed up the overall progress. Sometimes, for example, when streaming live data, the source produces the content at a fixed rate and effectively controls the amount of available data in the overlay. The receivers' "play points," defining the positions within the stream currently being played, are also expected to be close to each other. When compared to file distribution, both of these factors might limit the choices in requesting a data object while streaming live content.

Second, for any given data object, more than one of the senders might have it. How does the node choose the sender that is going to provide this particular data object, or in a pull-based system, how can senders prevent queuing the same data object for the same node? Further, should a node request the data object immediately after it learns about its existence at a sender, or should it wait until some of its other peers acquire the same data object? There is a tradeoff, because reacting quickly might bring the data object to this node sooner, and make it available for its own receivers to download sooner. However, the first sender over time that has this data object might not be the best one to serve it; in this case it might be prudent to wait a bit longer, because the data object download time from this sender might be high due to low available bandwidth within the network.

Third, how much data should be requested from any given sender? Requesting too few data objects might not fill the node's incoming pipe, whereas requesting too much data might force the receiver to wait too long for a data object that it could have requested and received from some other node.

Finally, where should the request logic be placed? One option is to have the receiver make explicit requests for data objects, which comes at the expense of maintaining data structures that describe the availability of each data object, the time of requests, etc. In addition, this approach might incur considerable CPU overhead for choosing the next data object to retrieve. If this logic is in the critical path, the throughput in high-bandwidth settings may suffer. Another option is to place the decision-making at the sender. This approach makes the receiver simpler, because it might just need to keep the sender up-to-date with a *digest* of data objects it currently has. Since a node might implicitly request the same data object from multiple senders by not having that data object in the digests, this approach is highly resilient. On the other hand, duplicate data objects could be sent from multiple senders if senders do not synchronize their behavior. Further, message overhead will be higher than in the receiver-driven approach due to digests.

2.5 Sending Strategies

All techniques for distributing content will need a strategy for sending data to peers to optimize the performance of the distribution. We define the sending strategy as "given a set of data items destined for a particular receiver, in what order should they be sent?" This is differentiated from the request strategy in that it is concerned with the *order* of queued data objects rather than *which* data objects to queue. Here, we separate the strategy of the single source from that of the many peers, since for any data distribution to be successful, the source must share all parts of the content with others. In addition, any inefficiency in the source sending strategy (e.g., duplicate data) has a profound effect on overall performance.

2.5.1 *Source.* For file distribution, we consider the case in this article where the source is available to send file data objects for the entire duration of a session. This puts it in a unique position to be able to both help all nodes, and to affect the performance of the entire download. As a result, it is especially important that the source not send the same data twice before sending the entire file once. Otherwise it may prevent fast nodes from completing because it is still "hoarding" the last data object. This can be accomplished by splitting the file in various ways to send to its peers. The source must also consider what kinds of reliability and retransmission it should use and what happens in the face of peer failure. For example, suppose the source sends out each data object exactly once. It then must handle the case when a peer that receives a particular data object crashes before sending it to other receivers. This problem is not as pronounced for nonsource nodes, where other peers can assist one another.

For streaming at a fixed rate, the source has the opportunity to use its available outgoing bandwidth in an attempt to improve the overall bandwidth achieved by receivers. Conceptually, one way of accomplishing this is by sending duplicate data after satisfying all data requests from receivers.

2.5.2 *Nonsource.* There are several options on the order to send nodes data. All of them fulfill the near-term goal for keeping a node's pipe full of useful data. But a forward-thinking sender has some other options available to it which help future downloads, by making it easier for nodes to locate disjoint data. Consider that a node A will send data objects 1, 2, and 3 to receivers B and C. If it sends them in numerical order each time, B and C will both get 1 first. Assuming the same transmission rate to both nodes, after completing transmission of object 1, the utility to nodes who have peered with both B and C is just 1 though, because there is only 1 new data object available. But if node A sends data objects in different orders, the utility to peers of B and C is doubled. A good sending strategy will create the greatest diversity of data objects in the system. In the streaming case, the nonsource nodes can try to aid the overall recovery of data by prudently using their outbound bandwidth.

2.6 Fair-first or Fast-first

An important consideration for the design of a file distribution or a streaming system is whether it is the highest priority that it be fair to network participants, or fast. Some protocols, like BitTorrent and SplitStream, have some notion of fairness and try to make all nodes both give and take. BitTorrent does this by using tit-for-tat [Cohen 2003] in its sending and requesting strategies, assuring that peers share and share alike. SplitStream does this by requiring that each node is forwarding at least as much as it is receiving. It is a nice side effect that, in a symmetric network, this equal sharing strategy can be close to optimal, since all peers can offer as much as they can take, and one peer is largely as good as the next. Techniques have been proposed [Sung et al. 2006] to provide incentives to nodes to contribute more bandwidth than they need.

But this decision fundamentally prevents the possibility that some nodes have more to offer and can be exploited to do so. It is clear that there exist scenarios where there are tradeoffs between being fair and being fast. The appropriate choice largely depends on the use of the application, financial matters, and the mindset of the users in the system. Experience with some peer-to-peer systems indicates that some nodes are willing to give more than they take, while

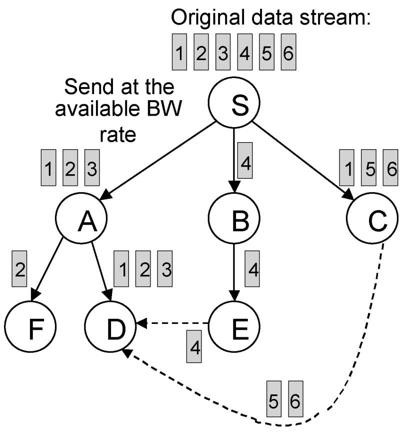

Fig. 2. High-level view of Bullet's operation.

other nodes cannot share fairly due to network configurations and firewalls. In the remainder of the section, we assume that nodes are willing to cooperate both during and after the data dissemination, and do not consider enforcing fairness. Instead of considering fairness, we concentrate on performance and reliability. Consider the following scenarios where a single entity has full control over all nodes participating in content distribution: (i) distribution of content among CDN nodes, (ii) infrastructure nodes in a video distribution system, (iii) transmitting the executable and datafiles as quickly as possible in a wide-area testbed (e.g., PlanetLab).

3. ARCHITECTURAL OVERVIEW AND BACKGROUND

In this section, we start with the high-level overviews of Bullet and Bullet′, our high-bandwidth data dissemination systems. We started with Bullet, envisioning a general-purpose data dissemination mechanism. We demonstrated superior streaming performance relative to overlay trees and gossiping mechanisms. We then shifted our attention to the large-file distribution problem, and identified a number of shortcomings with Bullet in this role. We describe our findings as discussions throughout the article. We then created Bullet′ as an embodiment of best practices as well as new techniques for adapting to dynamic network conditions. In this section, we include the background on each of the techniques that we employ as fundamental building blocks for our work. Section 4 then presents the details of the Bullet and Bullet′ architectures.

3.1 Bullet

Our approach to high-bandwidth streaming centers around the techniques depicted in Figure 2. First, we split the target data stream into segments which are further subdivided into individual (typically packet-sized) objects. Depending on the requirements of the target applications, objects may be encoded [Goyal 2001; Luby et al. 1997] to make data recovery more efficient. Next, we purposefully disseminate disjoint objects to different clients at a rate determined by the available bandwidth to each client. We use a congestion-friendly 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 2, node D only has sufficient bandwidth to receive three 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 three objects per time unit to six 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 summaries 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.

Bullet requires an underlying overlay tree for the protocol that delivers 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 [Floyd et al. 2000]. 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.

3.2 Bullet′

An evolution from Bullet, Bullet′ builds upon the overlay mesh data dissemination approach advocated by Bullet, and considers the problem of large file dissemination to a large group of Internet users. One difference from Bullet is in the hybrid push/pull (Section 2.1) architecture; we enforce that only the source pushes and the receivers pull. The main reason for this choice is the desire to

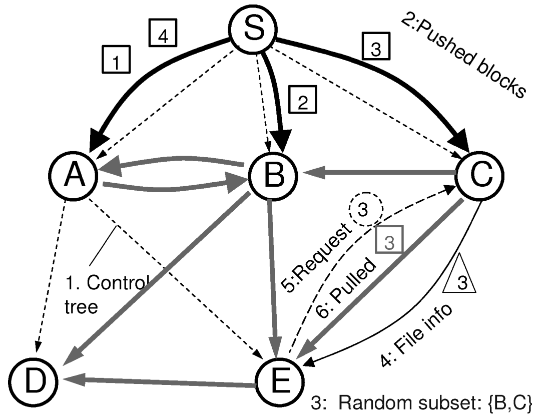

Fig. 3. Bullet' architectural overview. Gray lines represent a subset of peering relationships that carry explicitly pulled blocks.

minimize duplicate data transmission from the source. Further, to better address the challenges presented by the Internet where systems based on magic constants break down in unforeseen conditions, Bullet′ employs adaptive strategies to self-tune to dynamically changing network conditions and wide range of deployment settings. In addition, we minimize the number of user-adjustable parameters because in many cases it is impossible to find system-wide parameter settings that perform well across a range of conditions.

Figure 3 depicts the architectural overview of Bullet′. As in Bullet, we use a randomly constructed overlay tree for joining the system and for transmitting control information (shown in thin dashed lines, as step 1). To enforce diversity and to avoid wasting the source's outgoing bandwidth on duplicate data, however, the source pushes the file blocks to children in the control tree in a round-robin fashion, according to the available bandwidth to each child (step 2). We use a scalable, decentralized protocol to distribute changing, uniformly random subsets of system-wide file summaries over the control tree. Using this protocol, nodes advertise their identity and the block availability. Receivers use this information (step 3) to choose a set of senders to peer with, receive their file information (step 4), request (step 5), and subsequently receive (step 6) file blocks, effectively establishing an overlay mesh on top of the underlying control tree. Moreover, receivers make local decisions on the number of senders as well as the amount of outstanding data, adjusting the overlay mesh over time to conform to the characteristics of the underlying network. In departure from Bullet, Bullet′ request strategy is receiver-driven. Finally, during the file transfer, senders keep their receivers updated with the description of their newly received file blocks (step 4).

To deal with failures of control-tree children before they have a chance to transmit blocks to other nodes in the overlay, the source becomes a regular peer and switches to pull mode after sending out the file exactly once. This enables receivers to peer with the source as with any other peer and retrieve missing blocks.

3.3 System Components

3.3.1 *RanSub.* To address the challenge of locating disjoint content within the system, we use RanSub [Kostić et al. 2003a], 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 Banerjee et al. [2002], hua Chu et al. [2000], Jannotti et al. [2000], Kostić et al. [2003a], Rodriguez et al. [2004b], and Rowstron et al. [2001].

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

RanSub offers several ways of constructing distribute sets: (i) RanSub-ordered, (ii) RanSub-nondescendants, and (iii) RanSub-all. For Bullet, 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

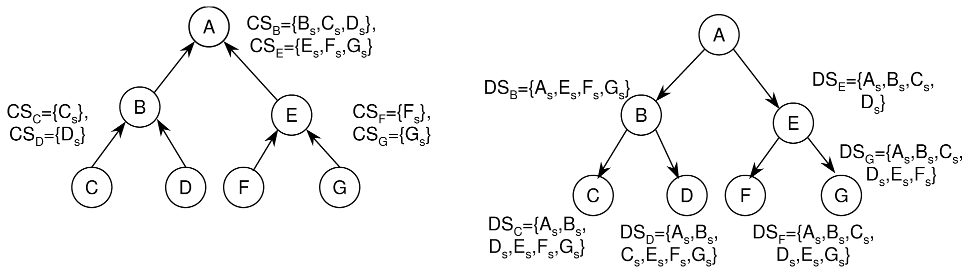

Fig. 4. An example of collect (on the left) and distribute (on the right) RanSub phases.

example of RanSub's collect-distribute process in Figure 4. 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. In the figure, A_S stands for node A's state.

For Bullet′, we choose the *RanSub-all* option. Here, each node receives a random subset consisting of all system nodes including its descendants.

3.3.2 *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 Byers et al. [2002] 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* [Broder 1997], 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.

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 [Bloom 1970], a bit array of size m with k independent associated hash functions. 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. To control the fraction of false positives, we carefully manage the number of entries in the Bloom filter while taking into account the size of filter and the number of hash functions.

In our Bullet′ evaluation, we transmitted unencoded files. In this case, the data objects' (blocks') sequence numbers do not come from a wide-range of encoded objects. Instead, they are simply numbered $1 \cdots n$, where n is the overall number of file blocks. We therefore switch to a different way to represent summaries in Bullet′, described in detail in Section 4.2.3.1.

4. IMPLEMENTATION

In this section we present the details of our systems, starting with the common framework for implementing them. Given that many of the Bullet′ mechanisms and strategies evolved from Bullet based on our experiences while using Bullet for file distribution, we intertwine the discussion of these two systems and organize it along the basic design principles described in Section 2. The topics described include the peering strategy, the request strategy, and the sending strategy. After describing the particulars of a Bullet mechanism, we follow up with a discussion before proceeding on to the Bullet′ equivalent.

4.1 Implementation Framework

We have implemented Bullet and Bullet′ using MACEDON. MACEDON [Rodriguez et al. 2004a] is a tool which makes the development of large scale distributed systems simpler by enabling 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.

4.1.1 *Layering.* MACEDON applications, services and libraries are layered. The full implementation of Bullet and Bullet′ consists of a generic file distribution application, the overlay network algorithms, and the algorithm for maintaining the random overlay tree. RanSub is embedded into the both Bullet and Bullet′ MACEDON protocol implementations. The overlay algorithms use transport protocols via transport abstractions provided by MACEDON that appear at the bottom of this hierarchy. The layering is not without shortcomings. The overlays cannot, for example, control the TCP or UDP socket buffer size. Instead, the overlay has control over the number of empty slots *in front of* of the socket buffers. In our implementation, the only data objects at Bullet's disposal were in its own cache. Even when transmitting a file using Bullet, we did not provide Bullet access to the file data stored on local disk. We refer to this kind of recovery as *main line* reconciliation.

4.1.2 *Streaming Application.* We have implemented a simple streaming application capable of streaming data over any specified tree or overlay multicast topology. In our implementation, we are able to stream data through overlay trees using UDP, TFRC [Floyd et al. 2000], or TCP. We used TFRC for the streaming experiments presented in this article.

4.1.3 *Download Application.* To evaluate various file distribution mechanisms, we implemented a generic file download application that can operate in either encoded or unencoded mode. In the encoded mode, it operates by generating continually increasing block numbers and transmitting them using the `macedon_multicast` API call on the overlay algorithm. In unencoded mode, it operates by transmitting the blocks of the file once using `macedon_multicast`. This makes it possible for us to compare download performance over the set of overlays specified in MACEDON. In both encoded and unencoded cases, the download application also supports a request function from the overlay network to provide the block of a given sequence number, if available, for transmission. The download application also supports the reconstruction of the file from the data blocks delivered to it by the overlay. The download application takes as parameters block size and file name. We use standard TCP for file download experiments in this article.

4.2 Peering Strategy

4.2.1 *Bullet.* We first describe Bullet and then the changes in Bullet′ for a number of peering strategy aspects.

4.2.1.1 *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-byte) summary of the data that each node contains. RanSub delivers subsets of these summary tickets to nodes every configurable *epoch* (5 s 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. To allow clean up, Bloom filter entries are integers. Hence, inserting an element involves incrementing the appropriate entries, while removal amounts to decrementing of the required entries. For network transmission, the “counting” Bloom filters are reduced to the “classic” form of 1 bit/entry to reduce control overhead.

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 nonfragmented 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 (the number of identical entries in the two summary tickets divided by the total number of summary ticket entries) 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 4.2.1.2). 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 4.2.1.2).

4.2.1.2 *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 (nonduplicate) 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 Kostić et al. [2003b].

4.2.2 *Discussion*

4.2.2.1 *Improving the Startup Time.* We observed that nodes in the Bullet mesh were taking considerable time to ramp-up to a high download rate after starting up. The long startup time is also evident from Figure 12. We attributed this behavior to the Bullet's peering strategy that grows the peer set by one node a time, with each delivery of the RanSub distribute set. Given that a node requires several sending peers (10 in the Bullet baseline code) for high bandwidth, and the 5-s RanSub epoch length, the startup time could be close to 1 min. Therefore, we decided to allow a node to try establishing peering relationships with as many senders as its current limitation on the sender set size allows. This change resulted in a considerable reduction in the file download time, especially for smaller files when the startup phase represents a considerable portion of the overall download time. With the new startup strategy, a node ramps up to several Mb/s download bandwidth in 5–10 s even in 250-node overlays.

4.2.2.2 *Managing Peer Sets.* For nodes to fill their pipes with useful data, it is imperative that they locate and maintain a set of peers that can provide them with good service. In the face of bandwidth changes and unstable network conditions, we determined that a fixed number of peers is suboptimal (see Section 5.10). Further, we noticed that Bullet's policy of closing the slowest sender each epoch might unnecessarily discard a sender that is only marginally slower than the rest of the sending peer set. These findings pointed to a revision of the mechanisms for improving the overlay mesh.

4.2.3 *Bullet′*

4.2.3.1 *Finding Overlay Peers.* Bullet′ nodes find peers in a way that is similar to Bullet. Here, we use *RanSub-all* to periodically delivers uniformly random subsets of *all* nodes to each participant in the overlay. Note that we depart from the non-descendants option used by Bullet. Since only the source's children receive pushed data and all other nodes pull file blocks, there is no need to instruct RanSub to avoid the descendant nodes when there is no tree-based data dissemination systemwide. We also use 10 summaries in Bullet′; the only difference is in the type of summaries. When transmitting an unencoded file, the summary can be an exact bitmap of the file blocks if it fits into the given space (120 bytes). Otherwise, it can either be a histogram-type summary with the number of blocks locally available in each 256-bit range (each block count can then be represented with 8 bits within the summary), or a nonzero bitmap of the last portion of the file recovered thus far that fits into the allotted size. In our evaluation, we use the latter option if the bitmap does not fit in its entirety.

4.2.3.2 *Improving the Bullet′ Mesh.* The peering mechanism used in Bullet′ evolved based on our experience with Bullet. Bullet′ must discard peers whose service degrades, and it must also adaptively decide how many peers it should be downloading from/sending to. Note that peering relationships are not inherently bidirectional; two nodes wishing to receive data from each other must establish peering links separately. Here we use the terms *sender* to refer to a peer a node is receiving data from and *receiver* to refer to a peer a node is sending data to.

Each node maintains two variables, namely `MAX_SENDERS` and `MAX_RECEIVERS`, which specify how many senders and receivers the node wishes to have at maximum. Initially, these values are set to 10, the value we have experimentally chosen for the released version of Bullet. Bullet′ also imposes hard limits of 6 and 25 for the number of minimum/maximum senders and receivers. Each time a node receives a RanSub distribute message containing a random subset of peers and the summaries of their file content, it makes an evaluation about its current peer sets and decides whether it should add or remove both

```
void ManageSenders() {

  if (size(senders) != MAX_SENDERS)
    return;
  if (num_prev_senders == 0) {
    // try to add a new peer by default
    MAX_SENDERS++;
  }
  else if(size(senders) > num_prev_senders) {
    if(incoming_bw > prev_incoming_bw)
      // bandwidth went up, try adding
      // a sender
      MAX_SENDERS++;
    else
      // adding a new sender was bad
      MAX_SENDERS--;
  }
  else if (size(senders) < num_prev_senders) {
    if(incoming_bw > prev_incoming_bw)
      // losing a sender made us faster,
      // try losing another
      MAX_SENDERS--;
    else
      // losing a sender was bad
      MAX_SENDERS++;
  }
}
```

Fig. 5. ManageSenders() pseudocode.

senders and receivers. If the node has its current maximum number of senders, it makes a decision as to whether it should either "try out" a new connection or close a current connection based on the number of senders and bandwidth received when the last distribute message arrived. A similar mechanism is used to handle adding/removing receivers, except in this case Bullet′ uses outgoing bandwidth instead of incoming bandwidth. Figure 5 shows the pseudocode for managing senders.

Every 5 s, Bullet′ calculates the average and standard deviation of bandwidth received from all of its senders. It then sorts the senders in order of least bandwidth received to most, and disconnects itself from any sender who is more than 1.5 standard deviations away from the mean, so long as it does not drop below the minimum number of connections (6). This way, Bullet′ is able to keep only the peers who are the most useful to it. A fixed minimum bandwidth was not used so as to not hamper nodes who are legitimately slow. In addition, the slowest sender is not always closed since if all of a peer's senders are approximately equal in terms of bandwidth provided, then none of them should be closed.

A nearly identical procedure is executed to remove receivers who are potentially limiting the outgoing bandwidth of a node. However, Bullet′ takes care to sort receivers based on the ratio of their bandwidth they are receiving from a particular sender to their total incoming bandwidth. This is important because

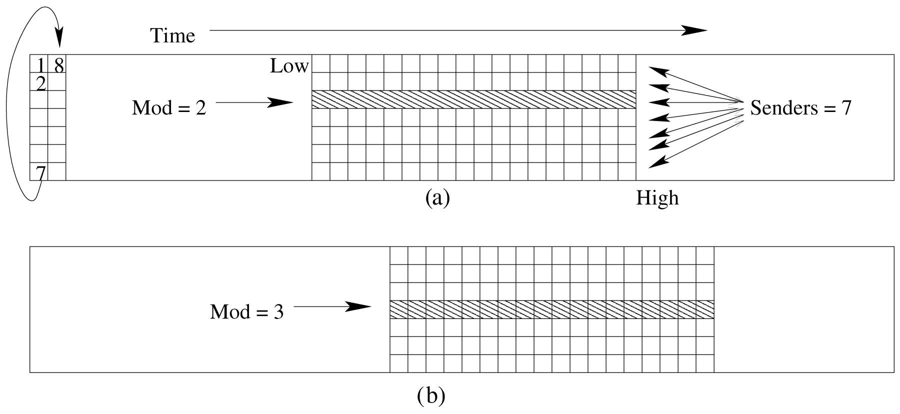

Fig. 6. A Bullet receiver's view of data.

we do not want to close peers who are getting a large fraction of their bandwidth from a given sender. We chose the value of 1.5 standard deviations because 1 would lead to too many nodes being closed whereas 2 would only permit a very few peers to ever be closed.

4.3 Request Strategy

4.3.1 *Bullet.* This section describes how Bullet receivers orchestrate data transfers from the chosen peers. Recall that the peering relationship is initiated by the receiver, and it involves sending the digest (a Bloom filter) of currently available data objects to the potential sender. Therefore, the Bullet request strategy is implicit, and it empowers the senders to decide which data objects will be sent to the receivers.

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 6, 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 s 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 6(b)). In addition, the receiving node assigns to each sender a row from the matrix, labeled *mod*. A sender will forward packets to 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. For streaming cases when the streaming rate is less than the node access link capacity, this wasteful reception of duplicate packets is tolerable; less than 10% of all received packets are duplicates in our experiments. Section 4.3.2.1 describes our experiences with this request strategy for file distribution.

4.3.2 *Discussion: Limitations of the Bullet Request Strategy for File Distribution.* This section describes our efforts to overcome the limitations of the Bullet request strategy while trying to distribute a file as quickly as possible with Bullet in the file distribution role.

4.3.2.1 *Duplicate Data Caused by the Request Strategy.* The first problem we examined was duplicate data. Specifically, every epoch (5 s by default), receivers were transmitting the digests of the locally available data to senders, along with a total count of senders and a sender-specific modulo index. Senders would compute a list of missing blocks for that particular digest, randomize the list, and then try transmitting those blocks in the nonblocking fashion to the requesting receiver. Modulo indices were rotated each epoch, which meant that a receiver could implicitly rerequest a block from another sender although that particular block was either "in-flight," or enqueued in the TCP socket buffer, or in the MACEDON TCP transport queue positioned directly in front of the TCP socket buffer. We substantially reduced duplicate block overhead by shrinking the MACEDON TCP transport queues to only one block per receiving peer. Nevertheless, we could not completely eliminate the duplicates and we were forced to keep relatively small block sizes that were not effectively neutralizing MACEDON header overhead. Experiments have shown that duplicate data amounted to between 5 and 10% of the overall data received. Since we were striving for the fastest file distribution mechanism possible, this performance loss was not acceptable. In addition, the Bullet request strategy incurs a potentially high latency to retrieve a particular block, which exacerbates the "last-block" problem of unencoded file distribution systems. Subsequently, we decided that we needed an improved request strategy.

4.3.2.2 *Imperfections in the Recovery Strategy.* In parallel with our efforts on reducing duplicate data reception, we identified several imperfections in the recovery strategy. Some of the imperfections were due to ťo the use of Bloom filters as digests and their inherent false positive rate, even though it was fixed at low 1%. The rest of the problems were due to "main-line" reconciliation, where the download application did not have a chance to send any data it has already received to its peers: reconciliation was limited to the buffer size within the Bullet code.

To cope with the imperfections in the recovery strategy we implemented the "rateless" erasure codes [Maymounkov and Mazieres 2003]. Summarized experiences that are generally applicable are described in Section 2.2; here we outline some implementation peculiarities.

4.3.2.3 *Erasure Encoding.* Once we started experimenting with encoded files, we began realizing the limitations of encoding from the system perspective. If the encoded blocks cannot fit in physical memory, the performance suffers as the receiver is forced to swap the blocks from the disk during decoding when blocks are accessed at random. This design constraint forced us to send the file in multiple segments, requiring resolution of some important issues. For example, we had to deal with "main-line" reconciliation (aiding the multicast mechanism provided by Bullet) and incorporate perpendicular downloads of segments from the peers that have already reconstructed those segments. We were achieving good performance, but the receivers behind slower access links had trouble keeping up. Subsequently, we wanted to reduce the protocol overhead to explore the limits of this dissemination model.

4.3.2.4 *Reducing Control Overhead.* In an effort to increase the throughput of the main line reconciliation mechanism, we decided to streamline it by reducing control overhead. For example, a Bullet receiver is sending the same bloom filter to each of the 10 senders along with the modulo number stating what portion of the filter was useful to that peer. Maintaining smaller bloom filters for each modulo number resulted in factor of 7 control overhead reduction (Bloom filters that were 10 times smaller were not able to capture the same amount of information as a single Bloom filter). The overall performance improvement depended on the speed of the node's access links, but it was no more than 10%.

4.3.3 *Bullet′.* Armed with the Bullet performance results and difficulties with its request strategy (Section 4.3.2.1), we decided to make the Bullet′ request strategy receiver driven. Here, we discuss how requests are ordered in Bullet′, how nodes choose the amount of outstanding data (flow control), and how senders keep receivers up-to-date.

4.3.3.1 *Ordering Requests.* We considered using four different strategies for ordering requests when designing Bullet′. All of the strategies are for making local decisions on either the unencoded or source-encoded file.

Given a per-peer list representing blocks that are available from that peer, the following are possible ways to order requests:

—*First encountered*. This strategy will simply arrange the lists based on block availability. That is, blocks that are just discovered by a node will be requested after blocks the node has known about for a while. As an example, this might correspond to all nodes proceeding in lockstep in terms of download progress. The resulting low block diversity this causes in the system could lead to lower performance.

—*Random*. This method will randomly order each list with the intent of improving the block diversity in the system. However, there is a possibility of requesting a block that many other nodes already have which does *not* help block diversity. As a result, this strategy might not significantly improve the overall system performance.

—*Rarest*. The *rarest* technique is the first that looks at block distributions among a node's peers when ordering the lists. Each list will be ordered with the least represented blocks appearing first. This strategy has no method for breaking ties in terms of rarity, so it is possible that blocks quickly go from being under represented to well represented when a set of nodes makes the same deterministic choice.

—*Rarest random*. The final strategy we describe is an improvement over the *rarest* approach in that it will choose uniformly at random from the blocks that have the highest rarity. This strategy eliminates the problem of deterministic choices leading to suboptimal conditions.

In order to decide which strategy worked the best, we implemented all four in Bullet′. Most of the time, rarest random outperformed other schemes. We present our findings in Section 5.9.

4.3.3.2 *Flow Control.* Although the *rarest random* request strategy enables Bullet′ to request blocks from peers in a way that encourages block diversity, it does not specify how many blocks a node should request from its peers at once. This choice presents a tradeoff between control overhead (making requests) and adaptivity. On one hand, a node could request one block at a time, not requesting another one until the first arrived. Although stopping and waiting would provide maximum insulation to changing network conditions, it would also leave pipes underutilized due to the round trip time involved in making the next request. At the other end of the spectrum is a strategy where a node would request everything that it knew about from its peers as soon as it learned about it. In this case, the control traffic is reduced and the node's pipe from each of its peers has a better chance of being full but this technique has major flaws when network conditions change. If a peer suddenly slows down, the node will find itself stuck waiting for a large number of blocks to come in at a slow rate. We have experimented with canceling of blocks that arrive "slowly," and found that in many cases these blocks are "in-flight" or in the sender's socket buffer, making it difficult to effectively stop their retrieval without closing the TCP connection.

As seen in Section 5.11, using a fixed number of outstanding blocks will not perform well under a wide variety of conditions. To remedy this situation, Bullet′ employs a novel flow control algorithm that attempts to dynamically change the maximum number of blocks a node is willing to have outstanding from each of its peers. Our control algorithm is similar to XCP's [Katabi et al. 2002] efficiency controller, the feedback control loop for calculating the aggregate feedback for all the flows traversing a link. XCP measures the difference in the rates of incoming and outgoing traffic on a link, and computes the total number of bytes by which the flows' congestion windows should increase or decrease. XCP's goal is to maintain zero packets queued on the bottleneck link. For the particular values of control parameters $\alpha = 0.4$, $\beta = 0.226$, the control loop is stable for any link bandwidth and delay [Katabi et al. 2002].

```
void ManageOutstanding (sender, block) {
// start with current value
sender->desired = sender->requested + 1;

if (block->wasted <= 0 || block->in_front <= 1)
    sender->desired -=
      0.4*block->wasted*sender->bandwidth/block_size);

if (block->wasted <= 0 && block->in_front > 1)
    sender->desired -= 0.226*(block->in_front - 1);
}
```

Fig. 7. Pseudocode for setting the maximum number of per-peer outstanding blocks.

We start with the original XCP formula and adapt it. Since we want to keep each pipe full while not risking waiting for too much data in case the TCP connection slows down, our goal is to maintain exactly one block in front of the TCP socket buffer, for each peer. With each block it sends, sender measures and reports two values to the receiver that runs the algorithm depicted in Figure 7 in the pseudocode. The first value is `in_front`, corresponding to the number of queued blocks in front of the socket buffer when the request for the particular block arrives. The second value is `wasted`, and it can be either positive or negative. If it is negative, it corresponds to the time that is "wasted" and could have been occupied by sending blocks. If it is positive, it represents the "service" time this block has spent waiting in the queue. Since this time includes the time to service each of the `in_front` blocks, we take care not to double count the service time in this case. To convert the wasted (service) time into units applicable to the formula, we multiply it by the bandwidth measured at the receiver, and divide by block size to derive the additional (reduced) number of blocks receiver could have requested. Once we decide to change the number of outstanding blocks, we mark a block request and do not make any adjustments until that block arrives. This technique allows us to observe any changes caused by our control algorithm before taking any additional action. Further, just matching the rate at which the blocks are requested with the sending bandwidth in an XCP manner would not saturate the TCP connection. Therefore, we take the `ceiling` of the noninteger value for the desired number of outstanding blocks whenever we increase this value.

Although Bullet′ knows how much data it should request and from whom, a mechanism is still needed that specifies when the requests should be made. Initially, the number of blocks outstanding for all peers starts at 3, so when a node gains a sender it will request up to three blocks from the new peer. Conceptually, this corresponds to the pipeline of one block arriving at the receiver, with one more in-flight, and the request for the third block reaching the sender. Whenever a block is received, the node reevaluates the potential from this peer and requests up to the new maximum outstanding.

4.3.3.3 *Staying Up-To-Date.* One small detail we have deferred until now is how nodes become aware of what their peers have. Bullet′ nodes use a simple bitmap structure to transmit *diffs* to their peers. These diffs are incremental, such that a node will only hear about a particular block from a peer once. This approach helps to minimize wasted bandwidth and decouples the size of the diff from the size of the file being distributed. Currently, a diff may be transmitted from node `A` to `B` in one of two circumstances—either because `B` has nothing requested of `A`, or because `B` specifically asked for a diff to be sent. The latter would occur when `B` is about to finish requesting all of the blocks `A` currently has. An interesting effect of this mechanism is that diff sending is automatically self clocking; there are no fixed timers or intervals where diffs are sent at a specific rate. Bullet′ automatically adjusts to the data consumption rates of each individual peer.

4.4 Sending Strategy

4.4.1 *Bullet.* Overall, the dissemination over the Bullet mesh is a combination of a system-wide push over the underlined tree, and pull over the peering links. In this section we present the details of the sending strategy that is used to push the data. The overarching goal of this strategy is to maximize the bandwidth achieved by each of the receivers in a node's subtree by increasing 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 nondescendant 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 nondescendants, 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 nonblocking; successful transmissions are send attempts that are accepted by the nonblocking 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 8.

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 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

```
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);
  }
}
```

Fig. 8. Pseudocode for Bullet's disjoint data send routine.

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.

4.4.2 *Discussion: Duplicate Data.* In this section we describe our efforts in minimizing duplicate data transmission at various points in the overlay.

4.4.2.1 *Potentially Duplicate Data from the Parent in the Underlying Tree.* We believed there was too much overhead created by Bullet's sending strategy. To eliminate potentially duplicate data from the parent in the underlying tree during file distribution, we closely examined the Bullet dissemination model that consists of the low-latency, push-based dissemination over the underlying tree, and the pull-based dissemination over the peering links that form the mesh. Pushing data over a tree requires coordination between each child and a parent to make sure that duplicates were minimized (otherwise, a node might receive a pushed block that it is currently retrieving from a peer). This coordination requires transmission of control messages and digests that form pure control overhead. In addition, we argued that a file distribution mechanism did not have to be optimized for the delivery latency of each block, thereby diminishing the need for the low latency provided by the tree dissemination. Taking into consideration these and other issues (see Section 2.1), we decided to stop the push-based data from being disseminated further from the source's children in the underlying tree.

4.4.2.2 *Duplicate Data Sent by the Source.* Each Bullet parent tries to send blocks optimistically to probe how much "extra data" it can send to each of its children, where "extra data" refers to duplicate blocks that are sent to more than one child (Section 4.4.1). We realized that we could not achieve near-optimal performance if we allowed the source to send extraneous data over its outbound access link. Therefore, we switched to a source strategy that sends blocks only once, in a round-robin fashion and according to available bandwidth, among the source's children.

4.4.3 *Bullet′.* As mentioned previously, Bullet′ uses a hybrid push/pull approach for data distribution where the source behaves differently from everyone else. This decision stems from the desire to minimize the overall file distribution time. Specifically, the source takes a rather simple approach: it sends a block to each of its RanSub children iteratively until the entire file has been sent. If a block cannot be sent to one child (because the pipe to it is already full), the source will try the next child in a round robin fashion until a suitable recipient is found. In this manner, the source never wastes bandwidth forcing a block on a node that is not ready to accept it. Once the source makes each of the file blocks available to the system, it will advertise itself in RanSub so that arbitrary nodes can benefit from having a peer with all of the file blocks. This mechanism takes care of the case when a source's child dies after receiving a block from the source, and before it manages to transmit it to any other node.

From the perspective of nonsource nodes, determining the order in which to send requested blocks is equally as simple. Since Bullet′ dynamically determines the number of outstanding requests, nodes should always have approximately one outstanding request at the application level on any peer at any one time. As a result, the sender can simply serve requests in FIFO order since there is not much of a choice to make among such few blocks. Note that this approach would not be optimal for all systems, but since Bullet′ dynamically adjusts the number of requests to have outstanding for each peer, it works well.

5. EVALUATION

In this section, we present the evaluation of Bullet and Bullet′ as well as the competing systems. Except BitTorrent, all of the evaluated systems are implemented in the MACEDON common development infrastructure. As a result, we believe that our comparisons qualitatively show algorithmic differences rather than implementation intricacies.

We have evaluated the performance of Bullet and Bullet′ in real Internet environments as well as the ModelNet [Vahdat et al. 2002] IP emulation framework. While the bulk of our experiments used ModelNet, we also report on our experience with Bullet′ on the PlanetLab Internet testbed [Peterson et al. 2002].

Our evaluation has two major parts. First, we evaluate the streaming performance of Bullet versus streaming over a bandwidth-optimized overlay tree. 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 5.3. We also present an evaluation of Bullet versus epidemic algorithms, and demonstrate Bullet's performance under failure.

In the second part of our evaluation, we use Bullet for file distribution. We also show Bullet′ performance results, and compare both of our systems with SplitStream [Castro et al. 2003], and BitTorrent [Cohen 2003]. We use a parametrized version of Bullet′ to illustrate the benefits of our adaptive mechanisms for peering strategy and flow control.

5.1 Summary

In this section we provide a summary of the key features in Bullet, Bullet′, and state-of-the-art systems for high-bandwidth overlay data dissemination. We created Bullet with the goal of maximizing the bandwidth delivered to each recipient. After demonstrating its superior performance on a large scale relative to overlay trees and epidemic approaches, we turned our attention to the large file distribution problem. We started with Bullet and realized its shortcomings in this role. Subsequently, we performed a detailed analysis of the design space, incorporated the lessons from our extensive performance evaluations and a number of systems developed contemporaneously to our own, creating Bullet′. We use Table I to highlight the major features of these systems. We believe that

Table I. Summary of Key Properties of Bullet, Bullet′, and State-of-the-Art High-Bandwidth Overlay Data Dissemination Systems

System property	Overlay trees	SplitStream	BitTorrent	Bullet	Bullet′
Push or pull	Push	Push	Pull	Hybrid	Hybrid
Encoded or unencoded	Either	Encoded	Unencoded	Either (requires encoding for file distribution)	Unencoded
Peering selection	None (all bandwidth comes from the parent)	Pastry/Scribe selected plus spare bandwidth tree traversal	Centralized (Tracker)	Scalable, decentralized, informed	Scalable, decentral-ized, informed
Peer set size	Fixed	Fixed	Fixed	Fixed	Dynamic
Topology adaptation	Yes	No	Implicit (tit-for-tat), prefers nodes with better bandwidth	Yes	Yes
Ordering of requests	None (push-based)	None (push-based)	Rarest random	Implicit (sender picks at random)	Rarest random (same as BitTorrent)
Flow control of requests	Implicit (Du-plicates created)	None (push-based)	Fixed	none	Dynamic sizing
Source sending strategy	Attempts to send same data to each child	Disjoint	Source peers pull data, duplicates possible	According to available bandwidth, with duplicates	According to available bandwidth, no duplicates
Nonsource sending strategy	Attempts to send the same data to each child	Attempts to send the same data to each child	FIFO for receiver requests	Push ac. to avail. bw, no duplicates; random selection from digests for pulled data	FIFO for receiver requests, targets 1 request to each receiver
Fairness	None	bandwidth-contribution based	Tit-for-tat (TFT)	None	None

the choices we made in Bullet′ can be viewed as preferred choices when building a single-source large file distribution mechanism that does not use encoding and does not enforce fairness among a set of receivers. We now discuss in more detail each of the dimensions presented in Table I.

—*Push or pull*. Our evaluations show that Bullet′'s hybrid push/pull data dissemination mechanism is a good choice for high-bandwidth data dissemination. In particular, the choice of performing push from the source is superior to BitTorrent's strategy of having receivers pull data from the source because Bullet′ completely avoids duplicates. The per-object delivery latency in SplitStream is likely to be superior to Bullet′ in cases when the network links are not being heavily utilized or under low congestion. On the other hand, when the socket buffer queues of a congestion-friendly transport protocol start filling up due to packet losses in the network, SplitStream's data object delivery latencies might start approaching Bullet's per-object latencies. We discuss general merits of push and pull in Section 2.1.

—*Peering selection.* A system that scales with the large number of receivers must include a scalable peer selection mechanism. Bullet′ accomplishes this by using RanSub, while SplitStream uses Pastry, a scalable p2p routing substrate, and Scribe, a mechanism for constructing overlay trees on top of Pastry. BitTorent, on the other hand, relies on a centralized tracker, which, in addition to being unscalable, represents a central point of failure.

—*Peer set size.* We believe that Bullet′'s ability to dynamically set its peer set size is important for filling each receiver's pipe under dynamic network conditions. We quantify this claim using a parametrized version of Bullet′ in Section 5.10.

—*Topology adaptation.* To match the conditions of the underlying network, an overlay data dissemination mechanism must be agile in adapting its topology. We back this claim in Section 5.8 by demonstrating superior Bullet′ performance relative to state-of-the-art systems.

—*Ordering of requests.* Although it is a strategy put into action via local, per-node decisions, request ordering is vital in ensuring diversity of data in the system so that receivers can help one another. Our rarest random policy for ordering of requests is similar to that employed by BitTorrent. The evaluation in Section 5.9 quantifies the claims made in Section 2.4.

—*Flow control of requests.* Controlling the number of outstanding requests (as is the case in Bullet′) over peering links with varying available bandwidth and latency is vital for good performance. Having motivated the problem in Section 2.4, we corroborate this claim in Section 5.11.

—*Sending strategies.* Bullet′'s superior performance (Section 5.8) is in part due to our careful choices to avoid transmission of duplicates. This is particularly important when bandwidth is scarce. We discuss the possible approaches in Section 2.5 and our work on reducing duplicates in Bullet in Section 4.4.2.

—*Fairness.* We made a decision to pursue a performance-first approach (Section 2.6) with Bullet′. We acknowledge that a system that enforces fairness, for example by using BitTorrent's tit-for-tat mechanism, might be more appropriate in certain scenarios. Adding this functionality to Bullet′ would be straightforward.

5.2 Experimental Setup

5.2.1 *Streaming.* Our streaming ModelNet experiments made use of 50 2-GHz Pentium 4's running Linux 2.4.20 and interconnected with 100-Mb/s and 1-Gb/s Ethernet switches. For the majority of these experiments, we

Table II. Bandwidth Ranges for Link Types Used in Our Topologies Expressed in kilobits/s

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	5000–10,000
High bandwidth	1600–5600	2000–8000	2000–8000	10,000–20,000

multiplexed 1000 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 1.4-GHz Pentium III's running FreeBSD-4.7 as emulators. This platform supports approximately 2-3 Gb/s of aggregate simultaneous communication among end hosts. For most of our ModelNet experiments, we use 20,000-node INET-generated topologies [Chang et al. 2002]. We randomly assign our participant nodes to act as clients connected to 1° 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 Calvert et al. [1997], 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 kb/s as specified in Table II. 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 "digital-fountain"-type erasure Codes, for example, when distributing a file.

5.2.2 *File Distribution.* Our file distribution ModelNet experiments made use of 25 2.0- and 2.8-Ghz Pentium 4s running Xeno-Linux 2.4.27 and interconnected by 100-Mb/s and 1-Gb/s Ethernet switches. In the experiments presented here, we multiplexed 100 logical end nodes running our download applications across the 25 Linux nodes (four per machine). ModelNet routes packets from the end nodes through an emulator responsible for accurately emulating the hop-by-hop delay, bandwidth, and congestion of a given network topology; a 1.4-GHz Pentium III running FreeBSD-4.7 served as the emulator for these experiments.

All of our experiments were run on a fully interconnected mesh topology, where each pair of overlay participants were directly connected. While admittedly not representative of actual Internet topologies, it allowed us maximum flexibility to affect the bandwidth and loss rate between any two peers. The inbound and outbound access links of each node are set to 6 Mb/s, while the nominal bandwidth on the core links is 2 Mb/s. In an attempt to model the wide-area environment (PingER Site-by-month History Table; go online to `http://www-iepm.slac.stanford.edu/pinger/tools/table.html`), we configured ModelNet to randomly drop packets on the core links with probability ranging from 0 to 3%. The loss rate on each link was chosen uniformly at random and fixed for the duration of an experiment. To approximate the latencies in the Internet [Dabek et al. 2004; PingER Site-by-month History Table; go online to `http://www-iepm.slac.stanford.edu/pinger/tools/table.html`], we set the propagation delay on the core links uniformly at random between 5 and 200 ms, while the access links have a 1-ms delay.

As described most of the following sections, we conducted identical experiments in two scenarios: a static-bandwidth case and a variable-bandwidth case. Our bandwidth-change scenario was designed to stress the peering strategies of the systems under evaluation; it models changes in the network bandwidth that correspond to correlated and cumulative decreases in bandwidth from a large set of sources from any vantage point. To effect these changes, we decreased the bandwidth in the core links with a period of 20 s. At the beginning of each period, we chose 50% of the overlay participants uniformly at random. For each participant selected, we then randomly chose 50% of the other overlay participants and decreased the bandwidth on the core links from those nodes to 50% of the current value, without affecting the links in the reverse direction. The changes we made are cumulative; that is, it was possible for an unlucky node pair to have 25% of the original bandwidth after two iterations. We did not alter the physical link loss rates that were chosen during topology generation.

5.2.3 *Large-Scale Bullet′ Performance Under Node Churn and Flashcrowds.* Our large-scale ModelNet [Vahdat et al. 2002] Bullet′ experiments under node churn and flash crowds scenarios made use of 27 dual 3.4-GHz Pentium 4 Xeons with 2 GB RAM running Linux 2.6.17 interconnected by a full-rate 1-Gb/s Ethernet switch. We multiplexed 250 logical end peers running our applications across the 27 Linux machines (slightly over four instances on average per processor); three 3.4-GHz Pentium 4 Xeons running FreeBSD 4.9 served as the emulator for these experiments.

We used a 5000-node INET [Chang et al. 2002] topology that we further annotated with bandwidth capacities for each link. The INET topology preserves the power law distribution of node degrees in the Internet. We kept the latencies generated by the topology generator; the average network RTT was 130 ms. We randomly assigned participants to act as clients connected to 1-degree stub nodes in the topology. We set transit-transit links links to be 150 Mb/s, while we set access links to 6-Mb/s inbound/outbound bandwidth. This setup provided an environment where the access links were constrained, and the network core was well provisioned. To emulate the effects of cross traffic, we instructed ModelNet to drop packets on client-stub (access) and transit-transit links at random with a probability chosen uniformly at random from [0, 0.003] and [0, 0.005], respectively, for each link separately.

5.3 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 would allow 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 5.7.

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 [Kim et al. 2002]. 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 [Cohen and Kaempfer 2001], even in the absence of link losses. For the purposes of this article, 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 [Padhye et al. 1998].
(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 $B : E \to R^+$, loss rate $L : E \to [0, 1]$, propagation delay $D : E \to R^+$ of each link, set of possible overlay links $O = \{(v, w) \mid v, w \in P, v \neq w\}$, and routing table $RT : O \times E \to \{0, 1\}$, 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

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

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 nonoptimal, but a similar approach was found to perform well [Cohen and Kaempfer 2001].

Our algorithm is similar to the Widest Path Heuristic (WPH) [Cohen and Kaempfer 2001], and more generally to Prim's MST algorithm [Prim 1957]. 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 reexamine 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.

5.4 Streaming over Bullet Versus Bandwidth-Optimized Tree

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

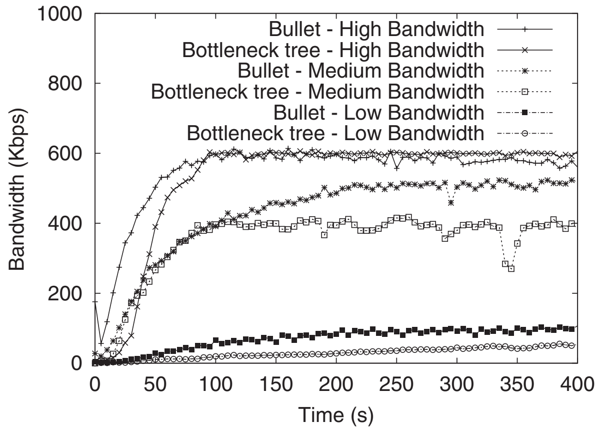

Fig. 9. Achieved bandwidth for Bullet and bottleneck tree over time for high-, medium-, and low-bandwidth topologies.

underlying topology. Table II describes representative bandwidth settings for our streaming rate of 600 kb/s. 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. The average Bullet per-node control overhead is approximately 30 kb/s. By tracing certain packets as they move through the system, we are able to acquire link stress (number of times the same data traverses a physical link) 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.

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 kb/s 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 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 [Jannotti et al. 2000]. The resulting dynamically constructed trees never achieved more than 75% of the bandwidth of our own offline algorithm.

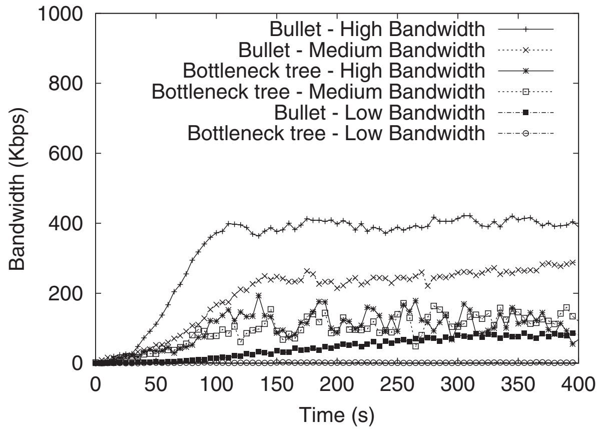

Fig. 10. Achieved bandwidths for Bullet and bottleneck bandwidth tree over a lossy network topology.

5.5 Streaming over Bullet Versus Bandwidth-Optimized Tree 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 nontransit 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%. Similarly to the approach in Padmanabhan et al. [2003a], 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 10 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.

5.6 Streaming over Bullet Versus Epidemic Approaches

In this section, we explore how Bullet compares to data dissemination approaches that use some form of epidemic routing [Demers et al. 1987]. We implemented a form of "gossiping," where a node forwards nonduplicate 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 [Eugster et al. 2001] (recently improved to incorporate retrieval of data objects [Eugster et al. 2003]).

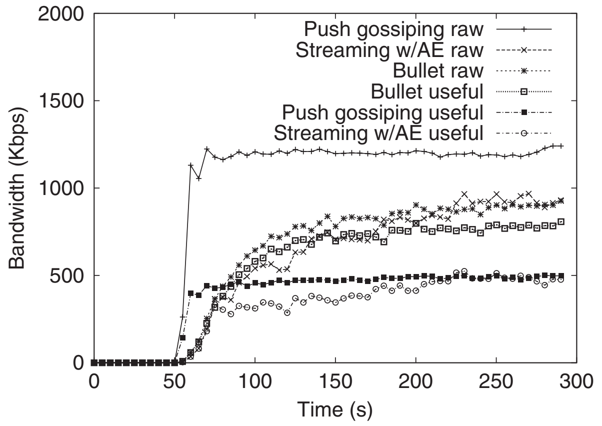

Fig. 11. Achieved bandwidth over time for Bullet and epidemic approaches.

We do not disseminate packets every T seconds; instead we forward them as soon as they arrive.

We also implemented a pbcast-like [Birman et al. 1999] 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 antientropy 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 antientropy recovery are able to maintain full group membership. While this might be difficult in practice, we assume that RanSub [Kostić et al. 2003a] could also be applied to these ideas, specifically in the case of antientropy 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 fewer peers in each round (five) than Bullet (10). For our configuration, we experimentally found that five 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 antientropy recovery to 20 s.

For these experiments, we used a 5000-node INET topology with no explicit physical link losses. We set link bandwidths according to the medium range from Table II, and randomly assigned 100 overlay participants. The randomly chosen root either streamed at 900 kb/s (over a random tree for Bullet and greedy bottleneck tree for antientropy recovery), or sent 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 came close to providing the target bandwidth to all participants, achieving approximately 60% more then gossiping and streaming with antientropy. 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 bandwidth 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.

5.7 Bullet Streaming 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 s 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. While the average achieved rate drops from 500 kb/s to 350 kb/s, 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 s 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 12, 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.

5.8 Overall File Download Performance

We begin by studying how Bullet and Bullet′ perform for file distribution, using the existing best-of-breed systems as comparison points. For reference,

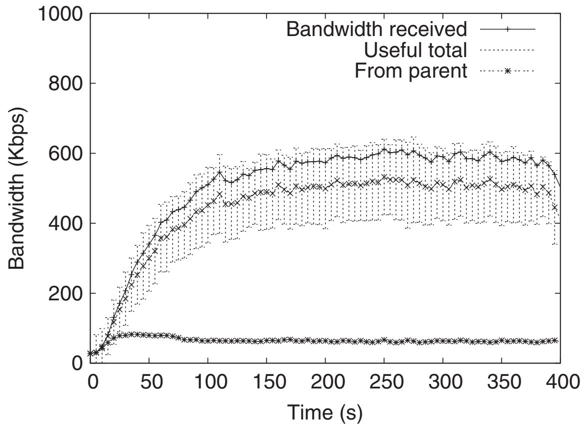

Fig. 12. Bullet's bandwidth over time with a worst-case node failure and RanSub recovery enabled.

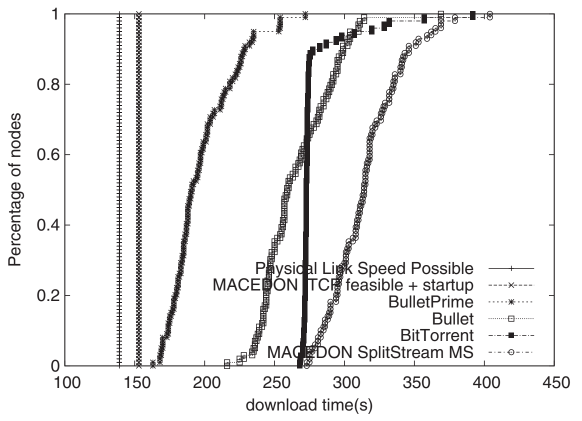

Fig. 13. Performance comparison for a 100-MB file download under random network packet losses.

we also calculate the best achievable performance given the overhead of our underlying transport protocols. In our experiments, all nodes start at t = 0. Figure 13 plots the results of downloading a 100-MB file on our ModelNet topology using a number of different systems. The graph plots the cumulative distribution function of node completion times for four experimental runs and two calculations. Starting at the left, we plot download times that are optimal with respect to access link bandwidth in the absence of any protocol overhead. We then estimate the best possible performance of a system built using MACEDON on top of TCP, accounting for the inherent delay required for nodes to achieve maximum download rate. The remaining four lines show the performance of Bullet′ running in the unencoded mode, Bullet, and BitTorrent, our MACEDON SplitStream implementation, in roughly that order. Bullet′ clearly outperforms all other schemes by approximately 25%. The slowest Bullet′ receiver finishes downloading 37% faster than for other systems.

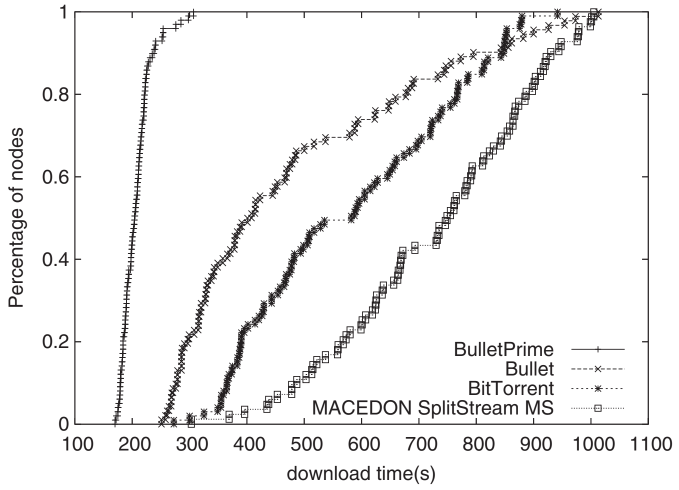

Fig. 14. Performance comparison for a 100-MB file download under synthetic bandwidth changes and random network packet losses.

The majority of BitTorrent nodes finish at an almost identical time because the source had difficulty sending data at a high rate in this topology; we attribute this to the BitTorrent's design of having a fixed number of receivers pulling data from the source with a constant number of outstanding blocks. Bullet′'s performance is even better in the dynamic scenario (faster by 32–70%), shown in Figure 14.

Sections 5.10 and 5.11 quantify the effects of Bullet′'s adaptive peer set sizing and flow control strategies, respectively, that contribute to Bullet′'s performance gains over BitTorrent. Here we explain the difference in performance of Bullet versus Bullet′. First, Bullet′'s source sending strategy does not send any block twice before sending the entire file exactly once. A Bullet source, on the other hand, sends some duplicate blocks deliberately, in an effort to determine the available bandwidth to each of its children in the underlying tree (as discussed in Section 4.4.2). Second, Bullet's request strategy causes additional duplicates to be sent (Section 4.3.2). Finally, the peering strategy uses a fixed number of peers, and always closes the slowest peer (Section 4.2.2).

We set the transfer block size to 16 kB in all of our experiments. This value corresponds to BitTorrent's subpiece size of 16 kB, and is also shared by the Bullet and SplitStream. For all of our experiments, we made sure that there was enough physical memory on the machines hosting the overlay participants to cache the entire file content in memory. Our goal was to concentrate on distributed algorithm performance and not worry about swapping file blocks to and from the disk. Bullet and SplitStream results were optimistic since we did not perform encoding and decoding of the file. Instead, we set the encoding overhead to 4%, assumed there would be enough physical memory to reconstruct the file without disk swapping during decoding, and declared the file complete when a node had received enough file blocks.

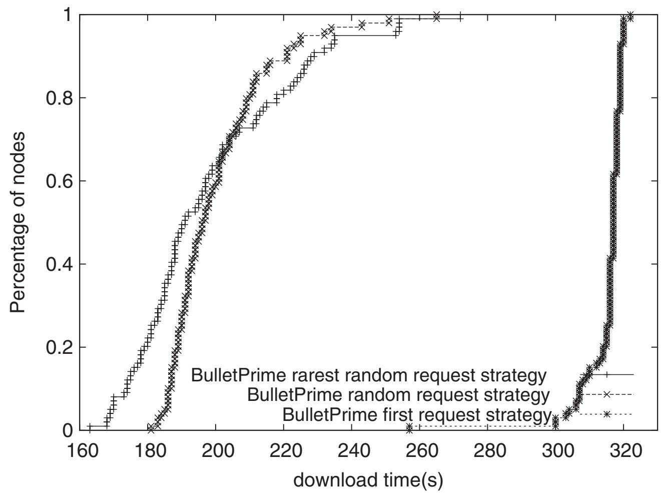

Fig. 15. Impact of request strategy on Bullet′ performance while downloading a 100-MB file under random network packet losses.

5.9 Request Strategy

Heartened by the performance of Bullet′ with respect to other systems, we now focus our attention on the various critical aspects of our design that we believe contribute to Bullet′'s superior performance. Figure 15 shows the performance of Bullet′ using three different peer request strategies, again using the CDF of node completion times. In this case each node is downloading a 100-MB file. We argue the goal of a request strategy is to promote block diversity in the system, allowing nodes to help each other. Not surprisingly, we see that the *first-encountered* request strategy performs the worst, while the rarest-random performs best among the strategies considered for 70% of the receivers. For the slowest nodes, the random strategy performs better. When a receiver is downloading from senders over lossy links, higher loss rates increase the latency of block availability messages due to TCP retransmissions and use of the congestion avoidance mechanism. Subsequently, choosing the next block to download uniformly at random does a better job of improving diversity than the rarest-random strategy that operates on potentially stale information.

5.10 Peer Selection

In this section we demonstrate the impossibility of choosing a single optimal number of senders and receivers for each peer in the system, arguing for a dynamic approach. In Figure 16 we contrast Bullet′'s performance with 10 and 14 peers (for both senders and receivers) while downloading a 100-MB file. The system configured with 14 peers outperforms the one with 10 because in a lossy topology like the one we are using, having more TCP flows, makes the node's incoming bandwidth more resilient to packet losses. Our dynamic approach is configured to start with 10 senders and receivers, but it closely tracks the

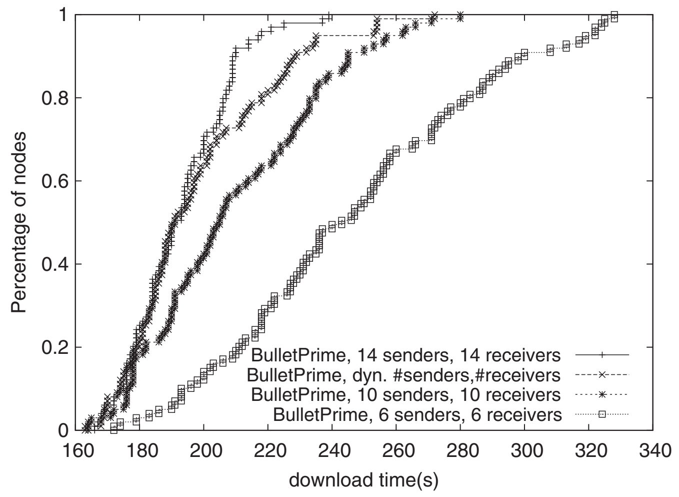

Fig. 16. Bullet′ performance under random packet losses for the static and the dynamic peer set sizing cases while downloading a 100-MB file.

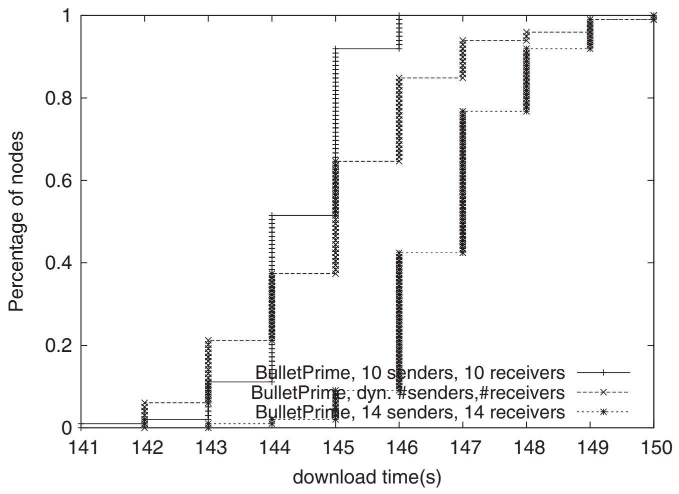

Fig. 17. Bullet′ performance without bandwidth changes and random packet losses for the static and the dynamic peer set size sizing cases while downloading a 10-MB file in a topology with constrained access links.

performance of the system with the number of peers fixed to 14 for 50% of receivers.

For our final peering example, we construct a 100 node topology with ample bandwidth in the core (10 Mb/s, 1-ms latency links) with 800-kb/s access links and without random network packet losses. Figure 17 shows that, unlike in the previous experiments, Bullet′ configured for 14 peers performs *worse* than in a setup with 10 peers. Having more peers in this constrained environment forces more maximizing TCP connections to compete for bandwidth. In addition, maintaining more peers requires sending more

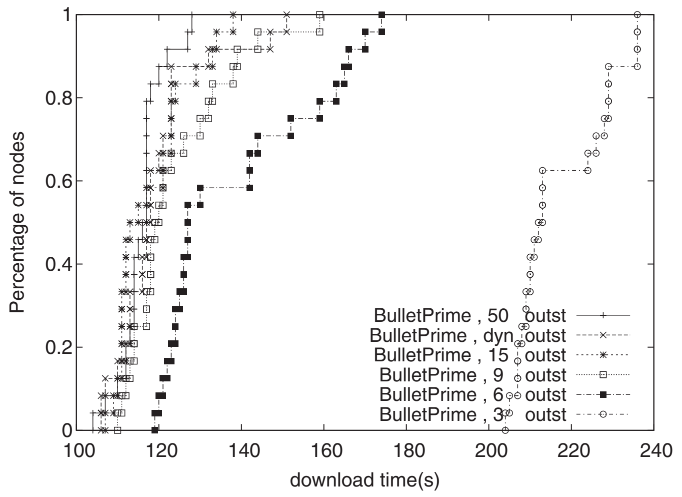

Fig. 18. Bullet′ performance with neither bandwidth changes nor random network packet losses for the static and the dynamic queue sizing cases while downloading a 100-MB file.

control messages, further decreasing the system performance. Our dynamic approach tracks, and sometimes exceeds, the performance of the better static setup.

These cases clearly demonstrate that no statically configured peer set size is appropriate for a wide range of network environments, and a well-tuned system must dynamically determine the appropriate peer set size.

5.11 Outstanding Requests

We now explore determining the optimal number of per-peer outstanding requests. Other systems use a fixed number of outstanding blocks. For example, BitTorrent tries to maintain five outstanding blocks from each peer by default. For the experiments in this section, we used an 8-kB block, and configured the Linux kernel to allow large receiver window sizes. In our first topology, there were 25 participants, interconnected with 10-Mb/s links with 100-ms latency. In Figure 18 we show Bullet′'s performance when configured with 3, 6, 9, 15, and 50 per-peer outstanding blocks for up to five senders. The number of outstanding requests refers to the *total* number of block requests to any given peer, including blocks that are queued for sending, and blocks and requests that are in-flight. As we can see, the dynamic technique closely tracks the performance of cases with a large number of outstanding blocks. Having too few outstanding requests is not enough to fill the bandwidth-delay product of high-bandwidth, high-latency links.

Although it is tempting to simplify the system by requesting the maximum number of blocks from each peer, Figure 19 illustrates the penalty of requesting more blocks than it is required to saturate the TCP connection. In this experiment, we instructed ModelNet to drop packets uniformly at random with probability ranging between 0 and 1.5% on the core links. Due to losses, TCP achieves lower bandwidths, requiring less data in-flight for maximum

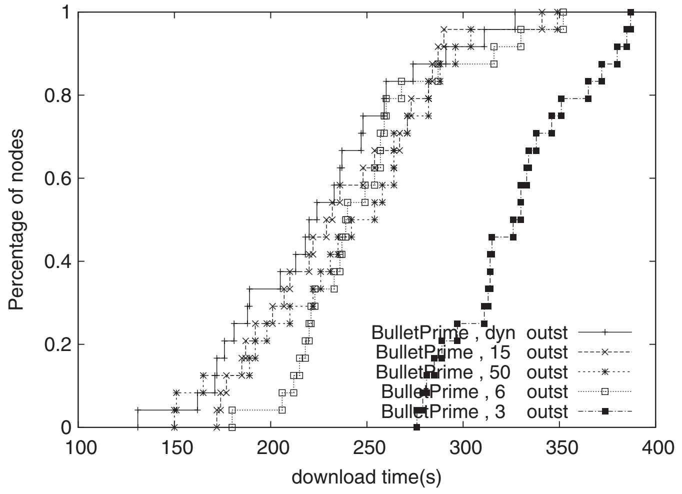

Fig. 19. Bullet′ performance under random network packet losses while downloading a 100-MB file.

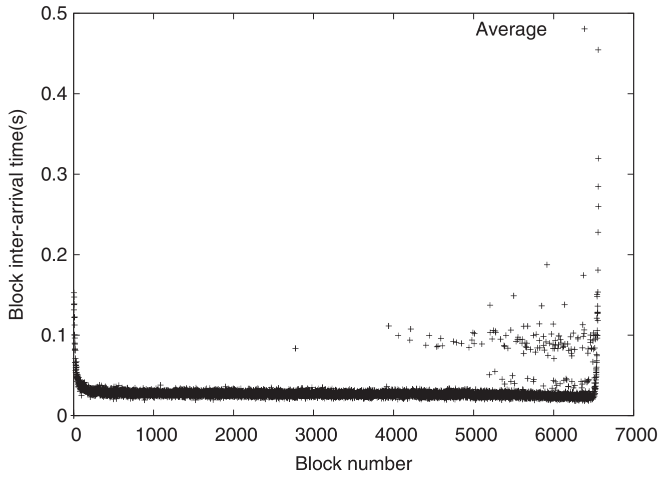

Fig. 20. Block interarrival times for a 100-MB file download under random network packet losses in the absence of bandwidth changes.

performance. Under these loss-induced TCP throughput fluctuations, our dynamic approach outperformed all static cases.

5.12 Potential Benefits of Source Encoding for File Distribution

The purpose of this section is to quantify the potential benefits of encoding the file at the source. Toward this end, Figure 20 shows the average block interarrival times among the 99 receivers while downloading a 100-MB file. The block numbers on the X axis correspond to the order in which nodes retrieve blocks, not the actual block numbers. Further, to improve the readability of the

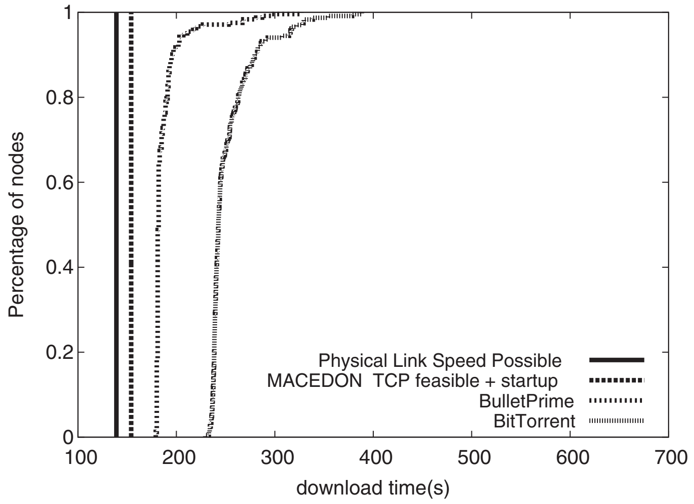

Fig. 21. Performance comparison for a 100-MB file download across 250 nodes under random network packet losses.

graph, we do not show the maximum block interarrival times, which observe a similar trend. A system that has a pronounced "last-block" problem would exhibit a sharp increase in the block interarrival time for the last several blocks. To quantify the potential benefits of encoding, we first compute the overall average block interarrival time t_b. We then consider the last 20 blocks and calculate the cumulative overage of the average block interarrival time over t_b. In this case overage amounts to 8.38 s. We contrast this value to the potential increase in the download time due to a fixed 4% encoding overhead of 7.60 s, while optimistically assuming that downloads using source encoding would not exhibit any deviation in the download times of the last few blocks. We conclude that encoding at the source in this scenario would not be of clear benefit in improving the average download time. This finding can be explained by the presence of a large number of nodes that will have a particular block and will be available to send it to other participants. Encoding at the source or within the network can be useful when the source becomes unavailable soon after sending the file once and with node churn [Gkantsidis and Rodriguez 2005].

5.13 Large-Scale Bullet′ Performance Under Node Churn and Flashcrowds

In this section, we conduct large-scale experiments to subject Bullet′ to demanding node churn and flashcrowd scenarios. To establish a baseline in this environment, we first contrast the 250-node download performance of Bullet′ and BitTorrent while downloading a 100-MB file. As Figure 21 shows, Bullet′ outperforms BitTorrent by 25% and comes close to optimal download times, as in previous experiments.

The next experiment stresses Bullet′ under a flashcrowd scenario. We have conducted all other experiments in our evaluation with all nodes starting simultaneously at t = 0. Here, we started 125 nodes at t = 0 and then waited before

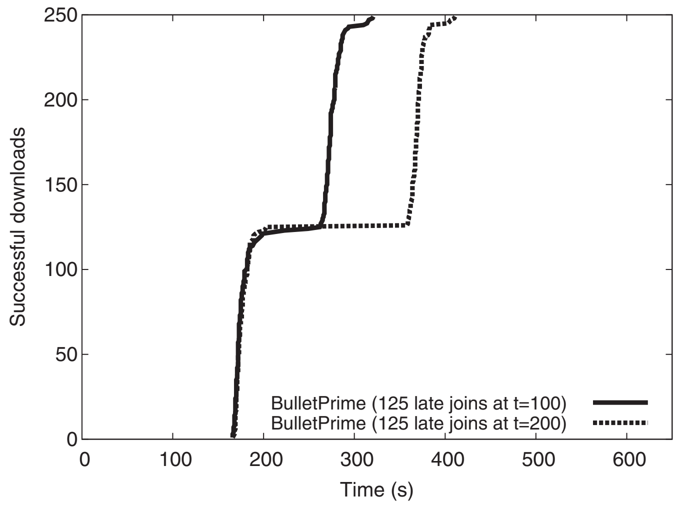

Fig. 22. Bullet′ performance for a 100-MB file download across 250 nodes under random network packet losses and two delayed flash crowd scenarios.

starting the other half. The goal of this experiment was to check whether the late arrivals have difficulty joining as a flash crowd and whether their presence affects the download times of nodes already downloading. We conducted two experiments, with late arrivals occurring at t = 100 and t = 200 s after the start of the experiment. We see in Figure 22 that the initial 125 nodes complete at the same time as when all 250 nodes join at t = 0 (Figure 21). Since in both cases the initial set of nodes finishes at the same time, we conclude that late arrivals have no impact on the download performance of the nodes that are already running when a flash crowd occurs. Further, the two lines in Figure 22 are 100 s apart, which corresponds to the spacing between late arrivals in these two experiments. Therefore, the time at which the late nodes arrive does not affect their download times. We checked the time an incoming node took to start receiving data and to come up to full download bandwidth (more than 5 Mb/s); on average, a late arrival received the first block after 5 s and was downloading at full rate after 10 s. Given that a late arrival needs to: (i) join the random tree, (ii) receive a RanSub distribute set (with RanSub running every 5 s, there is an average 2.5-s wait time) to enable peering, and (iii) explicitly learn of blocks after establishing peering relationships, we believe these startup latencies cannot be completely eliminated.

Finally, we subjected Bullet′ to a node churn scenario where each of the 249 receivers had an exponentially distributed mean lifetime of 5 min (300 s). Every node started downloading the file from scratch. Figure 23 shows the number of nodes that successfully downloaded the 100-MB file in this demanding environment. Comparing the results of this experiment to the baseline (Figure 21), we first see that the nodes that are given enough time to run when they start at t = 0 complete the download at a time that is roughly equivalent to the static scenario. Since this slowdown is less than 5%, we conclude that Bullet′ is resilient to sudden node departures. The remainder of the curve is steady,

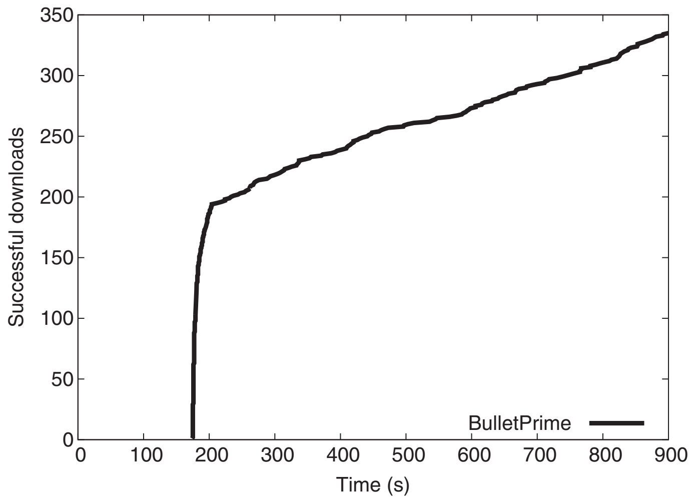

Fig. 23. Bullet′ performance for a 100-MB file download across 250 nodes under random network packet losses and churn (5-min average node uptime).

which means that incoming nodes have no difficulty completing the download under churn. As a matter of fact, most of them complete their downloads in 155 s on average, which is 15% faster than the nodes that started initially. Compared to the late-arriving flash crowds scenario (Figure 22), the download times are quicker because there are fewer nodes that are joining simultaneously. These individual download times are almost at the limit of the physical link bandwidth.

5.14 File Download on PlanetLab

This section contains results from the deployment of Bullet′ over the PlanetLab [Peterson et al. 2002] wide-area network testbed in October of 2004. For our first experiment, we chose 41 nodes for our deployment, with no two machines being deployed at the same site. We configured Bullet′, Bullet, and SplitStream (MACEDON MS implementation) to use a 100-kB block size. Bullet and SplitStream were not performing the file encoding/decoding; instead we marked the downloads as successful when a required number of distinct file blocks was successfully received, including fixed 4% overhead that an actual encoding scheme would incur. We see in Figure 24 that Bullet′ consistently outperforms other systems in the wide area. For example, the slowest Bullet′ node completes the 50-MB downloaded approximately 400 s sooner than BitTorrent's slowest downloader.

In our final PlanetLab experiment, we downloaded a 50-MB file across 250 nodes in May of 2007. We used Plush [Albrecht et al. 2006] to initiate experiments with 450 nodes across all of entire PlanetLab. We started the back-to-back experiments on the same set of nodes when 265 nodes joined the Plush controller and installed the target executables. We used `alice.cs.princeton.edu` as the source and ran the experiments for 900 s. Compared to our experiments

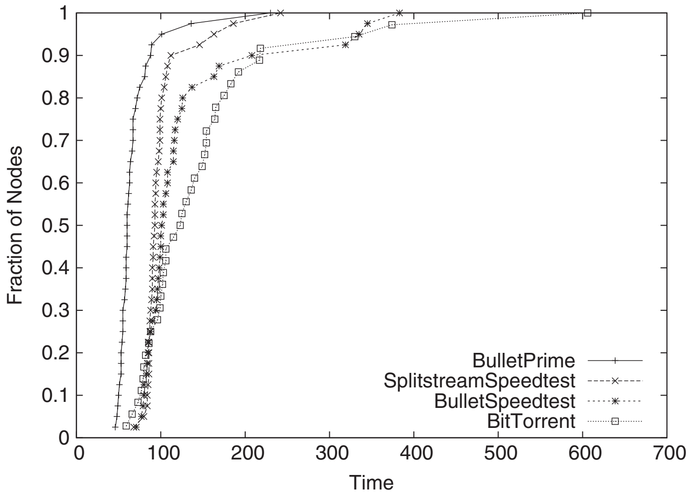

Fig. 24. Comparison of Bullet′ to Bullet, BitTorrent, and SplitStream for 50-MB file download on 41 PlanetLab nodes in October of 2004.

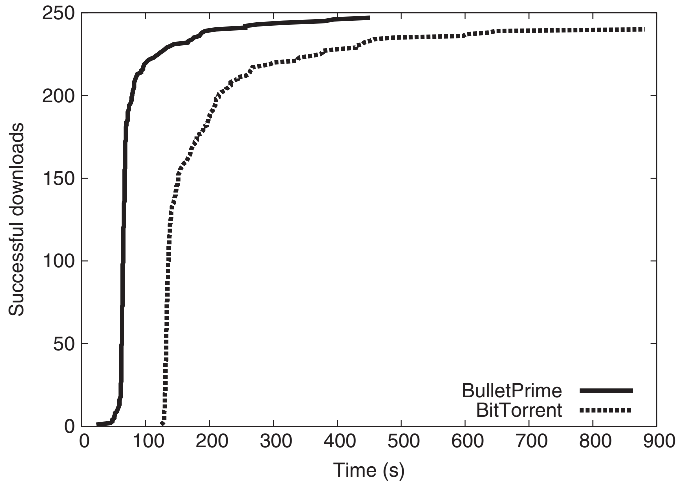

Fig. 25. Comparison of Bullet′ to BitTorrent for 50-MB file download on 250 PlanetLab nodes in May of 2007 (10 BitTorrent nodes did not finish in 900 s).

performed in 2004, we observe higher loads and greater difficulty in locating lightly loaded and responsive nodes; for example, BitTorrent was 10 nodes short in meeting the 250 node target in 900 s. Nevertheless, as Figure 25 shows, Bullet′ maintained the 50% performance advantage over BitTorrent.

6. RELATED WORK

This section describes the related work, divided into logical sections.

6.1 Reliable IP Multicast

The early content distribution mechanisms used IP multicast [Deering 1991], a network level service. To disseminate content to a group of receivers from a single source, routers would organize into a forwarding tree. A number of methods were proposed for IP multicast tree construction. However, numerous problems, including reliability, group management, scalability, congestion control, and dealing with heterogeneity, have prevented the widespread deployment of IP Multicast. Most importantly, even if all other problems were addressed, IP multicast would still be suboptimal for high-bandwidth data dissemination because it does not consider bandwidth when constructing its distribution tree. An IP multicast tree is typically built by reusing portions of the IP point-to-point routing paths [Dalal and Metcalfe 1978], which are known to be suboptimal [Andersen et al. 2001; Savage et al. 1999].

We discuss in more detail the numerous protocols that aim to add reliability to IP multicast. In Scalable Reliable Multicast (SRM) [Floyd et al. 1997], 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 [Birman et al. 1999] showed 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 nonreliable IP-multicast tree. In pbcast [Birman et al. 1999], 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 [Eugster et al. 2001] 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.

6.2 Content Distribution Networks (CDNs)

Content distribution networks (CDNs) [Akamai (`www.akamai.com`); Wang et al. 2004] are purpose-built for disseminating content to a large group of users and comprise tens of thousands of well-provisioned machines deployed at Internet service providers (ISPs). Since they serve HTTP content, these networks are tailored for the pull-based data model, where clients explicitly request a file and the CDN redirects the request to the Web server that is going to provide data quickly. To minimize the file transfer time, the CDN takes into account the network latencies to its server as well as the server loads when making the redirection decision. CDNs are expensive to deploy and operate. Moreover, they are tailored for Web-size content, which is less than 100 kB on average.

CoBlitz [Park and Pai 2006] builds upon CoDeeN [Wang et al. 2004], an existing HTTP Content distribution network, to support dissemination of large files. To prevent deterioration of CDN cache hit rates due to large objects, the system splits a large file into smaller units and reassembles it transparently to the user. In contrast, Bullet′ operates without infrastructure support to achieve high-bandwidth download rates.

6.3 High-Bandwidth Overlay Trees

Built as logical networks on top of the Internet, overlay networks have recently emerged as a fundamental building block for evolving the network architecture. By leveraging end-host storage, bandwidth, and computing power, overlay networks support efficient multicast-style data dissemination without requiring any network support beyond the ubiquitous IP unicast.

Mimicking the IP multicast approach, the typical data dissemination overlay allows participants to self-organize into a tree. Here, the source sends the same data to each of its children, which then serve as intermediaries and forward any data they receive to their own children, all the way down to the tree leaves. We argue that trees have two fundamental limitations for data dissemination. First, since all data comes from a single parent, participants are forced to continuously probe in search of a parent with an acceptable level of bandwidth. Probing for available bandwidth is still an active area of research [Hu and Steenkiste 2003; Jain and Dovrolis 2002], but probing usually involves sending small amounts of useless data or streaming at a specific rate for a short amount time. Systemwide, the probing traffic is a considerable overhead. More importantly, probing interferes with useful data traffic on network links, including the potentially constrained inbound and outbound access links, and reduces overall throughput. Second, due to packet losses and failures, the bandwidth in an overlay tree is monotonically decreasing down the tree. Several mechanisms were proposed for recovering from losses [Birman et al. 1999; Byers et al. 2002]; however the participants are limited to recovering data that is present in the overlay. Recall that every parent attempts to send identical data to all of its children, even when the outbound access bandwidth is constrained.

Narada [hua Chu et al. 2001] 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. A version of the protocol modified for single-source overlay multicast was used for broadcasting of conferences [hua Chu et al. 2004]. 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 [Jannotti et al. 2000] 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.

6.4 Overlay Meshes

The efforts in this category are loosely categorized as overlay meshes. We have categorized them into pioneering efforts, work that promotes use of perpendicular bandwidth, the transition to multiple dissemination trees, and we conclude with “unstructured” meshes that are perhaps closest in spirit to Bullet.

6.4.1 *Pioneering Efforts.* Snoeren et al. [2001] were perhaps the first to use an overlay mesh for data dissemination. In this system, every node chooses n “parents” from which to receive duplicate packet streams. The emphasis of this primarily push-based system is on reliability and timely delivery, so nodes flood the data over the mesh. 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.

6.4.2 *“Perpendicular” Downloads.* The power of “perpendicular” downloads was perhaps first widely demonstrated by Kazaa (Kazaa Media Desktop; go online to `http://www.kazaa.com`), 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.

Informed Content Delivery [Byers et al. 2002] work advocates the use of the perpendicular overlay links to recover data that is made disjoint by losses during dissemination over an overlay tree. Byers et al. offer quick and efficient methods for reconciliation of partially available content between two peers in the overlay tree. Our Bullet prototype uses these techniques and extends them to allow streaming and continuous reconciliation between peers. However, this work does not address an important issue of locating disjoint content. Similarly, authors do not offer a way of retrieving the disjoint data at a high rate. Bullet overcomes both of these shortcomings by using RanSub to locate missing data and by orchestrating data retrieval using the adaptive peering strategy and flow control of data. In addition, Bullet makes data deliberately disjoint at each node to increase throughput.

6.4.3 *Multiple Overlay Trees and Other Structured Overlay Meshes.* FastReplica [Cherkasova and Lee 2003] addresses the problem of reliable and efficient file distribution in content distribution networks. 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.

In parallel to our own work, SplitStream [Castro et al. 2003] 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 [Rowstron et al. 2001]. 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. Therefore, system throughput might be decreased by congestion that does not occur on the access links. 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 [Padmanabhan et al. 2003b] 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 [Goyal 2001] 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 [Padmanabhan et al. 2002] 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.

Young et al. [2004] constructed an overlay mesh of *k* edge-disjoint minimum-cost spanning trees (MSTs). The algorithm for distributed construction of trees uses overlay link metric information such as latency, loss rate, or bandwidth that is determined by a potentially long and bandwidth-consuming probing stage. The resulting trees might start resembling a random mesh if the links have to be excluded in an effort to reduce the probing overhead. In contrast, Bullet′ builds a content-informed mesh and completely eliminates the need for probing because it uses transfers of useful information to adapt to the characteristics of the underlying network.

The latest extension [Sung et al. 2006] of the Narada [hua Chu et al. 2001] work leverages a multitree framework and considers the design of a bandwidth-demanding overlay broadcasting (streaming) application that runs in environments where participants have limited and asymmetric bandwidth, with significant heterogeneity in outgoing bandwidth. To encourage participants to increase their contributions, the system is contribution-aware: it distributes more bandwidth to participants that contribute more. By leveraging this policy, all participants experience improved performance relative to the case where each participant is required to forward only as much bandwidth as it receives (tit-for-tat policy).

6.4.4 *Unstructured Overlay Meshes.* BitTorrent [Cohen 2003] is a system in wide use in the Internet for distribution of large files. Incoming nodes rely on the centralized *tracker* to provide a list of existing system participants and system-wide block distribution for random peering. The tracker presents a single point of failure and limits the system scalability. Lowering the tracker communication rate for file state updates could hurt the overall system performance, as the 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. BitTorrent enforces fairness via a tit-for-tat (TFT) mechanism based on bandwidth. Our inspection of the BitTorrent code reveals hard coded constants for request strategies and peering strategies, potentially limiting the adaptability of the system to a variety of network conditions relative to our approach.

BitTorrent has attracted considerable attention recently. The work on performance modelling of BitTorrent-like protocols [Qiu and Srikant 2004] has shown that unstructured overlay meshes scale well with the number of participants. Massoulie and Vojnovic [2005] further demonstrated that file swarming systems stabilize around a finite equilibrium point, regardless of the arrival rate.

Several improvements have been suggested to the baseline BitTorrent TFT protocol, including a block-based TFT with a limit on the difference between the number of uploaded and download blocks [Bharambe et al. 2006] (the baseline BitTorrent TFT protocol is rate-based). These modifications allow for superior protection against free riders and better integration of nodes with heterogeneous bandwidth. This work also shows that some BitTorrent nodes upload several times more content than they download. Legout et al. [2006] have instrumented a BitTorrent client and shown that the rarest first (referred to as *rarest-random* in this article) algorithm guarantees close to ideal diversity of the blocks among peers. Work on BitTyrant [Piatek et al. 2007] has shown that it is possible to take advantage of the standard tit-for-tat and choking algorithms in BitTorrent to maximize a downloader's performance at the same level of upload contribution. Like Bullet′, this work also shows that having a larger number of senders improves performance for peers with high inbound access bandwidth.

Slurpie [Sherwood et al. 2004] improves upon the performance of BitTorrent by using an adaptive downloading mechanism to scale the number of peers a node should have. However, it does not have a way of dynamically changing the number of outstanding blocks on a per-peer basis. In addition, although Slurpie has a random backoff algorithm that prevents too many nodes from going to the source simultaneously, nodes can connect to the Web server and request arbitrary blocks. This would increase the minimum amount of time it takes all blocks to be made available to the Slurpie network, hence leading to increased minimum download time.

CoolStreaming [Zhang et al. 2005] is a BitTorrent-like system for live streaming over the Internet. CoolStreaming uses a deadline-oriented scheduling algorithm to retrieve data in time for playback. Nodes gossip membership information, which they use to establish random peering relationships. This peering strategy does not account for diversity in the content available at possible peers. Further, in environments with high heterogeneity in terms of bandwidth capacity, the absence of a dynamic peer set sizing algorithm may result in underutilization of a node's access link. Chainsaw [Pai et al. 2005] is another example of an unstructured overlay mesh used for data streaming. Like CoolStreaming, Chainsaw nodes make random peering decisions. In further contrast to Bullet′, Chainsaw uses a fixed minimum number of sending peers, and has no dynamic flow control. Unlike these efforts that use gossiping, Bullet′ uses RanSub, which delivers provably uniformly random subsets of the systemwide state.

Some recent efforts concentrate on reducing the overall communication cost of content distribution. Instead of peering at random, nodes in the Julia Content Distribution Network [Bickson and Malkhi 2005] take latency, bandwidth, and download progress into account when making peering decisions. Compared to BitTorrent, the overall Julia communication cost is 33% lower, at the expense of a slightly slower average download time.

6.4.5 *Network Coding.* Since determining the right set of push-based trees (packing Steiner trees optimally) for data dissemination is APX-hard [Jain et al. 2003], there is an increased interest in the use of network coding [Ahlswede et al. 2000] for data dissemination [Chou et al. 2003]. With network coding, the receivers in the data distribution system act as intermediaries and generate encoded packets that are likely to be useful to downstream nodes, that is, to other receivers. At the cost of additional computation at intermediate nodes, network coding enables random overlay topologies to achieve high throughput [Jain et al. 2005].

Avalanche [Gkantsidis and Rodriguez 2005] is an overlay-based file distribution system that uses network coding. Through simulation, the authors demonstrated the usefulness of producing encoded blocks by all system participants under scenarios when the source departs soon after sending the file once, and on specific network topologies (e.g., two clusters connected with a congested link). In their most recent work on Avalanche, Gkantsidis and Rodriguez [Gkantsidis et al. 2006] demonstrated successful live dissemination of a large file to hundreds of users. They reported low CPU utilization, resilience to unreachable hosts, and successful downloads in the face of premature source departure. It is likely Avalanche will benefit from the techniques outlined in this article. For example, Avalanche participants will have to choose a number of sending peers that will fill their incoming pipes. In addition, receivers will have to negotiate carefully the transfers of encoded blocks produced at random to avoid bandwidth waste due to blocks that do not aid in file reconstruction, while keeping the incoming bandwidth high from each peer.

Producing a high-performance content distribution system based on network coding is not trivial. Much like with erasure encoding, decoding a file requires random access to the encoded blocks. This in turn might cause slow decoding performance if the file does not fit into physical memory. Further, decoding entails solving a system of linear equations, an $O(n^3)$ operation, where n is the number of original blocks. Reducing n by increasing the block size can result in network inefficiencies due to long wait times for block downloads.

7. CONCLUSIONS

Given the suboptimal performance and reliability of existing systems, this work operates on the premise that the model for high-bandwidth multicast data dissemination should be reexamined, and makes the following contributions:

One key contribution of this work is the design and analysis of Bullet, an overlay construction algorithm that creates a mesh over any distribution tree that matches the properties of the underlying network topology. As a benefit, Bullet eliminates the overhead required to probe for available bandwidth in traditional distributed tree construction techniques. Another contribution of this work is a mechanism for making data deliberately disjoint and then distributing it in a uniform way that makes the probability of finding a peer containing missing data equal for all nodes. An insight that makes Bullet scalable is the use of RanSub for locating missing data from peers in an efficient manner. Further, we propose a mechanism for recovering data from peers in a disjoint manner that minimizes retrieval of duplicate data objects. A large-scale evaluation shows that Bullet running over a random tree can achieve twice the throughput of streaming over a bandwidth tree computed by an off-line algorithm with perfect information about the network.

Bullet′ continues the overlay mesh data dissemination approach advocated by Bullet, and considers the problem of large file dissemination to a large group of Internet users. The first contribution of this work is the exploration of the design space of file distribution protocols. Specifically, we examine the problems of source sending strategy, push versus pull of data, need for data encoding, download peer set selection, data request strategy, and flow control. Second, we conduct a detailed performance evaluation of a number of competing systems running in both controlled emulation environments and live across the Internet. Our experience shows that protocols which have tunable parameters might perform well in a certain range of conditions, but that once outside that range they will break down and perform worse than if they had been tuned differently. To combat this problem, we employ adaptive strategies to self-tune to dynamically changing network conditions. For example, Bullet′ uses a feedback control loop to determine dynamically the quantity of outstanding data requested from each sending peer. This method of flow control keeps the data connection occupied while risking waiting for the least amount of data in case bandwidth from the peer deteriorates. We have compared Bullet′ with BitTorrent and SplitStream. In the scenarios we considered, Bullet′ outperforms the other systems.

REFERENCES

Ahlswede, R., Cai, N., Li, S.-Y. R., and Yeung, R. W. 2000. Network information flow. *IEEE Trans. Inform. Theor. 46*, 4 (Jul.), 1204–1216.

Albrecht, J., Tuttle, C., Snoeren, A. C., and Vahdat, A. 2006. PlanetLab application management using Plush. *ACM SIGOPS Operat. Syst. Rev. 40* 1 (Jan.), 33–40.

Andersen, D. G., Balakrishnan, H., Kaashoek, M. F., and Morris, R. 2001. Resilient overlay networks. In *Proceedings of the Symposium on Operating Systems Principles*.

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

Bharambe, A., Herley, C., and Padmanabhan, V. 2006. Analyzing and improving a BitTorrent network's performance mechanisms. In *Proceedings of IEEE INFOCOM*.

Bickson, D. and Malkhi, D. 2005. The Julia content distribution network. In *Proceedings of the Second Workshop on Real, Large Distributed Systems* (WORLDS).

Birman, K., Hayden, M., Ozkasap, O., Xiao, Z., Budiu, M., and Minsky, Y. 1999. Bimodal multicast. *ACM Trans. Comput. Syst. 17,* 2 (May), 41–88.

Bloom, B. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM 13,* 7 (Jul.), 422–426.

Broder, A. 1997. On the resemblance and containment of documents. In *Proceedings of the Conference on Compression and Complexity of Sequences* (SEQUENCES'97).

Byers, J. W., Considine, J., Mitzenmacher, M., and Rost, S. 2002. Informed content delivery across adaptive overlay networks. In *Proceedings of ACM SIGCOMM*.

Byers, J. W., Luby, M., Mitzenmacher, M., and Rege, A. 1998. A digital fountain approach to reliable distribution of bulk data. In *Proceedings of ACM SIGCOMM*.

Calvert, K., Doar, M., and Zegura, E. W. 1997. Modeling Internet topology. *IEEE Commun. Mag. 35*, 6 (Jun.), 160–163.

Castro, M., Druschel, P., Kermarrec, A.-M., Nandi, A., Rowstron, A., and Singh, A. 2003. Splitstream: High-bandwidth content distribution in cooperative environments. In *Proceedings of the 19th ACM Symposium on Operating System Principles*.

CHANG, H., GOVINDAN, R., JAMIN, S., SHENKER, S., AND WILLINGER, W. 2002. Towards capturing representative AS-level Internet topologies. In *Proceedings of ACM SIGMETRICS*.

CHERKASOVA, L. AND LEE, J. 2003. FastReplica: Efficient large file distribution within content delivery networks. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*.

CHOU, P. A., WU, Y., AND JAIN, K. 2003. Practical network coding. In *Proceedings of the Allerton Conference on Communication, Control, and Computing*.

COHEN, B. 2003. Incentives build robustness in bittorrent. In *Proceedings of the First Workshop on the Economics of Peer-to-Peer Systems*.

COHEN, R. AND KAEMPFER, G. 2001. A unicast-based approach for streaming multicast. In *Proceeding of INFOCOM*. 440–448.

DABEK, F., LI, J., SIT, E., KAASHOEK, F., MORRIS, R., AND BLAKE, C. 2004. Designing a DHT for low latency and high throughput. In *Proceedings of the USENIX/ACM Symposium on Networked Systems Design and Implementation* (NSDI, San Francisco, CA).

DALAL, Y. K. AND METCALFE, R. M. 1978. Reverse path forwarding of broadcast packets. *Commun. ACM 21,* 12, 1040–1048.

DEERING, S. E. 1991. Multicast routing in a datagram internetwork. Ph.D. dissertation. Stanford University, Stanford, CA.

DEMERS, A., GREENE, D., HAUSER, C., IRISH, W., LARSON, J., SHENKER, S., STURGIS, H., SWINEHART, D., AND TERRY, D. 1987. Epidemic algorithms for replicated database maintenance. In *Proceedings of the Sixth Symposium on Principles of Distributed Computing*. 1–12.

EUGSTER, P., GUERRAOUI, R., HANDURUKANDE, S., KOUZNETSOV, P., AND KERMARREC, A.-M. 2003. Lightweight probabilistic broadcast. *ACM Trans. Comput. Syst. 21,* 4, 341–374.

EUGSTER, P., HANDURUKANDE, S., GUERRAOUI, R., KERMARREC, A.-M., AND KOUZNETSOV, P. 2001. Lightweight probabilistic broadcast. In *Proceedings of The International Conference on Dependable Systems and Networks* (DSN).

FLOYD, S., HANDLEY, M., PADHYE, J., AND WIDMER, J. 2000. Equation-based congestion control for unicast applications. In *Proceedings of ACM SIGCOMM* (Stockholm, Sweden). 43–56.

FLOYD, S., JACOBSON, V., LIU, C.-G., MCCANNE, S., AND ZHANG, L. 1997. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Trans. Network. 5,* 6, 784–803.

GKANTSIDIS, C., MILLER, J., AND RODRIGUEZ, P. 2006. Anatomy of a p2p content distribution system with network coding. In *Proceedings of the Second International Peer to Peer Symposium* (IPTPS).

GKANTSIDIS, C. AND RODRIGUEZ, P. R. 2005. Network coding for large scale content distribution. In *Proceedings of IEEE INFOCOM*.

GOYAL, V. K. 2001. Multiple description coding: Compression meets the network. *IEEE Signal Process. Mag. 18,* 5, 74–93.

HU, N. AND STEENKISTE, P. 2003. Evaluation and characterization of available bandwidth probing techniques. *IEEE J. Select are Commun. 21,* 6, 879–895.

HUA CHU, Y., GANJAM, A., NG, T. S. E., RAO, S. G., SRIPANIDKULCHAI, K., ZHAN, J., AND ZHANG, H. 2004. Early experience with an Internet broadcast system based on overlay multicast. In *Proceedings of the USENIX 2004 Annual Technical Conference*.

HUA CHU, Y., RAO, S., AND ZHANG, H. 2000. A case for end system multicast. In *Proceedings of the ACM Sigmetrics 2000 International Conference on Measurement and Modeling of Computer Systems*.

HUA CHU, Y., RAO, S. G., SESHAN, S., AND ZHANG, H. 2001. Enabling Conferencing Applications on the Internet using an overlay multicast architecture. In *Proceedings of ACM SIGCOMM*.

INFORMATIONWEEK. 2004. Go online to `http://informationweek.com/story/showArticle.jhtml?articleID=50900297`.

JAIN, K., LOVASZ, L., AND CHOU, P. A. 2005. Building scalable and robust peer-to-peer overlay networks for broadcasting using network coding. In *Proceedings of the ACM Conference on Principles Of Distributed Computing* (PODC).

JAIN, K., MAHDIAN, M., AND SALAVATIPOUR, M. R. 2003. Packing Steiner trees. In *Proceeding of the 14th ACMSIAM Symposium on Discrete Algorithms*.

Jain, M. and Dovrolis, C. 2002. End-to-end available bandwidth: Measurement methodology, dynamics, and relation with TCP Throughput. In *Proceedings of ACM SIGCOMM*.

Jannotti, J., Gifford, D. K., Johnson, K. L., Kaashoek, M. F., and James W. O'Toole, J. 2000. Overcast: Reliable multicasting with an overlay network. In *Proceedings of the Conference on Operating Systems Design and Implementation* (OSDI).

Katabi, D., Handley, M., and Rohrs, C. 2002. Internet congestion control for high bandwidth-delay product networks. In *Proceedings of ACM SIGCOMM*.

Kim, M. S., Lam, S. S., and Lee, D.-Y. 2002. Optimal distribution tree for Internet streaming media. Tech. rep. TR-02-48. Department of Computer Sciences, University of Texas at Austin, Austin, TX.

Kostić, D., Braud, R., Killian, C., Vandekieft, E., Anderson, J. W., Snoeren, A. C., and Vahdat, A. 2005. Maintaining high bandwidth under dynamic network conditions. In *Proceedings of the USENIX 2005 Annual Technical Conference*.

Kostić, D., Rodriguez, A., Albrecht, J., Bhirud, A., and Vahdat, A. 2003a. Using random subsets to build scalable network services. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*.

Kostić, D., Rodriguez, A., Albrecht, J., and Vahdat, A. 2003b. Bullet: High bandwidth data dissemination using an overlay mesh. In *Proceedings of the 19th ACM Symposium on Operating System Principles*.

Krohn, M. N., Freedman, M. J., and Mazieres, D. 2004. On-the-fly verification of rateless erasure codes for efficient content distribution. In *Proceedings of the IEEE Symposium on Security and Privacy* (Oakland, CA).

Legout, A., Urvoy-Keller, G., and Michiardi, P. 2006. Rarest first and choke algorithms are enough. In *IMC'06: Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement*. 203–216.

Luby, M. 2002. LT codes. In *Proceedings of 43rd Annual IEEE Symposium on Foundations of Computer Science*.

Luby, M. G., Mitzenmacher, M., Shokrollahi, M. A., Spielman, D. A., and Stemann, V. 1997. Practical loss-resilient codes. In *Proceedings of the 29th Annual ACM Symposium on the Theory of Computing* (STOC '97). Ass. Comp. Mach. Press, New York, 150–159.

Massoulie, L. and Vojnovic, M. 2005. Coupon replication systems. In *Proceedings of ACM SIGMETRICS*.

Maymounkov, P. and Mazieres, D. 2003. Rateless codes and big downloads. In *Proceedings of the Second International Peer to Peer Symposium* (IPTPS).

Padhye, J., Firoiu, V., Towsley, D., and Krusoe, J. 1998. Modeling TCP throughput: A simple model and its empirical validation. In *Proceedings of ACM SIGCOMM* (Vancouver, Canada). 303–314.

Padmanabhan, V. N., Qiu, L., and Wang, H. J. 2003. Server-based inference of Internet link lossiness. In *Proceedings of the IEEE INFOCOM*. (San Francisco, CA).

Padmanabhan, V. N., Wang, H. J., and Chou, P. A. 2003b. Resilient peer-to-peer streaming. In *Proceedings of the 11th ICNP* (Atlanta, GA).

Padmanabhan, V. N., Wang, H. J., Chou, P. A., and Sripanidkulchai, K. 2002. Distributing streaming media content using cooperative networking. In *Proceedings of ACM/IEEE NOSSDAV*.

Pai, V., Kumar, K., Tamilmani, K., Sambamurthy, V., and Mohr, A. E. 2005. Chainsaw: Eliminating trees from overlay multicast. In *Proceedings of I PT PS*.

Park, K. and Pai, V. S. 2006. Scale and performance in the CoBlitz large-file distribution service. In *Proceedings of the 3rd USENIX/ACM Symposium on Networked Systems Design and Implementation* (NSDI).

Peterson, L., Anderson, T., Culler, D., and Roscoe, T. 2002. A blueprint for introducing disruptive technology into the Internet. In *Proceedings of ACM HotNets-I*.

Piatek, M., Isdal, T., Anderson, T., Krishnamurthy, A., and Venkataramani, A. 2007. Do incentives build robustness in BitTorrent? In *Proceedings of the 4th USENIX/ACM Symposium on Networked Systems Design and Implementation* (NSDI).

Prim, R. C. 1957. Shortest connection networks and some generalizations. In *Bell Syst. Tech. J. 36*, 6, 1389–1401.

QIU, D. AND SRIKANT, R. 2004. Modeling and performance analysis of Bittorrentlike peer-to-peer networks. In *Proceedings of ACM SIGCOMM*.

RODRIGUEZ, A., KILLIAN, C., BHAT, S., KOSTIĆ, D., AND VAHDAT, A. 2004a. MACEDON: Methodology for automatically creating, evaluating, and designing overlay networks. In *Proceedings of the USENIX/ACM Symposium on Networked Systems Design and Implementation* (NSDI, San Francisco, CA).

RODRIGUEZ, A., KOSTIĆ, D., AND VAHDAT, A. 2004b. Scalability in adaptive multi-metric overlays. In *Proceedings the 24th International Conference on Distributed Computing Systems* (ICDCS).

ROWSTRON, A., KERMARREC, A.-M., CASTRO, M., AND DRUSCHEL, P. 2001. SCRIBE: The design of a large-scale event notification infrastructure. In *Proceedings of the Third International Workshop on Networked Group Communication*.

SAROIU, S., GUMMADI, K. P., DUNN, R. J., GRIBBLE, S. D., AND LEVY, H. M. 2002. An analysis of Internet content delivery systems. In *Proceedings of 5th Symposium on Operating Systems Design and Implementation* (OSDI).

SAVAGE, S., COLLINS, A., HOFFMAN, E., SNELL, J., AND ANDERSON, T. 1999. The end-to-end effects of Internet path selection. In *Proceedings of ACM SIGCOMM*.

SHERWOOD, R., BRAUD, R., AND BHATTACHARJEE, B. 2004. Slurpie: A cooperative bulk data transfer protocol. In *Proceedings of IEEE INFOCOM*.

SHOKROLLAHI, A. 2003. Raptor codes. Tech. rep. DF2003-06-001. Digital Fountain, Inc.

SNOEREN, A. C., CONLEY, K., AND GIFFORD, D. K. 2001. Mesh-based content routing using XML. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles* (SOSP).

SUNG, Y.-W., BISHOP, M., AND RAO, S. 2006. Enabling contribution awareness in an overlay broadcasting system. In *Proceedings of ACM SIGCOMM*.

VAHDAT, A., YOCUM, K., WALSH, K., MAHADEVAN, P., KOSTIĆ, D., CHASE, J., AND BECKER, D. 2002. Scalability and accuracy in a large-scale network emulator. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (OSDI).

WANG, L., PARK, K., PANG, R., PAI, V. S., AND PETERSON, L. 2004. Reliability and Security in the CoDeeN content distribution network. In *Proceedings of the USENIX 2004 Annual Technical Conference*.

YOUNG, A., CHEN, J., MA, Z., KRISHNAMURTHY, A., PETERSON, L., AND WANG, R. Y. 2004. Overlay mesh construction using interleaved spanning trees. In *Proceedings of IEEE INFOCOM*.

ZHANG, X., LIUY, J., LIZ, B., AND YUM, T.-S. P. 2005. CoolStreaming/DONet: A data-driven overlay network for efficient live media streaming. In *Proceedings of IEEE INFOCOM*.

Received March 2006; revised June 2007; accepted November 2007