Maintaining High Bandwidth under Dynamic Network Conditions

Dejan Kostić, Ryan Braud, Charles Killian, Erik Vandekieft, James W. Anderson,
Alex C. Snoeren and Amin Vahdat *
Department of Computer Science and Engineering
University of California, San Diego
{dkostic, rbraud, ckillian, evandeki, jwanderson, snoeren, vahdat}@cs.ucsd.edu

Abstract

The need to distribute large files across multiple wide-area sites is becoming increasingly common, for instance, in support of scientific computing, configuring distributed systems, distributing software updates such as open source ISOs or Windows patches, or disseminating multimedia content. Recently a number of techniques have been proposed for simultaneously retrieving portions of a file from multiple remote sites with the twin goals of filling the client's pipe and overcoming any performance bottlenecks between the client and any individual server. While there are a number of interesting tradeoffs in locating appropriate download sites in the face of dynamically changing network conditions, to date there has been no systematic evaluation of the merits of different protocols. This paper explores the design space of file distribution protocols and conducts a detailed performance evaluation of a number of competing systems running in both controlled emulation environments and live across the Internet. Based on our experience with these systems under a variety of conditions, we propose, implement and evaluate Bullet$'$ (Bullet prime), a mesh based high bandwidth data dissemination system that outperforms previous techniques under both static and dynamic conditions.

1 Introduction

The rapid, reliable, and efficient transmission of a data object from a single source to a large number of receivers spread across the Internet has long been the subject of research and development spanning computer systems, networking, algorithms, and theory. Initial attempts at addressing this problem focused on IP multicast, a network level primitive for constructing efficient IP-level trees to deliver individual packets from the source to each receiver. Fundamental problems with reliability, congestion control, heterogeneity, and deployment limited the widespread success and availability of IP multicast. Building on the experience gained from IP multicast, a number of efforts then focused on building *application-level overlays* [2, 9, 10, 12, 24] where a source would transmit the content, for instance over multiple TCP connections, to a set of receivers that would implicitly act as interior nodes in a tree built up at the application level.

While certainly advancing the state of the art, tree-based techniques face fundamental reliability and performance limitations. First, bandwidth down an overlay tree is guaranteed to be monotonically decreasing. For instance, a single packet loss in a TCP transmission high up in a tree can significantly impact the throughput to all participants rooted at the node experiencing the loss. Hence, in a large overlay, participants near the leaves are likely to experience more limited bandwidth. Further, because each node only receives data from its parent, the failure of a single node can dramatically impact overall system reliability and may cause scalability concerns under churn as a potentially large number of nodes attempt to rejoin the tree. Reliability in overlay trees is perhaps even more of a concern than in IP-level multicast, since end hosts are more likely to fail or to suffer from performance anomalies relative to IP routers in the interior of the network.

Thus, "third-generation" data dissemination infrastructures have most recently focused on building application-level *overlay meshes* [3, 5, 13, 25]. The idea here is that end hosts self-organize into a general mesh by selecting multiple application-level peers. Each peer transmits a disjoint set of the target underlying data to the node, enabling the potential to greater resiliency to failure and increased performance. Reliability is improved because the failure of any single peer will typically only reduce the transmitted bandwidth by $1/n$ where n is the total number of peers being maintained by a given node. In fact, with a sufficient numbers of peers, the failure of any one peer may go unnoticed because the remaining peers may be able to quickly make up for the lost bandwidth. This additional redundancy also enables each

*This research is supported in part by the Center for Networked Systems, Hewlett Packard, IBM, and Intel. Vahdat and Snoeren are supported by NSF CAREER awards (CCR-9984328, CNS-0347949).

node to regenerate its peer set in a more leisurely manner, eliminating the "reconnection storm" that can result from the failure of a node in an overlay tree. Mesh-based approaches may also improve overall bandwidth delivered to all nodes as data can flow to receivers over multiple disjoint paths. It is straightforward to construct scenarios where two peers can deliver higher aggregate bandwidth to a given node than any one can as a result of bottlenecks in the middle of the network.

While a number of these mesh-based systems have demonstrated performance and reliability improvements over previous tree-based techniques, there is currently no detailed understanding of the relative merits of existing systems nor an understanding of the set of design issues that most significantly affect end-to-end system performance and reliability. We set out to explore this space, beginning with our own implementation of Bullet [13]. Our detailed performance comparison of these systems both live across PlanetLab [20] and in the ModelNet [28] emulation environment led us to identify a number of key design questions that appear to most significantly impact overall system performance and reliability, including: i) whether data is pushed from the source to receivers or pulled by individual receivers based on discovering missing data items, ii) peer selection algorithms where nodes determine the number and identity of peers likely to have a good combination of high bandwidth and a large volume of missing data items, iii) the strategy for determining which data items to request and in what quantity based on changing network conditions, iv) the strategy employed by the sender to transmit blocks to its current set of peers based on the data items currently queued for transmission, and v) whether the data stream should be encoded using, for instance, Erasure codes [17] or transmitted unmodified.

By examining these questions in detail, we designed and implemented *Bullet′* (pronounced as Bullet prime), a new mesh-based data dissemination protocol that outperforms existing systems, including Bullet, SplitStream, and BitTorrent, by 25-70% in terms of download times, under a variety of network conditions. Thus, the principal contribution of this work is an exploration and initial enumeration of the design space for a broad range of mesh-based overlays and the design, implementation, and evaluation of a system that we believe embodies some of the "best practices" in this space, enabling good performance and reliability under a variety of network conditions, ranging from the static to the highly dynamic.

It is becoming increasingly important to synchronize updates to a logical data volume across the Internet, such as synchronizing FTP mirrors or updating installed software in distributed testbeds such as PlanetLab and the Grid. Today, the state of the art for disseminating updates consists of having all clients contact a central repository known to contain the most current version of the data to retrieve any updates. Unfortunately, limited bandwidth and CPU at any central repository limit the speed and reliability with which the data can be synchronized across a large number of sites. We designed and implemented Shotgun, a set of extensions to the popular `rsync` tool to enable clients to synchronize their state with a centralized server orders of magnitude more quickly than previously possible.

The rest of this paper is organized as follows. Section 2 describes the design space for all systems doing large scale file distribution, Section 3 describes the specific implementation and architecture of Bullet′, and Section 4 presents the results of our experiments testing our experimental strategies and comparisons with other systems, including evaluation of Shotgun. Section 5 outlines the related work in the field, and the paper concludes in Section 6.

2 Design Principles and Patterns

Throughout the paper, we assume that the source transmits the file as a sequence of *blocks*, that serve as the smallest transfer unit. Otherwise, peers would not be able to help each other until they had the entire file downloaded. In addition, we concentrate on the case when the source of the file is the only node that has the file 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 a popular file over the Internet. In our terminology for peering relationships, a *sender* is a node sending data to a *receiver*.

The design of any large-scale file distribution system can be realized as careful decisions made regarding the fundamental tenets of peer-to-peer applications and data transfer. As we see them, these tenets are push vs. 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 of downloading files is *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 which all systems disseminating objects must make in more detail.

2.1 Push or Pull

Options for distributing the file 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, 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 have multiple senders to them, 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 which data 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 the file involves just sending the original, or *unencoded*, blocks into the overlay. An advantage of this approach is that receivers typically do not have to fit the entire file into physical memory to sustain high-performance. Incoming blocks can be cached in memory, and later written to disk. Blocks only have to be read when the peers request them. Even if the blocks 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 blocks of the file.

Recently, a number of erasure-correcting codes that implement the "digital fountain" [4] approach were suggested by researchers [15, 16, 17]. When the source encodes the file with these codes, *any* $(1+\epsilon)n$ correctly received *encoded* blocks are sufficient to reconstruct the original n blocks, 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 block, as long as it recovers a sufficient number of distinct blocks. In this paper, we assume that only the source is capable of encoding the file, and do not consider the potential benefits of *network coding* [1], 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 [17]. 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 blocks is successfully received. Even with n received blocks (i.e., corresponding to the original file size), only 30 percent of the file content can be reconstructed [14]. Further, since the encoded blocks are computed by XOR-ing random sets of original blocks, the decoding stage requires random access to all of the reconstructed file blocks. This pattern of access, coupled with the bursty nature of the decoding process, causes increased decoding time due to excessive disk swapping if all of the decoded file blocks cannot fit into physical memory [1]. Consequently, the source is forced to transmit the file as a serious of *segments* that can fit into physical memory of the receivers. Even if all receivers have homogeneous memory size, this arrangement presents few problems. First, the source needs to decide when to start transmitting encoded blocks that belong to the next segment. If the file distribution mechanism exhibits considerable latency between the time a block is first generated and the time when nodes receive it, the source might send too many unnecessary blocks 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 4 percent overhead when encoding and decoding files of tens of MBs in size. Although mathematically possible, it is difficult to make this overhead arbitrary small via parameter settings or a large number of blocks. Since the file blocks should be sufficiently large to overcome the fixed overhead due to per-block headers, we cannot use an excessive number of blocks. Similarly, since a file consisting of a large number of blocks may not fit into physical memory, a segment may not have enough blocks to reduce the decoding overhead. Finally, the decoding process is sensitive to the number of recovered degree-1 (unencoded) blocks that are generated with relatively low probability (e.g. 0.01). These blocks are necessary to start the decoding process and without a sufficient number of these blocks the decoding process cannot complete.

In the remainder of the paper, we optimistically assume that the entire file can fit into physical memory and quantify the potential benefits of using encoding at the source in Section 4.6.

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

2.3 Peering Strategy

A node receives the file by peering with neighbors and receiving blocks from them. To enable this, the node requires techniques to learn about nodes in the system, selecting ones to peer with which have useful data, 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 distribution of blocks throughout the system and would be able to download any block 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 blocks 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 blocks. 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 block. Overall, a good approach to picking peers would be one which neither causes nodes to maintain global knowledge, nor 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 which is 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 blocks should be queued to send to which peers. In a pull based system, this is a request for block. 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 blocks? For example, if all nodes make requests for the blocks 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.

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

Third, how much data should be requested from any given sender? Requesting too few blocks might not fill the node's incoming pipe, whereas requesting too much data might force the receiver to wait too long for a block 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 blocks, which comes at the expense of maintaining data structures that describe the availability of each block, the time of requests, etc. In addition, this approach might incur considerable CPU overhead for choosing the next block 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 blocks it currently has. Since a node might implicitly request the same block from multiple senders, this approach is highly resilient. On the other hand, duplicate blocks 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 a file 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 blocks rather than *which* blocks to queue. Here, we separate the strategy of the single source from that of the many peers, since for any file distribution to be successful, the source must share all parts of the file with others.

Source

In this paper, we consider the case where the source is available to send file blocks for the entire duration of a file distribution 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 block. 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.

Non-source

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 blocks 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. The utility to nodes who have peered with both B and C is just 1 though, since there is only 1 new block available. But if node A sends blocks in different orders, the utility to peers of B and C is doubled. A good sending strategy will create the greatest diversity of blocks in the system.

2.6 Fair-first or Fast-first

An important consideration for the design of a file distribution 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 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.

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 indicate that there exist nodes willing to give more than they take, while other nodes cannot share fairly due to network configurations and firewalls. In the remainder of the paper, we assume that nodes are willing to cooperate both during and after the file download, and do not consider enforcing fairness.

3 Implementation and Architecture

After a careful analysis of the design space, we came to the following conclusions about how Bullet′ was to be structured:

- Bullet′ would use a hybrid push/pull (Section 2.1) architecture where the source pushes and the receivers pull.
- The number of parameters that could be tweaked by the end user to increase performance must be minimized.
- Our system would support both the unencoded and the source-encoded file transmission, described in Section 2.2. We will quantify the potential benefits of encoding.

3.1 Architectural Overview

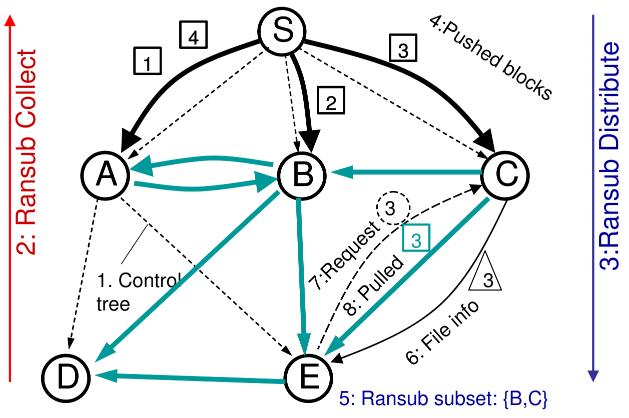

Figure 1: Bullet′ architectural overview. Gray lines represent a subset of peering relationships that carry explicitly pulled blocks.

Figure 1 depicts the architectural overview of Bullet′. We use an overlay tree for joining the system and for transmitting control information (shown in thin dashed lines, as step 1). We use RanSub [12], a scalable, decentralized protocol, to distribute changing, uniformly random subsets of file summaries over the control tree (steps 2 and 3). The source pushes the file blocks to children in the control tree (step 4). Using RanSub, nodes advertise their identity and the block availability. Receivers use this information (step 5) to choose a set of senders to peer with, receive their file information (step 6), request (step 7) and subsequently receive (step 8) 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. Meanwhile, senders keep their receivers updated with the description of their newly received file blocks (step 6). The specifics of our implementation are described below.

3.2 Implementation

We have implemented Bullet′ using MACEDON. MACEDON [22] is a tool which makes the development of large scale distributed systems simpler by allowing us to specify the overlay network algorithm without simultaneously implementing code to handle data transmission, timers, etc. The implementation of Bullet′ consists of a generic file distribution application, the Bullet′ overlay network algorithm, the existing basic random tree and RanSub algorithms, and the library code in MACEDON.

3.2.1 Download Application

The generic download application implemented for Bullet′ can operate in either encoded or unencoded mode. In the encoded mode, it operates by generating continually increasing block numbers and transmitting them using `macedon_multicast` 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 is parameterized and can take as parameters block size and file name.

3.2.2 RanSub

RanSub is the protocol we use for distributing uniformly random subsets of peers to all nodes in the overlay. Unlike a centralized solution or one which requires state on the order of the size of the overlay, Ransub is decentralized and can scale better with both the number of nodes and the state maintained. RanSub requires an overlay tree to propagate random subsets, and in Bullet′ we use the control tree for this purpose. RanSub works by periodically distributing a message to all members of the overlay, and then collecting data back up the tree. At each layer of the tree, data is randomized and compacted, assuring that all peers see a changing, uniformly random subset of data. The period of this distribute and collect is configurable, but for Bullet′, it is set for 5 seconds. Also, by decentralizing the random subset distribution, we can include more application-state, an abstract which nodes can use to make more intelligent decisions about which nodes would be good to peer with.

3.3 Algorithms

3.3.1 Peering Strategy

In order for nodes to fill their pipes with useful data, it is imperative that they are able to locate and maintain a set of peers that can provide them with good service. In the face of bandwidth changes and unstable network conditions, keeping a fixed number of peers is suboptimal (see Section 4.4). Not only must Bullet′ discard peers whose service degrades, 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 term “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 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 2 shows the pseudocode for managing senders.

Once `MAX_SENDERS` and `MAX_RECEIVERS` have been modified, 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

```
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++;
  }
}
```

Figure 2: ManageSenders() Pseudocode

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 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.

3.3.2 Request Strategy

We considered using four different request strategies 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′. We present our findings in Section 4.3.

3.3.3 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

```
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);
}
```

Figure 3: Pseudocode for setting the maximum number of per-peer outstanding blocks.

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 4.5, 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 [11] 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 0 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.

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 1 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 3 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 non-integer 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 3 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 re-evaluates the potential from this peer and requests up to the new maximum outstanding.

3.3.4 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.

3.3.5 Sending Strategy

As mentioned previously, Bullet$'$ uses a hybrid push/pull approach for data distribution where the source behaves differently from everyone else. 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

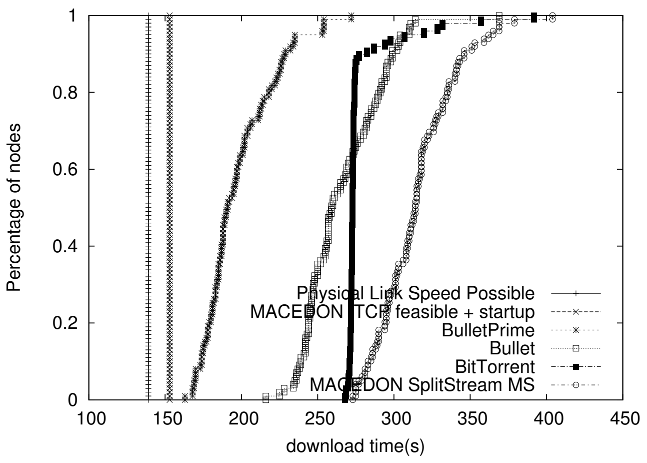

Figure 4: Comparison of Bullet′ to BitTorrent, Bullet, SplitStream and optimal case for 100MB file size under random network packet losses. The legend is ordered top to bottom, while the lines are ordered from left to right.

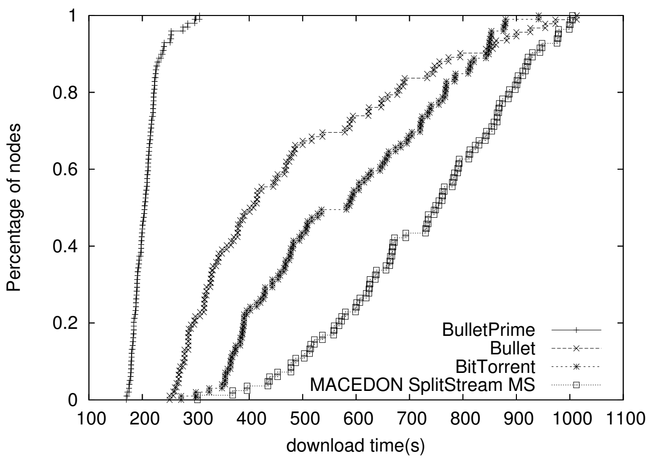

Figure 5: Comparison of Bullet′ to BitTorrent, Bullet, and SplitStream for 100MB file size under synthetic bandwidth changes and random network packet losses.

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.

From the perspective of non-source 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.

4 Evaluation

To conduct a rigorous analysis of our various design tradeoffs, we needed to construct a controlled experimental environment where we could conduct multiple tests under identical conditions. Rather than attempt to construct a necessarily small isolated network test-bed, we present results from experiments using the ModelNet [28] network emulator, which allowed us to evaluate Bullet′ on topologies consisting of 100 nodes or more. In addition, we present experimental results over the wide-area network using the PlanetLab [20] testbed.

4.1 Experimental Setup

Our ModelNet experiments make use of 25 2.0 and 2.8-Ghz Pentium-4s running Xeno-Linux 2.4.27 and interconnected by 100-Mbps and 1-Gbps Ethernet switches. In the experiments presented here, we multiplex one hundred logical end nodes running our download applications across the 25 Linux nodes (4 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 are run on a fully interconnected mesh topology, where each pair of overlay participants are directly connected. While admittedly not representative of actual Internet topologies, it allows 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 Mbps, while the nominal bandwidth on the core links is 2 Mbps. In an attempt to model the wide-area environment [21], we configure ModelNet to randomly drop packets on the core links with probability ranging from 0 to 3 percent. The loss rate on each link is chosen uniformly at random and fixed for the duration of an experiment. To approximate the latencies in the Internet [7, 21], we set the propagation delay on the core links uniformly at random between 5 and 200 milliseconds, while the access links have one millisecond delay.

For most of the following sections, we conduct identical experiments in two scenarios: a static bandwidth case and a variable bandwidth case. Our bandwidth-change scenario models changes in the network band-

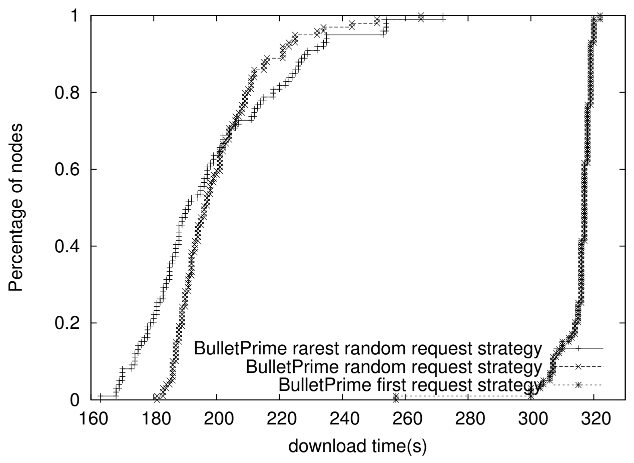

Figure 6: Impact of request strategy on Bullet′ performance while downloading a 100 MB file under random network packet losses.

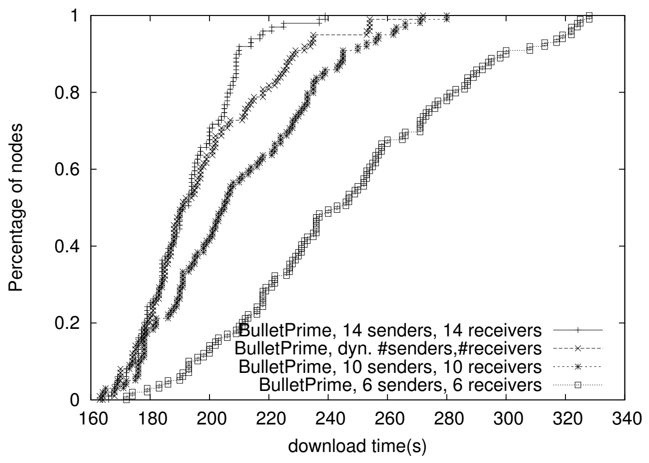

Figure 7: Bullet′ performance under random network packet losses for static peer set sizes of 6, 10, 14, and the dynamic peer set size sizing case while downloading a 100 MB file.

width that correspond to correlated and cumulative decreases in bandwidth from a large set of sources from any vantage point. To effect these changes, we decrease the bandwidth in the core links with a period of 20 seconds. At the beginning of each period, we choose 50 percent of the overlay participants uniformly at random. For each participant selected, we then randomly choose 50 percent of the other overlay participants and decrease the bandwidth on the core links from those nodes to 50 percent of the current value, without affecting the links in the reverse direction. The changes we make are cumulative; i.e., it is possible for an unlucky node pair to have 25% of the original bandwidth after two iterations. We do not alter the physical link loss rates that were chosen during topology generation.

4.2 Overall Performance

We begin by studying how Bullet′ performs overall, using the existing best-of-breed systems as comparison points. For reference, we also calculate the best achievable performance given the overhead of our underlying transport protocols. Figure 4 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. Bullet′'s performance is even better in the dynamic scenario (faster by 32%-70%), shown in Figure 5.

We set the transfer block size to 16 KB in all of our experiments. This value corresponds to BitTorrent's subpiece size of 16KB, and is also shared by the Bullet and SplitStream. For all of our experiments, we make sure that there is enough physical memory on the machines hosting the overlay participants to cache the entire file content in memory. Our goal is to concentrate on distributed algorithm performance and not worry about swapping file blocks to and from the disk. Bullet and SplitStream results are optimistic since we do not perform encoding and decoding of the file. Instead, we set the encoding overhead to 4% and declare the file complete when a node has received enough file blocks.

4.3 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 6 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*

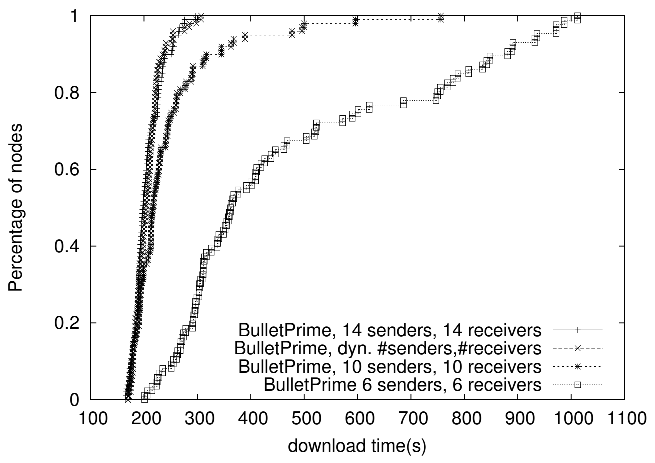

Figure 8: Bullet′ performance under synthetic bandwidth changes and random network packet losses for static peer set sizes of 6, 10, 14, and the dynamic peer set size sizing case while downloading a 100 MB file.

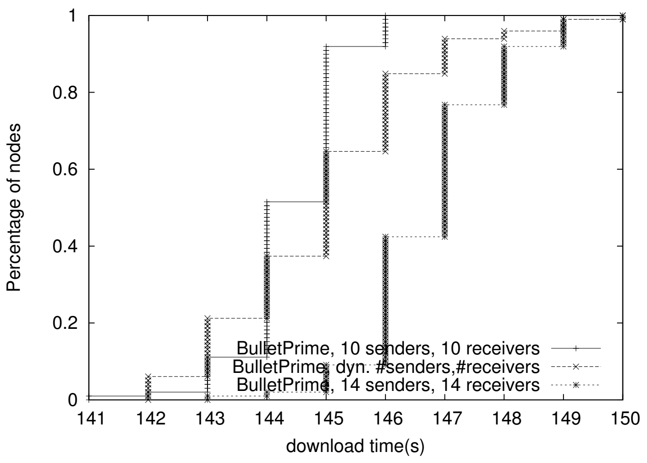

Figure 9: Bullet′ performance in the absence of bandwidth changes and random packet losses for static peer set sizes of 10, 14, and the dynamic peer set size sizing case while downloading a 10 MB file in a topology with constrained access links.

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.

4.4 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 7 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 performance of the system with the number of peers fixed to 14 for 50% of receivers. Under synthetic bandwidth changes (Figure 8), our dynamic approach matches, and sometimes exceeds the performance of static setups.

For our final peering example, we construct a 100 node topology with ample bandwidth in the core (10Mbps, 1ms latency links) with 800 Kbps access links and without random network packet losses. Figure 9 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 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.

4.5 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 use an 8KB block, and configure the Linux kernel to allow large receiver window sizes. In our first topology, there are 25 participants, interconnected with 10Mbps links with 100ms latency. In Figure 10 we show Bullet′'s performance when configured with 3, 6, 9, 15, and 50 per-peer outstanding blocks for up to 5 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

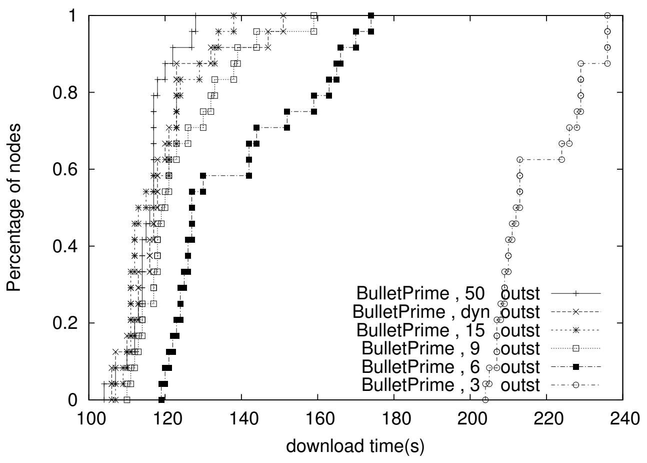

Figure 10: Bullet′ performance with neither bandwidth changes nor random network packet losses for 3, 6, 9, 15, 50 outstanding blocks, and for the dynamic queue sizing case while downloading a 100 MB file

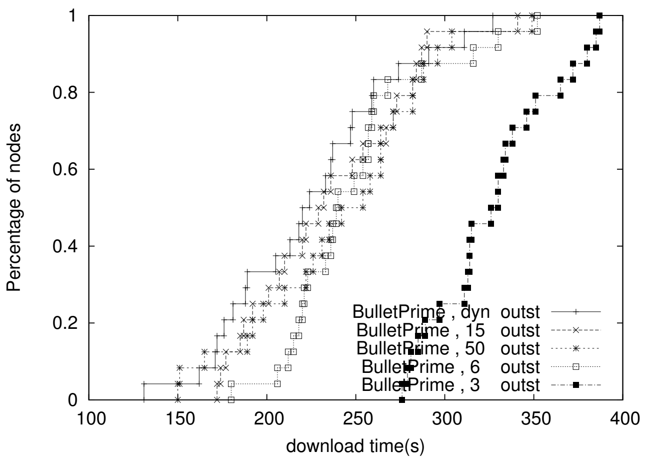

Figure 11: Bullet′ performance under random network packet losses while downloading a 100 MB file. CDF line for 9 blocks omitted for readability.

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 11 illustrates the penalty of requesting more blocks than it is required to saturate the TCP connection. In this experiment, we instruct ModelNet to drop packets uniformly at random with probability ranging between 0 and 1.5 percent on the core links. Due to losses, TCP achieves lower bandwidths, requiring less data in-flight for maximum performance. Under these loss-induced TCP throughput fluctuations, our dynamic approach outperforms all static cases. Figure 12 provides more insight into this case. For this experiment, we have 8 participants including the source, with 6 nodes receiving data from the source and reconciling among each other over 10Mbps, 1ms latency links. We use 8KB blocks and disable peer management. The 8th node is only downloading from the 6 peers over dedicated 5Mbps, 100ms latency links. Every 25 seconds, we choose another peer from these 6, and reduce the bandwidth on its link toward the 8th node to 100Kbps. These cascading bandwidth changes are cumulative, i.e. in the end, the 8th node will have only 100Kbps links from its peers. Our dynamic scheme outperforms all fixed sizing choices for the slowest, 8th node, by 7 to 22 percent. Placing too many requests on a connection to a node that suddenly slows down forces the receiver to wait too long for these blocks to arrive, instead of retrieving them from some other peer.

4.6 Potential Benefits of Source Encoding

The purpose of this section is to quantify the potential benefits of encoding the file at the source. Towards this end, Figure 13 shows the average block inter-arrival times among the 99 receivers, while downloading a 100MB file. To improve the readability of the graph, we do not show the maximum block inter-arrival times, which observe a similar trend. A system that has a pronounced “last-block” problem would exhibit a sharp increase in the block inter-arrival time for the last several blocks. To quantify the potential benefits of encoding, we first compute the overall average block inter-arrival time t_b. We then consider the last twenty blocks and calculate the cumulative overage of the average block inter-arrival time over t_b. In this case overage amounts to 8.38 seconds. We contrast this value to the potential increase in the download time due to a fixed 4 percent encoding overhead of 7.60 seconds, 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 [8].

4.7 PlanetLab

This section contains results from the deployment of Bullet′ over the PlanetLab [20] wide-area network

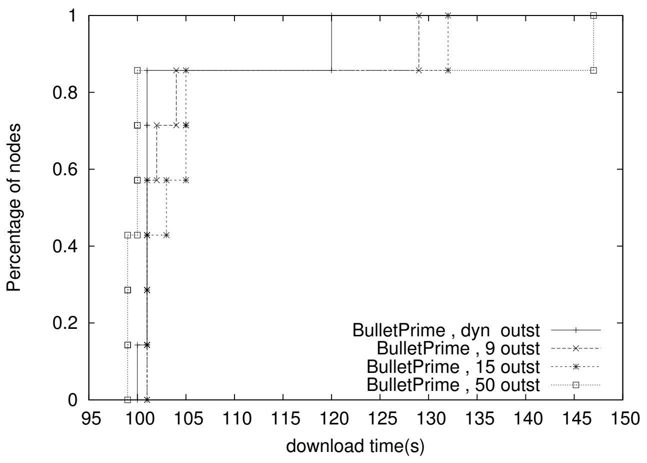

Figure 12: Bullet′ performance under synthetic bandwidth changes while downloading a 100 MB file. CDF lines for 3 and 6 omitted because the 8th node takes considerably more time to finish in these cases.

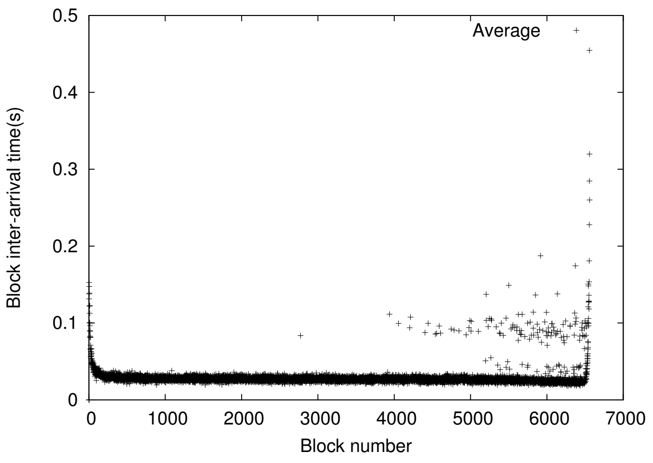

Figure 13: Block inter-arrival times for a 100 MB file under random network packet losses in the absence of bandwidth changes. The block numbers on the X axis correspond to the order in which nodes retrieve blocks, not the actual block numbers.

testbed. 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 100KB block size. Bullet and SplitStream were not performing the file encoding/decoding; instead we mark the downloads as successful when a required number of distinct file blocks is successfully received, including fixed 4 percent overhead that an actual encoding scheme would incur. We see in Figure 14 that Bullet′ consistently outperforms other systems in the wide-area. For example, the slowest Bullet′ node completes the 50MB download approximately 400 seconds sooner than BitTorrent's slowest downloader.

4.8 Shotgun: a Rapid Synchronization Tool

This section presents Shotgun, a set of extensions to the popular `rsync` [27] tool to enable clients to synchronize their state with a centralized server orders of magnitude more quickly than previously possible. At a high-level, Shotgun works by wrapping appropriate interfaces to Bullet′ around `rsync`. Shotgun is useful, for example, for a user who wants to run an experiment on a set of PlanetLab[20] nodes. Because each node only has local storage, the user must somehow copy (using `scp` or `rsync`) her program files to each node individually. Each time she makes any change to her program, she must re-copy the updated files to all the nodes she is using. If the experiment spans hundreds of nodes, then copying the files requires opening hundreds of ssh connections, all of which compete for bandwidth.

To use Shotgun, the user simply starts `shotgund`, the Shotgun multicast daemon, on each of his nodes. To distribute an update, the user runs `shotgun_sync`, providing as arguments a path to the new software image, a path to the current software image, and the host name of the source of the Shotgun multicast tree (this can be the local machine). Next, `shotgun_sync` runs `rsync` in batch mode between the two software image paths, generating a set of logs describing the differences and records the version numbers of the old and new files. `shotgun_sync` then archives the logs into a single `tar` file and sends it to the source, which then rapidly disseminates it to all the clients using the multicast overlay. Each client's `shotgund` will download the update, and then invoke `shotgun_sync` locally to apply the update if the update's version is greater than the node's current version.

Running an `rsync` instance for each target node overloads the source node's CPU with a large number of `rsync` processes all competing for the disk, CPU, and bandwidth. Therefore, we have attempted to experimentally determine the number of simultaneous rsync processes that give the optimal overall performance using the staggered approach. Figure 15 shows that Shotgun outperforms `rsync` (1, 2, 4, 8, and 16 parallel instances) by two orders of magnitude. Another interesting result from this graph is that the constraining factor for PlanetLab nodes is the disk, not the network. On average, most nodes spent twice as much time replaying the *rsync* logs locally then they spent downloading the data.

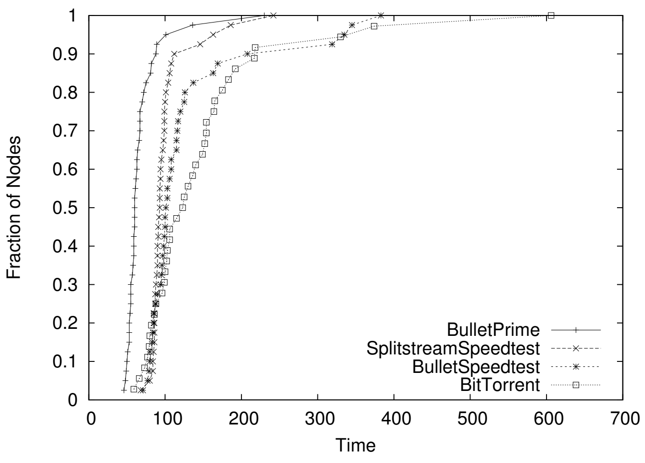

Figure 14: Comparison of Bullet′ to Bullet, BitTorrent, and SplitStream for 50MB file size on PlanetLab.

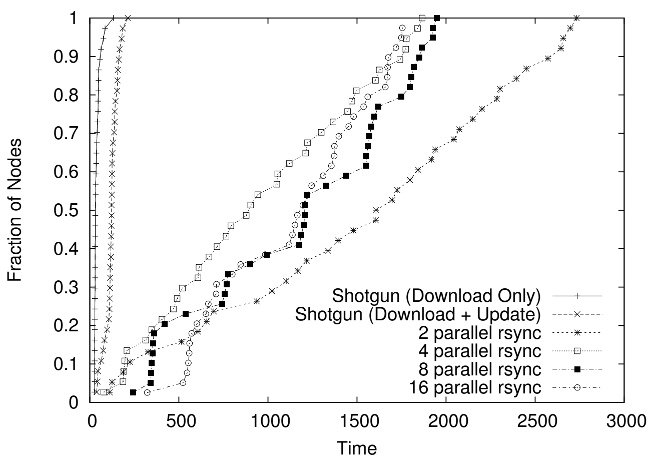

Figure 15: The aggregate completion time for varying number of parallel rsync processes for an update with 24MB of deltas on 40 PlanetLab nodes.

5 Related Work

Overcast [10] constructs a bandwidth-optimized overlay tree among dedicated infrastructure nodes. An incoming node joins at the source and probes for acceptable bandwidth under one if its siblings to descend down the tree. A node's bandwidth and reliability is determined by characteristics of the network between itself and its parent and is lower than the performance and reliability provided by an overlay mesh.

Narada [9] constructs a mesh based on a *k-spanner* graph, and uses bandwidth and latency probing to improve the quality of the mesh. It then employs standard routing algorithms to compute per-source forwarding trees, in which a node's performance is still defined by connectivity to its parent. In addition, the group membership protocol limits the scale of the system.

Snoeren et al. [26] use an overlay mesh to send XML-encoded data. The mesh is structured by enforcing k parents for each participant. The emphasis of this primarily push-based system is on reliability and timely delivery, so nodes flood the data over the mesh.

In FastReplica [6] file distribution system, the source of a file divides the file into n blocks, sends a different block to each of the receivers, and then instructs the receivers to retrieve the blocks from each other. Since the system treats every node pair equally, overall performance is determined by the transfer rate of the slowest end-to-end connection.

BitTorrent [3] 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. BitTorrent enforces fairness via a tit-for-tat 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. In addition, tracker presents a single point of failure and limits the system scalability.

SplitStream [5] aims to construct an interior-node disjoint forest of k Scribe [24] trees on top of a scalable peer-to-peer substrate [23]. The content is split into k *stripes*, each of which is pushed along one of the trees. This system takes into account physical inbound and outbound link bandwidth of a node when determining the number of stripes a node can forward. It does not, however, consider the overall end-to-end performance of an overlay path. Therefore, system throughput might be decreased by congestion that does not occur on the access links.

In CoopNet [18], the source of the multimedia content computes locally random or node-disjoint forests of trees in a manner similar to SplitStream. The trees are primarily designed for resilience to node departures, with network efficiency as the second goal.

Slurpie [25] 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 webserver 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.

Avalanche [8] is a file distribution system that uses network coding [1]. The authors demonstrate 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. There are no live evaluation results of the system, but it is likely Avalanche will benefit from the techniques outlined in this paper. 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.

CoDeploy [19] builds upon CoDeeN, an existing HTTP Content Distribution Network (CDN), to support dissemination of large files. In contrast, Bullet′ operates without infrastructure support and achieves bandwidth rates (7Mbps on average with a source limited to 10Mbps) that exceed CoDeploy's published results.

Young et al. [29] construct 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 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.

6 Conclusions

We have presented Bullet′, a system for distributing large files across multiple wide-area sites in a wide range dynamic of network conditions. Through a careful evaluation of design space parameters, we have designed Bullet′ to keep its incoming pipe full of useful data with a minimal amount of control overhead from a dynamic set of peers. In the process, we have defined the important aspects of the general data dissemination problem and explored several possibilities within each aspect. Our results validate that each of these tenets of file distribution is an important consideration in building file distribution systems. Our experience also shows that strategies 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, Bullet′ employs adaptive strategies which can adjust over time to self-tune to conditions which will perform well in a much wider range of conditions, and indeed in many scenarios of dynamically changing conditions. Additionally, we have compared Bullet′ with BitTorrent, Bullet and SplitStream. In all cases, Bullet′ outperforms other systems.

Acknowledgments

We would like to thank David Becker for his invaluable help with our ModelNet experiments and Ken Yocum for his comments on our flow control algorithm. In addition, we thank our shepherd Atul Adya and our anonymous reviewers who provided excellent feedback.

References

[1] Rudolf Ahlswede, Ning Cai, Shuo-Yen Robert Li, and Raymond W. Yeung. Network Information Flow. In *IEEE Transactions on Information Theory, vol. 46, no. 4*, 2000.

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

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

[4] John W. Byers, Michael Luby, Michael Mitzenmacher, and Ashutosh Rege. A Digital Fountain Approach to Reliable Distribution of Bulk Data. In *Proceedings of ACM SIGCOMM*, 1998.

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

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

[7] Frank Dabek, Jinyang Li, Emil Sit, Frans Kaashoek, Robert Morris, and Chuck Blake. 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, March 2004.

[8] Christos Gkantsidis and Pablo Rodriguez Rodriguez. Network Coding for Large Scale Content Distribution. In *Proceedings of IEEE Infocom*, 2005.

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

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

[11] Dina Katabi, Mark Handley, and Charles Rohrs. Internet congestion control for high bandwidth-delay product networks. In *Proceedings of ACM SIGCOMM*, August 2002.

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

[13] Dejan Kostić, Adolfo Rodriguez, Jeannie Albrecht, and Amin Vahdat. Bullet: High Bandwidth Data Dissemination Using and Overlay Mesh. In *Proceedings of the 19th ACM Symposium on Operating System Principles*, October 2003.

[14] Maxwell N. Krohn, Michael J. Freedman, and David Mazieres. On-the-Fly Verification of Rateless Erasure Codes for Efficient Content Distribution. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, 2004.

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

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

[17] Petar Maymounkov and David Mazieres. Rateless codes and big downloads. In *Proceedings of the Second International Peer to Peer Symposium (IPTPS)*, March 2003.

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

[19] KyoungSoo Park and Vivek S. Pai. Deploying large file transfer on an http content distribution network. In *Proceedings of the First Workshop on Real, Large Distributed Systems (WORLDS '04)*, 2004.

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

[21] Pinger site-by-month history table. http://www-iepm.slac.stanford.edu/cgi-wrap/pingtable.pl.

[22] Adolfo Rodriguez, Charles Killian, Sooraj Bhat, Dejan Kostić, and Amin Vahdat. 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, March 2004.

[23] Antony Rowstron and Peter Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-Peer Systems. In *Middleware'2001*, November 2001.

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

[25] Rob Sherwood, Ryan Braud, and Bobby Bhattacharjee. Slurpie: A Cooperative Bulk Data Transfer Protocol. In *Proceedings of IEEE INFOCOM*, 2004.

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

[27] Andrew Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, 1999.

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

[29] Anthony Young, Jiang Chen, Zheng Ma, Arvind Krishnamurthy, Larry Peterson, and Randolph Y. Wang. Overlay mesh construction using interleaved spanning trees. In *Proceedings of IEEE INFOCOM*, 2004.