Server-assisted Latency Management for Wide-area Distributed Systems

Wonho Kim[1], KyoungSoo Park[2], and Vivek S. Pai[1]

[1]*Department of Computer Science, Princeton University*
[2]*Department of Electrical Engineering, KAIST*

Abstract

Recently many Internet services employ wide-area platforms to improve the end-user experience in the WAN. To maintain close control over their remote nodes, the wide-area systems require low-latency dissemination of new updates for system configurations, customer requirements, and task lists at runtime. However, we observe that existing data transfer systems focus on resource efficiency for open client populations, rather than focusing on completion latency for a known set of nodes. In examining this problem, we find that optimizing for latency produces strategies radically different from existing systems, and can dramatically reduce latency across a wide range of scenarios.

This paper presents a latency-sensitive file transfer system, Lsync that can be used as synchronization building block for wide-area systems where latency matters. Lsync performs novel node selection, scheduling, and adaptive policy switching that dynamically chooses the best strategy using information available at runtime. Our evaluation results from a PlanetLab deployment show that Lsync outperforms a wide variety of data transfer systems and achieves significantly higher synchronization ratio even under frequent file updates.

1 Introduction

Low-latency data dissemination is essential for coordinating remote nodes in wide-area distributed systems. The systems need to disseminate new data to remote nodes with minimal latency when distributing new configurations, when coordinating task lists among multiple endpoints, or when optimizing system performance under dynamically changing network conditions. All of the scenarios involve *latency-sensitive synchronization*, where the enforced synchronization barrier can limit overall system performance and responsiveness.

The effects of synchronization latency will become increasingly more important as more Internet services leverage distributed platforms to accelerate their applications in the WAN. Recent trends show that many companies employ distributed caching services [4, 18] and WAN optimization appliances [24], or deploy their applications at the Internet edges close to the end users [12] to improve the user experience. These platforms should handle frequently-changing customer requirements and adapt to dynamic network conditions at runtime. For instance, Akamai [2] reports that its management server receives five configuration updates per minute that need to be propagated to remote CDN nodes immediately [26]. If these systems face long synchronization delays, possibilities include service disruption, inconsistent behavior at different replicas visible to end users, or increased application complexity to try to mask such effects.

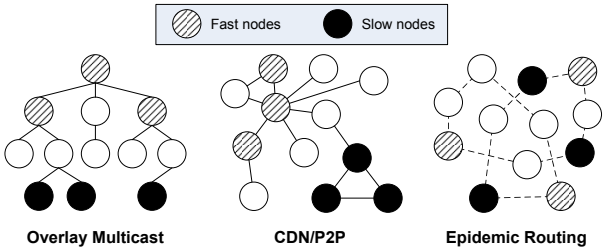


Figure 1: Slow Nodes in Overlay – Peering strategies in scalable one-to-many data transfer systems are not favorable to slow nodes.

The latency is measured as the total completion time of file transfer to all target remote nodes. In wide-area systems, it is usual that a number of nodes experience network performance problems and/or lag far behind up-to-date synchronization state at any given time. We observe these *slow nodes* typically dominate the completion time, which means that managing their tail latency is crucial for latency-sensitive synchronization.

Although numerous systems have been proposed for scalable one-to-many data transfers [8, 10, 16, 17, 19, 21], they largely ignore the latency issue because re-

source efficiency is typically their primary concern for serving an open client population. In an *open client population*, there is no upper bound on the number of clients, so the systems aim to maximize average performance or aggregate throughput in the system.

As a result, those systems are not favorable to slow nodes (Figure 1). For instance, many overlay multicast systems attempt to place well-provisioned *fast nodes* close to the root of the multicast tree while pushing slow nodes down the tree. Likewise, the peering strategies used in CDN/P2P systems make slow nodes have little chance to download from fast nodes. The random gossiping in epidemic routing protocol helps slow nodes peer with fast nodes, but only for a short-term period. These peering strategies not only produce long completion time but make the systems highly vulnerable to changes in slow node's network condition. Despite these disadvantages, it is common that existing services rely on one of the data transfer schemes for coordinating their remote nodes.

In this paper, we explore general file transfer policies for latency-sensitive synchronization with the goal of minimizing completion time in a *closed client population*. This completion time metric drives us to examine new optimization opportunities that may not be advisable for systems with open client populations. In particular, we aggressively use spare bandwidth in the origin server to assist nodes that experience transient/persistent performance problems in the overlay mesh at runtime. The server allocates its bandwidth in a manner favorable to the slow nodes while synchronizing the other nodes through existing overlay mesh. This server-assisted synchronization reduces the tail latency in slow nodes without sacrificing scalable data transfers in the overlay mesh, which drastically improves the completion time and achieves stable file transfer.

For evaluation of our policies, we develop Lsync, a low-latency file transfer system for wide-area distributed systems. Lsync can be used as synchronization building block for wide-area distributed systems where latency matters. Lsync continuously disseminates files in the background, monitoring file changes and choosing the best strategy based on information available at runtime. Lsync is designed to be easily pluggable into existing systems. Users can specify a local directory to be synchronized across remote machines, and give Lsync the information about target remote nodes. Other systems can use Lsync by simply dropping files into the directory monitored by Lsync when the files need latency-sensitive synchronization.

We evaluate Lsync against a wide variety of data transfer systems, including a commercial CDN. Our evaluation results from a PlanetLab [1] deployment show that Lsync can drastically reduce latency compared to existing file transfer systems, often needing only a few seconds for synchronizing hundreds of nodes. When we generate a 1-hour workload similar to the frequent configuration updates in Akamai CDN, Lsync achieves significantly higher synchronization ratio than the alternatives throughout the experiment.

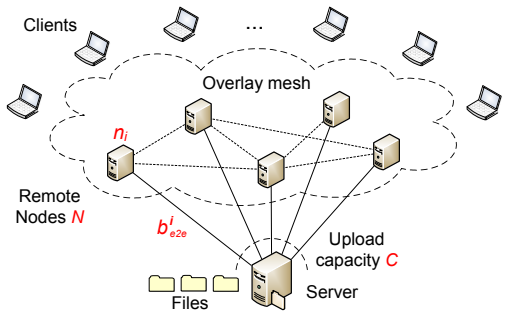


Figure 2: Synchronization Environment – the server has files to transfer to remote nodes with low completion latency. The remote nodes construct an overlay mesh for providing services to external clients.

The rest of this paper is organized into following sections. In Section 2, we describe the model and assumptions for our problem. Based on the model, we discuss how to allocate server's bandwidth to slow nodes in Section 3. In Section 4, we describe how to divide nodes between server and overlay mesh in a way to minimize the total latency. We describe the implementation in Section 5 and evaluate it on PlanetLab in Section 6.

2 Synchronization Environment

In this section, we describe the basic operational model used for Lsync, along with the assumptions we make about its usage.

Operational model We assume one dedicated server that coordinates a set of remote nodes, N, geographically distributed in the WAN, as shown in Figure 2. The management server has an uplink capacity of C, and can communicate with each remote node n_i with bandwidth b^i_{e2e}. The C value is configurable, and represents the maximum bandwidth that can be used for remote synchronization. The b^i_{e2e} values are updated using a history-based adaptation technique, described in Section 4.5, to adjust to variations in available bandwidth. The server knows the amount of new data by detecting file changes in the background as described in Section 5.

Many distributed systems construct an overlay mesh among their remote nodes to use scalable data transfer systems. If an overlay mesh is available and certain conditions are met, Lsync leverages it for fast synchronization. To make Lsync easily pluggable into existing systems, we do not require modification of a given overlay's behavior. Instead, Lsync characterizes the overlay

mesh using a black-box approach to estimate its startup latency. Based on the estimation, Lsync determines how to adjust workload across the server and the overlay.

Target environments This model is appropriate for our target environments, where a large number of nodes are geographically distributed without heavy concentrations in any single datacenter.[1] Our target environments also require that we exclude IP multicast, due to the problems stemming from lack of widespread deployment and coordination across different ASes.

Examples of current systems where our intended approach may be applicable include distributed caching services [4, 18], WAN optimization appliances [24], distributed computation systems [25], edge computing platforms [12], wide-area testbeds (PlanetLab, OneLab), and managed P2P systems [20]. In addition, with most major switch/router vendors announcing support for programmable blades for their next-generation networks, virtually every large network will soon be capable of being a distributed service platform.

Target remote nodes In Lsync, users can specify target remote nodes in various ways. For global updates, users will configure Lsync to optimize the latency for updating all remote nodes. For partial updates, it is possible to select a subset of specific remote nodes to synchronize. Users can also specify a certain fraction of nodes that need to be synchronized fast for satisfying certain consistency models or incremental rollouts of new configurations in the system. We name the parameter *target synchronization ratio* denoted by r, for the rest of the paper. This enables Lsync to use intelligent node selection, which is particularly effective for disseminating frequent updates.

3 Server Bandwidth Allocation

Lsync's file transfer policy combines multiple factors that contribute to the overall time reduction. We quantify the benefits separately, so we describe steps separately in the paper. After adjusting workload across server and overlay, Lsync exploits the server's spare bandwidth to speed up synchronizing the overlay mesh. In this section, we begin by examining the effects of the server's bandwidth allocation policies on completion latency.

Figure 3 shows an example of the timeline when the server transfers files to a set of target remote nodes, $N_{target} \subset N$. The server detects a new file f_{new} at time t_0. Each horizontal bar corresponds to a remote node, $n_i \in N_{target}$, that has variable-sized unsynchronized data f^i_{prev} remaining from previous transfers. The areas of f^i_{prev} and f_{new} represent the sizes of the files, $|f^i_{prev}|$ and

[1]The intra-datacenter bandwidths are sufficient to make the synchronization latency less of an issue in that environment.

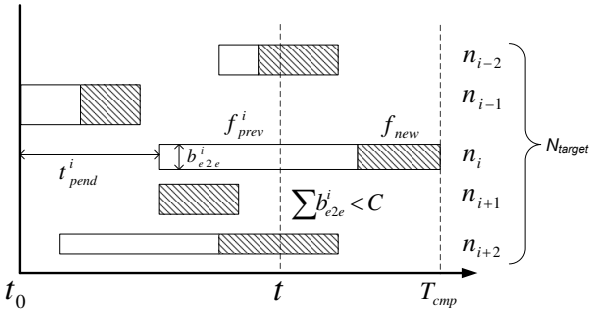


Figure 3: Timeline of Synchronization – The completion time T_{cmp} is determined by the node with the latest finish time among the target nodes N_{target}.

$|f_{new}|$, respectively. The height of the bar, b^i_{e2e}, is the end-to-end bandwidth from the server to n_i that starts its transfer after a pending time t^i_{pend}. At any given time t, the sum of the heights of the bars should not exceed the server's upload capacity C in order to avoid self-congestion and associated problems stemming from the server overload [3].

The completion time, T_{cmp}, is calculated as

$$T_{cmp} = t_0 + \max_i \left(t^i_{pend} + \frac{|f^i_{prev}| + |f_{new}|}{b^i_{e2e}} \right) \quad (1)$$

for $n_i \in N_{target}$. There are two variables that the server can control transparently to the remote nodes. The server can determine t^i_{pend} that n_i should wait before starting its transfer. The server can also select nodes for N_{target} if a target synchronization ratio is given. From the perspective of the server, controlling these two variables corresponds to *node scheduling* and *node selection* policies in the server, respectively. The basic intuition behind the policies is that we could reduce the latency by giving low t^i_{pend} to slow nodes and carefully selecting nodes for N_{target}. In the following sections, we compare different policies using real measurements on PlanetLab to quantify their effects on latency.

3.1 Node Scheduling

We begin by examining how we schedule transfers when the number of transfers exceeds the outbound capacity of the server. This is the problem of minimizing makespan, which is NP-hard [7]. We compare two basic scheduling heuristics, Fast First and Slow First. The *Fast First* is similar to Shortest Remaining Processing Time (SRPT) scheduling in that the server's resource is allocated to the node who has the shortest expected completion time. SRPT is known to be optimal for minimizing mean response time [6]. The *Slow First* is the opposite of the Fast First policy, and selects the nodes with the longest expected completion time when the server has available bandwidth.

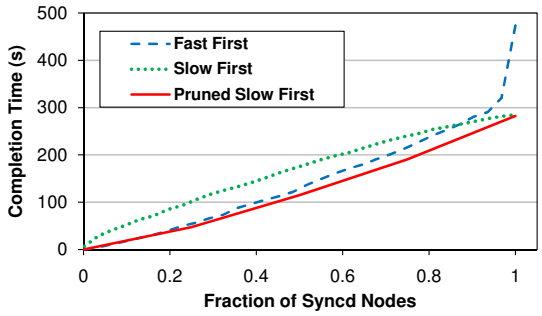


Figure 4: Node Scheduling and Node Selection – Pruned Slow First captures both the initial speed advantage of Fast First, as well as the total overall advantage of Slow First.

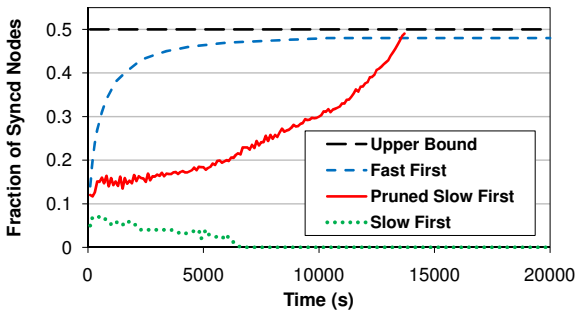


Figure 5: Synchronizing Frequent Updates – While Fast First synchronizes quickly at first, Pruned Slow First actually reaches the upper bound more quickly.

We compare the two schedulers on PlanetLab nodes. We implement the policies in a dedicated server that has 100 Mbps upload capacity. Then we generate two files – f_{new} (1 MB), and f_{prev} (10 MB) that has been in the process of synchronization on the nodes, from 1% to 99% complete. We measure the time to synchronize all live PlanetLab nodes (559 nodes at the time of the experiments) with either of the two scheduling modes enabled.

The result of the measurement is shown in Figure 4. Fast First synchronizes most nodes faster than Slow First, but at high target ratios, Fast First performs much worse. The reason for the difference is somewhat obvious: near the end of the transfers in Fast First, only slow nodes remain, and the server's uplink becomes underutilized. This underutilization occurs for the final 175 seconds in Fast First, but only for 0.5 seconds in Slow First. We also evaluated *Random* scheduling, which selects a random node to allocate available bandwidth. From 10 repeated experiments, we found that Random scheduling yields completion times between Fast First and Slow First, but generally closer to Slow First for all target ratios.

This result implies that slow nodes dominate the completion time in the WAN and that the Slow First scheduling can mask their effects on latency. Another implication of the result is that offline optimization will provide little benefit in the scenario. Note that the server's bandwidth is underutilized only for 0.5 seconds in Slow First. This means that no scheduler can reduce the latency by more than 0.5 seconds in the setting. In addition, our evaluation results show that runtime adaptation has a significant impact on latency, which offline schedulers cannot provide.

3.2 Node Selection

In this section, we examine the effect of node selection. In Lsync, users can specify their requirements in the form of target synchronization ratio, a fraction of nodes that need to be synchronized fast. If the ratio is given, Lsync attempts to find a subset of nodes that could further reduce latency.

We show that integrating node selection with Slow First scheduling can blend the best behaviors of Slow First and Fast First. Given target ratio r, we first sort all nodes in an increasing order of the estimated remaining synchronization time. We then pick $N_s \cdot r$ nodes where N_s denotes the number of nodes in N, and use Slow First scheduling for the selected nodes. The remaining $N_s \cdot (1-r)$ nodes are synchronized using Slow First as well after the selected nodes are finished. As a result, the completion time does not suffer either from the slow synchronization in the beginning (Slow First) or the long tail at the end (Fast First). We name the integrated scheme *Pruned Slow First* for comparison. Figure 4 shows that Pruned Slow First outperforms the other scheduling policies across all target ratios.

We examine a dynamic file update scenario where new files are frequently added to the server, which is common in large-scale distributed services. For an in-depth analysis, we use simulations for the experiment. We generate 2000 nodes with bandwidths drawn from the distribution of the inter-node bandwidths on PlanetLab.

We study how these frequent updates affect the synchronization process. Given information about available resources, we can determine how much change the server can afford to propagate. We define *update rate*, u, as the amount of new content to be synchronized per unit time (one second). Then, $N_s \cdot u$ is the minimum bandwidth required to synchronize all N_s nodes with u rate of change. The upper bound on the achievable synchronization ratio is $\frac{C}{N_s \cdot u}$ where C is the server's upload capacity. C is set to 100 Mbps in the experiment.

Figure 5 shows each policy's performance under frequent updates. As before, the server has two files, 1 MB and 10 MB that are in the process of synchronization, and new files are constantly added to the server with an update rate 100 Kbps. The upper bound is 0.5 in this setting, and no policy can reach beyond this limit.

We see that the completion time drastically changes as we use different policies. In particular, the synchronization ratio of Slow First drops over time and reaches

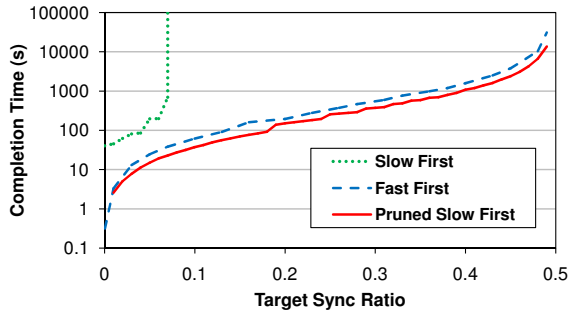


Figure 6: Synchronization Latency for Frequent Updates – While Slow First leads to failure, integrating node selection with the Slow First scheduling reduces latency for all target ratios (y-axis is in log-scale).

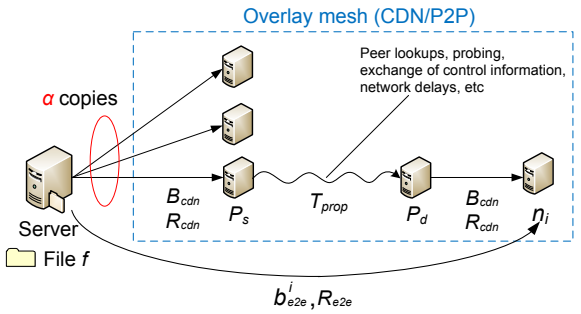


Figure 7: Startup Latency in CDN/P2P – To leverage a given overlay system, Lsync estimates the startup latency for fetching a new file f from the server and propagating to the remote nodes n_i in the overlay.

0 near 6800 seconds because it gives high priority to the nodes that are far behind up-to-date synchronization state. Fast First synchronizes nodes quickly in the beginning, but asymptotically approaches the upper bound since the remaining slower nodes make little forward progress given the rate of change.

To examine the performance of Pruned Slow First, we set r to 0.49, which is slightly below the upper bound 0.5 in this setting. In Figure 5, Pruned Slow First is worse than Fast First in the beginning, but it reaches its target ratio much earlier than the other policies. Pruned Slow First reduces the completion time by 56% compared with Fast First, showing that the best policy can provide significant latency gains, while the worst policy, Slow First, actually leads to failure in this scenario. In Figure 6, we show the completion time of each policy for a range of feasible r values. The Pruned Slow First outperforms the other policies for every target ratio (the y-axis is in log-scale).

We showed that the Slow First scheduling helps reduce completion latency, and integrating node selection with the Slow First scheduling can further reduce latency particularly when handling frequent updates. Based on the observations, we extend our discussion to examine how to leverage overlay mesh for latency-sensitive synchronization in the next section.

4 Leveraging Overlay Mesh

With a better understanding of the bandwidth allocation policies in the server, we focus on understanding how to leverage scalable one-to-many data transfer systems which we collectively name CDN/P2P systems for the rest of the paper. Many large-scale distributed services construct an overlay mesh for scalable data transfer to external clients, and often use the overlay mesh for internal data dissemination to remote nodes as well [26]. In this section, we explore how to leverage the overlay mesh to reduce synchronization latency without changing its behaviors.

4.1 Startup Latency in Overlay Mesh

To leverage a given overlay mesh, Lsync needs to predict *startup latency* for distributing a new file that is not cached on the remote nodes in the overlay. However, it is difficult to predict the accurate latency since CDN/P2P systems typically have diverse peering strategies and dynamic routing mechanisms. To allow easy integration with existing systems, Lsync uses a black-box approach to characterizing a given overlay mesh to estimate its startup latency.

Figure 7 presents a general model showing how a new file f in the server is propagated to a remote node n_i via overlay mesh. P_s is a peer node contacting the server to fetch f, and P_d is a peer node that n_i contacts. If f is already cached in P_d, n_i will receive it directly from P_d. However, in latency-sensitive synchronization, all target nodes attempt to fetch f as soon as it is available in the server. In case f should be fetched from the server, the CDN/P2P system selects node P_s to contact the server. Depending on the system's configuration, multiple copies of the file can be fetched into the overlay mesh. The fetched file is propagated from P_s to P_d possibly through some intermediate overlay nodes, and delivered to n_i. T_{prop} is the propagation delay from P_s to P_d. In addition to the network delay between peer nodes, T_{prop} also includes other overheads such as peer lookups and exchange of control messages, which are system-specific. The bandwidth and RTT between CDN/P2P peer nodes are B_{cdn} and R_{cdn} respectively.

For the simplicity of the model, we begin with an ideal assumption that nodes in CDN/P2P are uniformly distributed, and thus the bandwidth and RTT between neighboring nodes are constants, B_{cdn} and R_{cdn}. However, we will adjust the parameter values later to account for their variations on real deployment in the WAN. This model is not tied to a particular CDN/P2P system or any specific algorithms such as peer selection and request redirection. We apply the model to different types of deployed CDN/P2P systems in Section 6.2. We show

that the model captures the salient characteristics of these systems, and that Lsync can adjust its transfers to utilize each of these systems.

4.2 Completion Time Estimation

To schedule workload across server and overlay mesh, Lsync first estimates the expected completion time in the overlay mesh. The *setup cost*, δ, measures the first-byte latency for fetching newly-created content through the overlay mesh. Specifically, δ is defined as

$$\delta = 2 \cdot R_{cdn} + T_{prop} \tag{2}$$

The overall overlay completion time, T_{cdn}, can be calculated as follows: To distribute f to n_i, the server informs n_i of the new content availability, taking time R_{e2e}. Then, n_i contacts P_d, and starts receiving the file with delay δ. Delivering the entire file to n_i with bandwidth B_{cdn} requires time $\frac{f_s}{B_{cdn}}$ where f_s denotes the size of f. The total time, T_{cdn} is then the sum,

$$T_{cdn} = R_{e2e} + \delta + \frac{f_s}{B_{cdn}} \tag{3}$$

Note that T_{cdn} does not depend on r because all target nodes fetch f simultaneously via the overlay. In comparison, the end-to-end completion time, $T_{e2e}(r)$, for transferring f to remote nodes using Pruned Slow First is

$$T_{e2e}(r) = R_{e2e} + \frac{N_s \cdot r \cdot f_s}{C} \tag{4}$$

where C is the server's upload capacity.

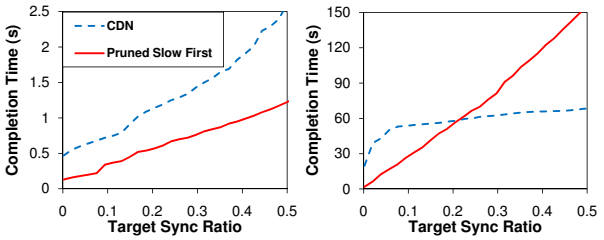
4.3 Selective Use of Overlay Mesh

If a given CDN/P2P system has high startup latency, it will outperform end-to-end transfers only when its bandwidth efficiency can outweigh the cost. After the server estimates T_{cdn} and $T_{e2e}(r)$, the server can dynamically choose between end-to-end transfers and the overlay mesh to get better latency. In this section, we examine the conditions that such a selective use of overlay mesh can provide benefits in a real deployment.

To get a more accurate estimation of the completion time, we extend T_{cdn} in (3) to reflect the fact that the bandwidth distribution of a CDN/P2P system is not uniform. Rather than modeling each node's bandwidth separately, we use the minimum bandwidth value for the top $N_s \cdot r$ nodes, yielding

$$T_{cdn}(r) = R_{e2e} + \delta^r + \frac{f_s}{B^r_{cdn}} \tag{5}$$

B^r_{cdn} is the $N_s \cdot r$ th largest B_{cdn}, and δ^r is the $N_s \cdot r$ th smallest setup cost. As we increase r, by definition, B^r_{cdn} will monotonically decrease, and δ^r will monotonically increase.



(a) Small file (5 KB) (b) Large file (5 MB)

Figure 8: End-to-End Connections vs. Overlay Mesh – For small file, the latency of overlay mesh is hampered by the long setup time, but its efficient bandwidth usage outweighs the cost for large file.

Now we can compare $T_{cdn}(r)$ with $T_{e2e}(r)$ and then use either end-to-end connections or the overlay mesh as appropriate. The decision process can be formally defined as following. The server uses end-to-end connections when either of the following conditions is met.

$$f_s < \delta^r \cdot \frac{C \cdot B^r_{cdn}}{N_s \cdot r \cdot B^r_{cdn} - C} \tag{6}$$

$$B^r_{cdn} < \frac{C}{N_s \cdot r} \tag{7}$$

From the above test conditions, we can draw the following general guidelines. Using end-to-end connections is better when (1) the file size f_s is small, (2) the overlay setup cost δ^r is large, (3) the server's upload capacity C is high, (4) the target synchronization ratio r is low, (5) the client population N_s is small, or (6) the overlay bandwidth B^r_{cdn} is significantly smaller than the server's bandwidth.

We examine the potential benefits of the scheme by comparing end-to-end transfers with CoBlitz CDN [19] on PlanetLab. Beyond PlanetLab, CoBlitz has been used in a number of commercial trial services [11], and we believe CoBlitz represents one of the typical CDN services currently available. For end-to-end transfers, we use Pruned Slow First policy for allocating the server's bandwidth.

Figure 8 shows the latency for synchronizing a small file (5 KB) and a large file (5 MB). For a small file, using end-to-end transfers shows better performance than the CDN because the completion time of the CDN is hampered by its setup cost. When transferring a large file, the CDN shows better performance since its efficient bandwidth usage outweighs the setup cost in the CDN.

Wide-area systems typically disseminate files of varying sizes. For instance, Akamai management server has file transfers spanning 1 KB to 100 MB. The measurement results in Figure 8 imply that Lsync will need different strategies for different file sizes to address the tradeoffs between startup latency and bandwidth efficiency of the overlay mesh.

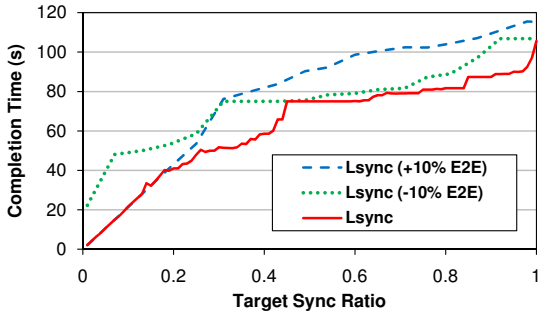


Figure 9: Optimality of the Division – The split between overlay and E2E is not improved by moving some nodes to the other mechanism, suggesting that Lsync's split is close to optimal.

4.4 Using Spare Bandwidth in Server

When nodes are being served by the overlay mesh, the origin server load is greatly reduced, leading to spare bandwidth that can then be used to serve some nodes bypassing the overlay. Given r, Lsync divides remote nodes into two groups. One group directly contacts the origin server, and the other group downloads the file via the overlay mesh. Lsync adds nodes with the worst overlay performance to the end-to-end group until the expected end-to-end completion time matches the overlay completion time.

More formally, Lsync calculates r_{e2e}, the ratio of nodes to place in the end-to-end group. After r_{e2e} is determined, Lsync selects the $N_s \cdot (r - r_{e2e})$ nodes having the fastest overlay connections. From (5), the overlay completion time is estimated as $T_{cdn}(r - r_{e2e})$. The remaining $N_s \cdot r_{e2e}$ nodes will use end-to-end connections. To estimate the spare bandwidth in the server, we consider *CDN load factor* α, which represents how many copies are fetched from the origin server into the overlay mesh. Different systems have different load factors, and the CoBlitz fetches 9 copies from the origin server. As the origin server is supposed to send α copies of f to the overlay group, Lsync should reserve B_{load} bandwidth which is $\frac{\alpha \cdot f_s}{T_{cdn}(r-r_{e2e})}$ for overlay traffic, yielding a value of $(C - B_{load})$ for the server's spare bandwidth. Using this spare bandwidth, the end-to-end completion time is

$$T_{e2e}(r_{e2e}) = R_{e2e} + \frac{N_s \cdot r_{e2e} \cdot f_s}{C - B_{load}} \qquad (8)$$

Then, Lsync calculates r_{e2e} that makes the two groups complete at the same time.

We simulate synchronizing PlanetLab nodes using the bandwidths and setup costs measured in all PlanetLab nodes. Figure 9 shows a sensitivity analysis of r_{e2e}, with two other simulations for slightly higher and lower values of r_{e2e}, in sending a 5 MB file. Lsync outperforms both for all target ratios, suggesting that it is choosing the optimal balance of the two groups.

4.5 Adaptive Switching in Remote Nodes

To mitigate the effect of real-world bandwidth fluctuations, we add a dynamic adaptation technique to Lsync. The main observation behind the technique is that minor variations in performance do not matter for most nodes, since most nodes will not be the bottleneck nodes in the transfer. However, when a node that is close to being the slowest in the overlay group becomes slower, it risks becoming the bottleneck during file transfer. The lower bound on the overlay bandwidth is $B_{cdn}^{r-r_{e2e}}$.

When the origin server informs the remote nodes of new content availability, the server sends $B_{cdn}^{r-r_{e2e}}$ and $T_{cdn}(r - r_{e2e})$. After $T_{cdn}(r - r_{e2e})$ passes, if a node is not finished, it compares the current overlay performance with $B_{cdn}^{r-r_{e2e}}$. If the current performance is significantly lower than the expected lower bound, the node stops downloading from the overlay mesh, and directly goes to the origin server to download the remaining data of the file. In our evaluation, we configure the node to switch to the origin server when its overlay performance drops below 75% of its expected value. Our evaluation results show that using adaptive switching improves the completion time while lowering variations.

5 Implementation

Lsync is a daemon that performs two functions – in one setting, it operates in the background on the server to detect file changes, manage histories, and plan the synchronization process. In its second setting, it runs on all remote nodes to coordinate the synchronization process. Both modes of operation are implemented in the same binary, which is created from 6,586 lines of C code. Lsync daemon implements all the techniques described in the previous sections.

Since Lsync is intended to be easily deployable, it operates entirely in user space. Using Linux's inotify mechanism, the daemon specifies files and directories that are to be watched for changes – any change results in an event being generated, which eliminates the need to constantly poll all files for changes. When the Lsync daemon starts, it performs a per-chunk checksum of all files in all of the directories it has been told to watch using Rabin's fingerprinting algorithm [23] and SHA-1 hashes. Once Lsync is told a file has been changed, it recomputes the checksums to determine what parts of the file have been changed. If new files are created, Lsync receives a notification that the directory itself has changed, and the directory is searched to see if files have been created or deleted.

Once Lsync detects changes, it writes the changes into a log file, along with other identifying information. This log file is sent to the chosen remote Lsync daemons, either by direct transfer, or by copying the log file to a Web-

Systems	δ^r	B^r_{cdn}	Division Ratio	
			E2E	Overlay
CoBlitz	1.4	1.2	0.24	0.76
Coral	7.6	0.6	0.52	0.48
BitTorrent	29.7	4.7	0.30	0.70

Table 1: Division of Nodes between E2E and overlay mesh. r is 0.5, and file size is 5 MB. We also tested small files (up to 30 KB), but E2E outperformed all these systems. δ^r is in seconds, and B^r_{cdn} is in Mbps.

accessible directory and informing the remote daemons to grab the file using a CDN/P2P system. Once the remote daemon receives a log file, it applies the necessary changes to its local copies of the file.

6 Evaluation

In this section, we evaluate Lsync and its underlying policies on PlanetLab. Each experiment is repeated 10 times with each setting, and 95^{th} percentile confidence intervals are provided where appropriate.

6.1 Settings

We deploy Lsync on all live PlanetLab nodes (528 nodes at the time of our experiments), and run a dedicated origin server with 100 Mbps outbound capacity. The server has a 2.4 GHz dual-core Intel processor with 4 MB cache, and runs Apache 2.2.6 on Linux 2.6.23. We measure latency for a set of target synchronization ratios including 0.05, 0.25, 0.5, 0.75, and 0.98. We use a maximum synchronization level of 0.98 to account for nodes that may become unreachable during our experiments. Lsync attempts to achieve the given target ratio fast while leaving other available nodes synchronized via overlay in the background.

6.2 Startup Latency in CDN/P2P Systems

CDN/P2P designers typically expect that the steady state of the CDN/P2P system is that the content is already pulled from the origin, and is being served to clients over a much longer lifetime with high cache hit ratio. However, for synchronization, remote nodes typically request changes that are not in their overlay mesh. Therefore, Lsync monitors the performance of first fetching content from the origin server, which was not a major issue in existing CDN/P2P systems.

Using our black-box model described in Section 4.1, we measured parameters, δ^r and B^r_{cdn}, for two running CDN systems and one P2P system that we deploy on PlanetLab. Table 1 shows the setup costs and bandwidths of the three systems. Each system was characterized by simply fetching new content from the remote nodes, and measuring each node's first-byte latency and bandwidth. The three systems show interesting differences in behavior. BitTorrent spends more time getting the content initially, but then has higher bandwidth. The CDN systems show relatively low setup costs because they were designed for delivering web objects to clients rather than sharing large files.

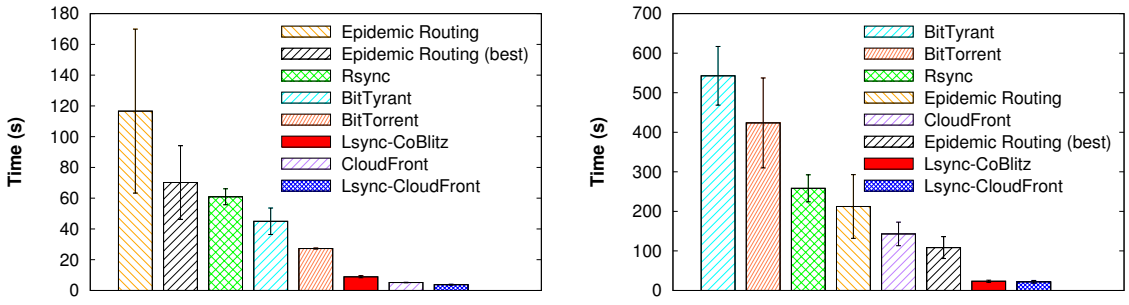
The table also shows how Lsync would allocate nodes between overlay transfers and end-to-end transfers for a synchronization level of 0.5. For small file transfers, Lsync opts to use end-to-end connections rather than any of the systems. For larger transfers, the fraction assigned to direct end-to-end transfers is determined by the tradeoff between latency and bandwidth. For CoBlitz (with the lowest latency) and BitTorrent (with the highest bandwidth), Lsync assigns most nodes to the overlay group. For Coral, with slightly higher latency and lower bandwidth, Lsync assigns nodes roughly equally between overlay and E2E. We select CoBlitz for the rest of our experiments, mostly due to its low latency.

6.3 Comparison with Other Systems

We compare Lsync with different types of data transfer systems. We transfer our CoBlitz web proxy executable file (600 KB) to all PlanetLab nodes using each of the systems, and measure completion times (Figure 10). Each system is designed for robust broadcast (epidemic routing), simple cloning (Rsync), bandwidth efficiency (CDN), and high throughput and fairness (P2P systems). We evaluate the performance of the systems when they are used for latency-sensitive synchronization in the WAN.

Rsync *Rsync* [27] is an end-to-end file synchronization tool widely used for cloning files across remote machines. It uses delta encoding to minimize the amount of transferred data for changes in existing files. In this experiment, however, the server synchronizes a newly generated file not available in remote nodes, so the amount of the transferred data is the same as in the other tested systems. The Rsync server relies only on end-to-end connections to remote nodes for synchronizing the file, and does not use any policies in allocating server's bandwidth. Therefore, the result of Rsync represents the performance of end-to-end transfers with no policy applied.

P2P systems We use *BitTorrent* [9] and *BitTyrant* [21] as examples of P2P systems. Although BitTorrent has a high setup cost (Table 1), it performs better than end-to-end copying tools (Rsync) for target ratio of 0.5 because the majority of nodes have high throughput. However, BitTorrent ends up with very long completion time (424 seconds) and high variations (standard deviation 193) for the target ratio of 0.98. This is because, with BitTorrent, slow nodes have little chance to download from peer nodes with good network conditions, which makes the



(a) Target Sync Ratio 0.50

(b) Target Sync Ratio 0.98

Figure 10: Comparison with Other Systems – We compare Lsync with various data transfer systems in terms of the latency for synchronizing CoBlitz web proxy executable file (600 KB).

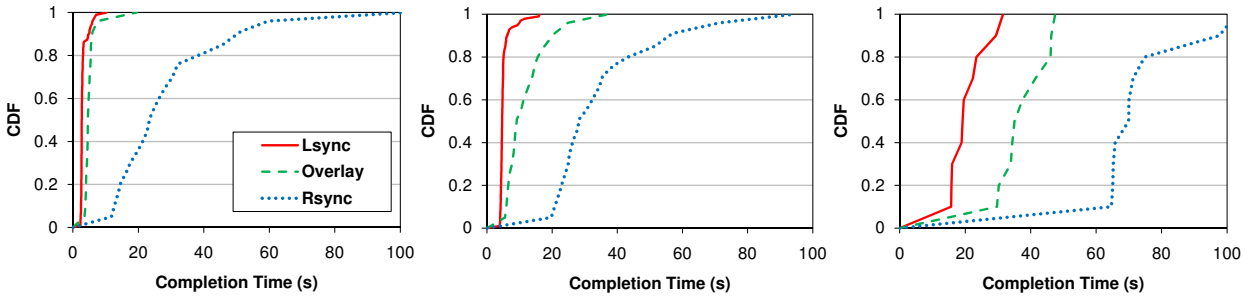
slow nodes finish more slowly than in other systems. We see the similar trend with BitTyrant.

Epidemic routing We also implemented *epidemic routing* (or gossip) protocol [8] to measure its latency in the WAN. In the protocol, each node picks a random subset of remote nodes periodically, and exchanges file chunks. For a conservative evaluation, we assume that every node has the full membership information to avoid membership management, which is known to be the main overhead of the protocol [15]. There are two key parameters in the gossip protocol: how often the nodes perform gossip (*step interval*) and how many peers to gossip with (*fanout*). These two parameters directly impact on the protocol's performance, but it is hard to tune them because of their sensitivity to network conditions. To find the best configuration for our experiments, we tried a range of values for step interval (0.5, 1, 5, and 10 seconds) and fanout (1, 5, 10, 50, and 100 nodes). Then we picked a configuration that generated the shortest latency. Since we use our own implementation of the protocol, we first compared our implementation with published performance results of the protocol in a similar setting. CREW [13] used the gossip protocol for rapid file dissemination in the WAN and outperformed Bullet [17] and SplitStream [10] in terms of completion time in the evaluation. Our implementation showed a comparable latency (141 seconds) to CREW's result (200 seconds) under the same setting (60 nodes, 600 KB file, 200 Kbps bandwidth).[2]

"Epidemic Routing (best)" in Figure 10 represents the results with the best configuration that we found, (1 second interval and 10 fanout), and "Epidemic Routing" shows the average performance of all configurations that we tested (excluding cases with 30 minutes timeout). The result with the tuned configuration always outperforms the average case. Both of the results show interesting patterns in completion time. The protocol works worse than both Rsync and P2P systems at low target ratio because file chunks are disseminated only during gossip rounds. The file data is disseminated relatively slow in the beginning. However, the gossip protocol outperforms the other systems at target ratio of 0.98. Unlike in P2P systems, the slow nodes have better chances to peer with fast nodes during file transfer. As a result, they do not become bottleneck in overall completion time. The random peering policy in the gossip protocol helps slow nodes to catch up with other nodes, but the protocol is typically more optimized for robust dissemination than latency.

Commercial CDNs Since our CDN is deployed on PlanetLab, one may think its performance is adversely affected by some of overloaded PlanetLab nodes. We measure synchronization latency using a commercial CDN, Amazon CloudFront [4]. CloudFront is faster than the other systems for the low target ratio due to its well-provisioned CDN nodes. However, we observe that some nodes are still slow in fetching new file from the CDN. Specifically, 11% of PlanetLab nodes spend more than 20 seconds downloading the file from CloudFront, and the slowest nodes are in Egypt, Tunisia, Argentina, and Australia. As the file is not cached in the CDN, the slow nodes should fetch the file possibly through multiple intermediate nodes in the overlay. This internal behavior depends on system-specific routing mechanisms, which is not visible outside of the overlay. We implemented another version of Lsync, Lsync-CloudFront, which incorporates CloudFront as its underlying overlay mesh. Lsync-CloudFront estimates the overlay's startup latency, and focuses the server's resource on the bottleneck nodes at runtime. We find that the slow nodes can be served better from the server than via the well-provisioned overlay for uncached files, leading to the best performance among the tested systems.

[2]As the topology is not specified, we randomly selected 60 PlanetLab nodes, and averaged over 10 repeated experiments. We used a user-level traffic shaper, Trickle [14], for setting bandwidth on the selected PlanetLab nodes.



(a) 10 KB (b) 100 KB (c) 1 MB

Figure 11: Distribution of Completion Times – For all file sizes, Lsync outperforms the other systems because Lsync adjusts its file transfer policies based on file size as well as network conditions.

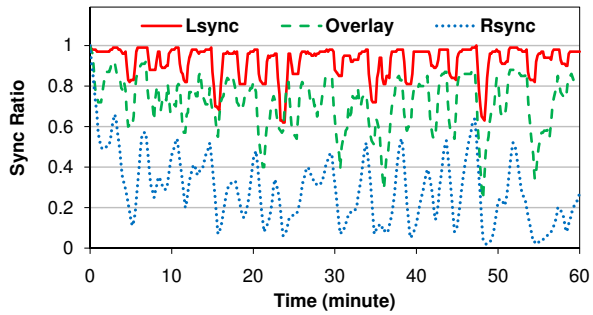


Figure 12: Frequently Added Files – Lsync makes most nodes fully synchronized during the entire period of the experiment.

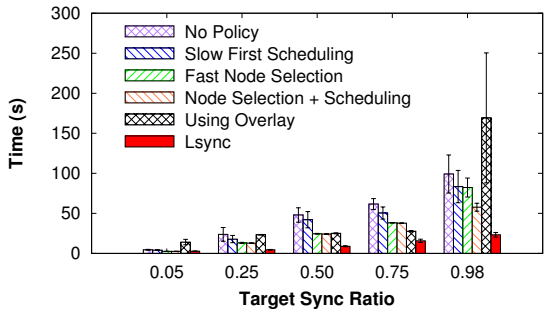


Figure 13: Lsync Contributing Factors – We see that each component in Lsync contributes to the overall time reduction.

6.4 Frequently Added Files

In large-scale distributed services, file updates occur frequently, and new files can be added to the server before the previous updates are fully synchronized across nodes. To evaluate Lsync under frequent updates, we generate a 1-hour workload based on the reported workload of Akamai CDN's configuration updates [26]. A new file is added to the server every 10 seconds, and the file size is drawn from the reported distribution in Akamai. We compare Lsync with CoBlitz and Rsync. The two systems represent alternative approaches: using overlay mesh only (CoBlitz) and using end-to-end transfers only (Rsync).

When a new file is added, we measure how many remote nodes are fully synchronized with all previous files and compute the average of the synchronization ratios over one minute. Figure 12 shows the results over the tested 1-hour period. The ratio temporarily drops for several large file transfers, but Lsync makes 90% of nodes remain fully synchronized for 72% of the tested period while the other approaches do not reach the synchronization ratio 0.9. Figure 11 shows the distributions of the completion latency for 10 KB, 100 KB, and 1 MB files.

6.5 Lsync Contributing Factors

Lsync combines various techniques discussed in the paper, including node scheduling, node selection, workload division across server and overlay, and adaptive switching in remote nodes. We examine the contribution of each factor individually. We transfer CoBlitz web proxy executable file (600 KB) to all remote nodes as before. The results of these tests are shown in Figure 13. Variations of Lsync are shown with no scheduling or node selection (No Policy), with only node scheduling (Slow First Scheduling), with only node selection (Fast Node Selection), with only overlay mesh (Using Overlay), and with all factors enabled (Lsync).

At a high level, we see that the individual contributions are significant, reducing the synchronization latency by a factor of 4-5 versus having no intelligence in the system. We see that performing scheduling improves the completion time for every target ratio, but that intelligent node selection is more critical at lower ratios. This result makes sense, since finding the fast nodes is more important when only a small fraction of the nodes are needed. However, when the ratio is high and even slower nodes are being included in the synchronization process, scheduling is needed to mask the effects of the slow nodes on the latency. The CDN is slow at low target ratios, and becomes comparable to the best end-to-end synchronization latency at high ratios due to its scalable file transfers. However, it shows the worst latency with high variations at the target ratio of 0.98 because some nodes experience performance problems in the overlay.

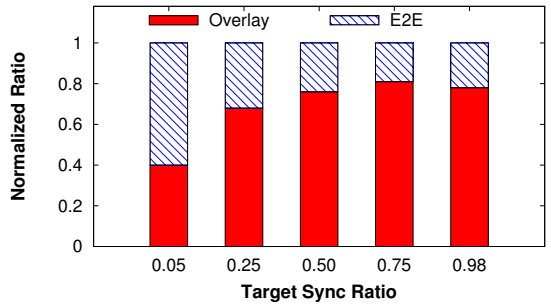


Figure 14: Division of Nodes in Lsync – We see that the fraction of nodes served by overlay mesh changes across target ratios, and that the fraction is not monotonically changing with target ratio.

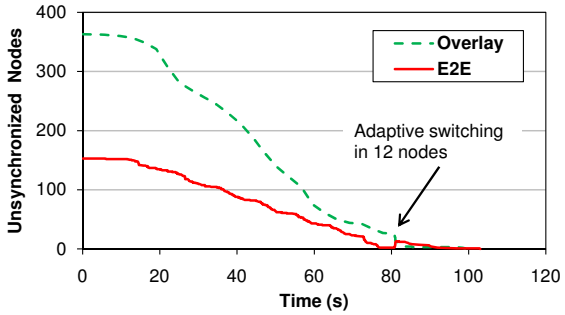


Figure 15: Adaptive Switching in Lsync – At 80 seconds, 12 nodes dynamically switch to end-to-end connections and finish downloading from the origin server.

Lsync combines all the factors in a manner to reduce the latency for all target ratios.

6.6 Nodes Division and Adaptive Switching

To see how Lsync adjusts workload across server and overlay mesh, we further analyze how nodes are divided into the overlay group and the end-to-end group. The nodes in the groups are selected so that both groups are expected to finish file transfer at the same time. In Figure 14, we plot the normalized sizes of the two groups during a large file (5MB) transfer. As the target synchronization ratio increases, a smaller fraction of nodes are served using end-to-end transfers. However, for the target ratio of 0.98, the overlay group's estimated completion time increases because of nodes with slow overlay connectivity. Therefore, the ratio for the origin server increases compared to the case of target ratio of 0.75.

Lsync makes adjustment at runtime, as can be seen in Figure 15, where we plot the number of pending nodes during file synchronization. The target synchronization ratio is 0.98, and r_{e2e}, the ratio of nodes for end-to-end connections, is 0.29. The two groups start downloading a 5 MB file through the overlay mesh and end-to-end connections respectively. At 80 seconds, however, 12 nodes in the overlay group detect that they are having unexpected overlay performance problems, and could negatively affect the completion time. The nodes dynamically switch to the end-to-end connections, and directly download the remaining content from the origin server. This behavior explains the small bump near 80 seconds – when these nodes switch from the overlay group to the end-to-end group, the number of unsynchronized end-to-end nodes increases.

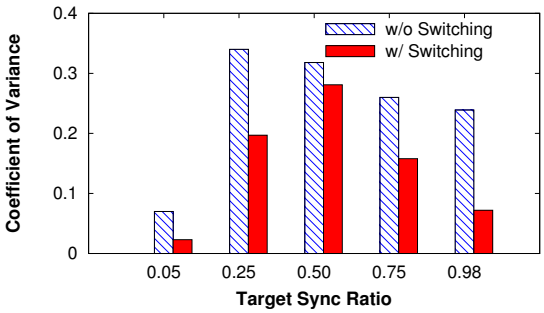


Figure 16: Stable File Transfers in Lsync – Adaptive switching in Lsync lowers variance of the latency.

During 10 repeated experiments, an average of 27 nodes dynamically switched to end-to-end connections. By assisting a few nodes in trouble, Lsync reduces the completion time by 16%. In addition to the reduced completion time, the adaptive switching in Lsync provides stable file transfers because it masks unpredictable variations in the overlay performance at runtime. Figure 16 plots the variations of the completion times with/without adaptive switching in Lsync. The vertical bars plot the coefficient of variance (normalized deviation) of completion times during repeated experiments. For every target ratio, Lsync shows smaller variations compared to the version with the switching disabled.

7 Related Work

CDNs and P2P systems [9, 16, 18, 19, 21] scalably distribute the content to a large number of clients. While these systems achieve bandwidth efficiency and load reduction at the origin server, they typically sacrifice start-up time and total synchronization latency for smaller node groups. Likewise, peer-assisted swarm transfer systems [20] manage server's bandwidth using a global optimization, but they address bandwidth efficiency in multi-swarm environments, not latency.

Our work on managing latency in distributed systems is related in spirit to partial barrier [3] that is a relaxed synchronization barrier for loosely coupled networked systems. The proposed primitive provides dynamic knee detection in the node arrival process, and allows applications to release the barrier early before slow nodes arrive. Mantri [5] uses similar techniques to improve job completion time in Map-Reduce clusters.

Lsync aggressively uses available resources because it would otherwise remain unused. Another example is *dsync* [22] that aggressively draws data from multiple sources with varying performance. dsync schedules the resources based on the estimated cost and benefit of operations on each resource, and makes locally-optimal decisions, whereas Lsync tries to perform globally-optimal scheduling to reduce overall latency.

Gossip-based broadcast [8, 15] provides scalable and robust event dissemination to a large number of nodes. Our measurements demonstrate that the random peering strategy in the protocol helps reduce latency because slow nodes are likely to find nodes with the disseminated data after most fast nodes are synchronized. Lsync shows better latency than the gossip protocol because it targets the slow nodes from the beginning of file transfers, preventing the slow nodes from being the bottleneck.

8 Conclusion

Low-latency data dissemination is important for coordinating remote nodes in large-scale wide-area distributed systems, but the latency has been largely ignored in designing one-to-many file transfer systems. We found that the latency is highly variable in heterogeneous network environments, and many file transfer systems are suboptimal for the metric. To solve this problem, we have identified the sources of the latency and described techniques to reduce their impact on the latency (node scheduling, node selection, workload adjustment, and dynamic switching). We have presented Lsync that integrates all the techniques in a manner to minimize latency based on information available at runtime. In addition to standalone application, we expect that Lsync is useful to many wide-area distributed systems that need to coordinate the behavior of remote nodes with minimal latency.

9 Acknowledgments

We would like to thank our shepherd, Frank Dabek, as well as the anonymous reviewers. We also thank Michael Golightly, Sunghwan Ihm, and Anirudh Badam for useful discussion and their insightful comments on earlier drafts of this paper. This research was partially supported by the NSF Awards CNS-0615237, CNS-0916204, and KCA (Korea Communications Agency) in South Korea, KCA-2012-11913-05004.

References

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

[2] Akamai Inc. www.akamai.com/.

[3] J. Albrecht, C. Tuttle, A. C. Snoeren, and A. Vahdat. Loose synchronization for large-scale networked systems. In *Proceedings of USENIX ATC*, 2006.

[4] Amazon CloudFront. http://aws.amazon.com/cloudfront/.

[5] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in Map-Reduce clusters using Mantri. In *Proceedings of USENIX OSDI*, 2010.

[6] N. Bansal and M. Harchol-Balter. Analysis of SRPT scheduling: investigating unfairness. In *Proceedings of ACM SIGMETRICS*, 2001.

[7] O. Beaumont, N. Bonichon, and L. Eyraud-Dubois. Scheduling divisible workloads on heterogeneous platforms under bounded multi-port model. In *IEEE IPDPS*, 2008.

[8] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Transactions on Computer Systems*, May 1999.

[9] BitTorrent. http://bittorrent.com/.

[10] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: High-bandwidth multicast in cooperative environments. In *Proceedings of ACM SOSP*, 2003.

[11] CoBlitz Inc. http://www.verivue.com/.

[12] A. Davis, J. Parikh, and W. E. Weihl. EdgeComputing: Extending enterprise applications to the edge of the internet. In *Proceedings of ACM WWW*, 2004.

[13] M. Deshpande, B. Xing, I. Lazardis, B. Hore, N. Venkatasubramanian, and S. Mehrotra. CREW: A gossip-based flash-dissemination system. In *IEEE ICDCS*, 2006.

[14] M. A. Eriksen. Trickle: A userland bandwidth shaper for unix-like systems. In *Proceedings of USENIX ATC*, 2005.

[15] P. T. Eugster, R. Guerraoui, S. B. Handurukande, P. Kouznetsov, and A.-M. Kermarrec. Lightweight probabilistic broadcast. *ACM Transactions on Computer Systems*, November 2003.

[16] M. J. Freedman, E. Freudenthal, and D. Mazières. Democratizing content publication with Coral. In *Proceedings of USENIX NSDI*, 2004.

[17] D. Kostic, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. In *Proceedings of ACM SOSP*, 2003.

[18] E. Nygren, R. K. Sitaraman, and J. Sun. The Akamai network: A platform for high-performance internet applications. *ACM Operating Systems Review*, 2010.

[19] K. Park and V. S. Pai. Scale and performance in the CoBlitz large-file distribution service. In *Proceedings of USENIX NSDI*, 2006.

[20] R. S. Peterson, B. Wong, and E. G. Sirer. A content propagation metric for efficient content distribution. In *Proceedings of ACM SIGCOMM*, 2011.

[21] M. Piatek, T. Isdal, T. Anderson, A. Krishnamurthy, and A. Venkataramani. Do incentives build robustness in BitTorrent? In *Proceedings of USENIX NSDI*, 2007.

[22] H. Pucha, M. Kaminsky, D. G. Andersen, and M. A. Kozuch. Adaptive file transfers for diverse environments. In *Proceedings of USENIX ATC*, 2008.

[23] M. O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Harvard University, 1981.

[24] Riverbed Technology Inc. http://www.riverbed.com/.

[25] SETI@home. http://setiathome.berkeley.edu/.

[26] A. Sherman, P. A. Lisiecki, A. Berkheimer, and J. Wein. ACMS: The Akamai configuration management system. In *Proceedings of USENIX NSDI*, 2005.

[27] A. Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, The Australian National University, 1999.