Understanding Resource Usage and Performance in Wide-Area Distributed Systems

Wonho Kim

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer Science
Adviser: Vivek S. Pai

November 2012

Abstract

Many Internet services employ wide-area frameworks to deliver exponentially growing network traffic to end users with low response time. These systems typically leverage a large number of remote nodes at the edge of the Internet, which makes the systems difficult to develop and test. Therefore, federated testbeds are essential infrastructures for developing wide-area systems because they allow researchers to deploy new services under realistic network conditions. In this dissertation, we study resource usage in PlanetLab to understand and characterize user behavior in federated testbeds. We also present Lsync, a low-latency file transfer system for coordinating remote nodes in wide-area platforms, including testbeds.

To support the development of new network services on a global scale, the next generation of federated testbeds are under active development, but very little is known about resource usage in these shared infrastructures. We conduct an extensive study of the usage profiles in PlanetLab that we collected for six years by running CoMon, a PlanetLab monitoring service. We examine various aspects of node-level behavior as well as experiment-centric behavior, and describe their implications for resource management in federated testbeds. We find that the usage is much different from shared compute clusters, that conventional wisdom does not hold for PlanetLab, and that several properties of PlanetLab as a network testbed are largely responsible for this difference.

We also present a low-latency file transfer system, Lsync, that can be used as a synchronization building block for wide-area distributed systems where latency matters. While many distributed systems depend on fast data synchronization for coordinating remote nodes, current data dissemination systems focus on 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 distribution tools, and can dramatically

reduce latency across a wide range of scenarios. Lsync performs novel node selection, scheduling, and adaptive policy switching that dynamically chooses the best synchronization method using information available at runtime. Our evaluation results show that Lsync reduces latency by more than a factor of 14 compared to a widely used synchronization tool, and makes most remote nodes fully synchronized even under frequent file updates.

Acknowledgements

First and foremost, I cannot thank enough my advisor, Professor Vivek S. Pai, for his incredible inspiration, support and patience. Throughout my graduate career, he encouraged me to explore important problems and to stay passionate about discovering right solutions. I learned from him that self-confidence comes only if a researcher is fully accountable for and knowledgeable about what he or she has done. All these lessons will never be forgotten.

I am extremely grateful to the members of my dissertation committee, Professor Larry Peterson, Professor Michael Freedman, Professor David Walker, Professor Brian Kernighan, and Professor Jennifer Rexford. They were generous with their time and provided thoughtful reviews of the original manuscript of this dissertation.

I have also benefited greatly from the advice and assistance of senior Korean colleagues. In particular, I owe a debt of gratitude to KyoungSoo Park who was a former Ph.D. student in our research group and is now a professor at KAIST. Not only did he give me valuable comments on my research but he was also an invaluable personal mentor starting in my first year. I also thank Changhoon Kim, Yung Yi, and Sangtae Ha who have always been willing to share their valuable time with me.

I thank Ajay Roopakalu and Katherine Y. Li who were great colleagues armed with amazing research skills. I was fortunate to have friends who are both academically enthusiastic and warmhearted. Discussions with Sunghwan Ihm, Anirudh Badam, Chris Park, Taewook Oh, Donghun Lee, David Shue, Wyatt Lloyd, Sid Sen, Hanjun Kim, Ana Bell, and CJ Bell cheered me on through my ups and downs. I also thank my roommates, Ilhee Kim and Ringi Kim, who were willing to bear the burdens of sharing their apartment with me from time to time.

I am indebted to incredible administrative support from Melissa Lawson. She was understanding and always came up with effective solutions at times when I had difficulties handling administrative issues.

I thank my mentors and collaborators at HP Labs: Sujata Banerjee, Puneet Sharma, Jean Tourrilhes, and Praveen Yalagandula. During my two internships at HP Labs, they kindly helped me understand network issues in datacenters. My sincere gratitude goes to Sung-Ju Lee and Jeongkeun Lee who have always supported me from the very beginning of my research career.

I also thank my former advisors in Seoul National University: Professor Yanghee Choi and Professor Taekyoung Kwon. The start of my research career would not have been possible without their continuous support.

I sincerely thank my father Moonbong Kim, my mother Youngsim Ko, and my dear sisters, Mihyung and Mihee for their unconditional love. I also thank my mother-in-law Kyoungsook Kim and late father-in-law Dongshik Shin. Their encouragement and deep understanding helped me stay focused on my research work.

Last, but not least, I am greatly thankful to my wife Nah-Yoon Shin, who has stood by me through a very difficult period of my life. I could never have finished this long journey without her love, patience, and support. Thank you Nah-Yoon.

This dissertation was supported by the NSF Awards CNS-0615237 and CNS-0916204.

To my loving wife Nah-Yoon Shin.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Many Internet services employ wide-area platforms [2, 6, 22, 23, 24, 68] to deliver their exponentially-growing network traffic over the public Internet. These systems typically leverage servers at the edge of the Internet to improve user experience. In addition to caching static objects, the edge servers implement enhanced protocols, optimize routes, and even host business applications.

However, it is well known that wide-area distributed systems are notoriously difficult to build [18, 78]. The systems have to manage a large number of remote nodes and adapt to dynamically-changing network conditions at runtime. Also, it is common that rare corner cases are found after the systems are deployed at scale. This means that, to develop production quality wide-area systems, system developers need a way to deploy and test new wide-area services under realistic network conditions in the Wide Area Network (WAN).

The next generation of testbeds have been under active development to provide large-scale experimental facilities to researchers. These testbeds aim to federate existing testbeds or donated servers. However, we note that very little is known about resource usage and user behavior in federated testbeds, leading to the development of conservative resource allocation policies. In this dissertation, we analyze and char-

acterize resource usage in PlanetLab [1], a global network testbed that has served a variety of research projects since it was launched in 2002. Based on real usage patterns we identify, we discuss their design implications for future network testbeds that have similar architectures and design goals.

From the measurement study of PlanetLab usage, we find that most experiments are short-lived but a large number of them often expand to over half the testbed. This means that software development process in PlanetLab is not interactive because long deployment delay will hamper every develop/deploy/test cycle during the short period of the usage. Besides testbeds, the delay is also a problem for many wide-area systems that attempt to continuously coordinate or change the behaviors of their remote nodes for their operations at runtime.

In the second part of this dissertation, we address this delay issue by developing Lsync [40], a low-latency file transfer system for wide-area systems. In our setting, latency is measured as *completion time* of file transfer to all target remote nodes in the WAN. We find that existing file transfer systems are suboptimal for this metric and that we need radically different resource allocation strategies for reducing completion time in one-to-many file transfer in the WAN.

1.1 Resource Allocation in Federated Testbeds

Building on the unprecedented success of PlanetLab, the next generation of testbeds have been under active development recently. In its design phase, the GENI [31] project aims to federate multiple testbeds that are owned and operated by autonomous organizations. It plans to cover diverse networks including PlanetLab-like wide-area testbeds, fiber optics, and even sensor grids.

In the designs of federated testbeds, resource allocators are considered to be key components of the platforms. The resource allocator defines how to allocate the

testbed's available resources to multiple users from different organizations. PlanetLab provides simple fair sharing between concurrent experiments running on the same server, but the common belief is that the federated infrastructures need an extensive policy framework and more sophisticated incentive systems because users will be competing for the shared resources across organizational boundaries.

1.1.1 Previous Approach

Designing a resource allocation policy requires a deep understanding of user behavior in the target system. Unfortunately, very little is known about the usage of the federated testbeds because they are much different from traditional compute clusters. Given the limited knowledge of the usage, it is not surprising that research efforts have been focused on developing resource reservation systems as conventional wisdom suggests that shared testbeds would suffer from a tragedy of the commons if no proper regulation is enforced.

Market-based resource allocation systems [10, 44, 75] attempt to allocate resource in an economically efficient way. The systems take user's valuation of resources as input and provide auction mechanisms to trade resources. Distributed resource management systems [29] provide secure resource peering between autonomous parties through cryptographically-protected tickets. PlanetLab implements a brokerage service to allow users to request more than their fair shares for a fixed amount of time [72]. However, the proposed systems received very little usage when deployed on PlanetLab, though users could obtain free and dedicated resources from the systems with a simple sign-up process. This implies that PlanetLab users may not perceive the amount of resources as the main utility of the wide-area testbeds.

1.1.2 Our Approach and Contributions

Our approach is to conduct a data-driven analysis of usage logs to understand how the wide-area testbeds are actually utilized by real users [41]. Instead of assuming what features users appreciate the most, we attempt to identify the usage patterns from activity logs in PlanetLab and discuss their design implications for similar federated testbeds.

Characterizing PlanetLab usage is a challenging task because (1) the testbed runs a mixture of different kinds of services at any given time and (2) its workload is affected by external events such as major conference deadlines. Therefore, examining a few snapshots is not sufficient for characterizing the overall usage. For this analysis, we used large-scale (1.8 TB) and long-term (six years) datasets that we have collected from 2005 to 2010 through CoMon [55], a scalable monitoring system in PlanetLab.

From the datasets, we characterize various aspects of node-centric behavior as well as experiment-centric behavior. We examine resource consumption (CPU, memory, bandwidth), network reach, workload distribution, and failures in every node and every experiment available during the period. We also explore the effectiveness of previously proposed market-based resource allocation systems by simulating the approaches against the datasets.

Our extensive analysis results provide the following main observations:

- PlanetLab shows no indication of the tragedy of the commons.
- Unlike in compute clusters, PlanetLab users are not aggressive in acquiring available resources, but active in extending their network reach.
- Market-based resource allocation schemes can account only for a small fraction of total resource usage in testbeds.

- Workload is persistently unbalanced among PlanetLab nodes because users do not migrate to new and powerful nodes, but prefer to stay with known healthy nodes.

1.2 Reducing Latency in File Dissemination

Unlike P2P systems, many wide-area systems attempt to maintain a close control over their remote nodes at runtime, which requires *low-latency one-to-many file dissemination*. For instance, commercial Content Distribution Network (CDN) providers generate new configurations as frequently as every 10 seconds [71]. These updates should be disseminated to the remote nodes with minimal latency to provide guaranteed performance to their customers. Other systems often require coordinating multiple remote nodes for their operations. One of the examples is distributed monitoring systems [39, 79] that start a measurement phase only after multiple nodes are coordinated. Reducing the latency has a direct impact on the overall performance and responsiveness of the systems.

System developers also rely on low-latency file dissemination to deploy their modules to remote nodes. Deploying files to remote nodes takes several minutes when hundreds of nodes are distributed in the WAN. As a result, programmers are constantly interrupted by the long deployment delay in every develop/deploy/test cycle. This non-interactive environment can seriously degrade the productivity of the software engineers because people spend more time in recovering from the frequent interruptions than the time they are interrupted [37]. Given that software engineers are much more expensive than hardware servers and network bandwidth now, this loss cannot be ignored.

1.2.1 Previous Approach

Numerous systems have been proposed for scalable data transfer in the WAN. CDN and P2P systems [16, 28, 51, 52, 56, 60, 69] construct overlay meshes among participating nodes and implement request redirection to serve a large number of clients. Likewise, overlay-based multicast systems [11, 19, 38] create multicast trees in the overlay to improve aggregate bandwidth utilization. Gossip-based broadcast systems [15, 27, 46] provide robust file dissemination through random peering and short-term gossip rounds.

The existing systems aim to serve an *open client population* with a limited bandwidth. In an open client population, there is no upper bound on the number of clients being served. Therefore, the main goal of the systems should be to improve average performance in individual clients or aggregate throughput in the system. These performance metrics shaped the design of the systems so that they are mainly optimized for bandwidth efficiency, not the latency in file dissemination.

1.2.2 Our Approach and Contributions

In this dissertation, we design and develop a file transfer system optimized for a different performance metric, latency, in the WAN. Specifically, latency is measured as the completion time of file transfer to multiple remote nodes. In coordinating remote nodes in wide-area systems, it is important to reduce the latest finish time among a fixed client population because the application should wait until all target nodes are synchronized.

Focusing on completion time is a completely different problem that requires different file transfer strategies. As nodes have heterogeneous network conditions in the WAN, the completion time is determined by the slowest node in the system. However, existing systems are not favorable to the slow nodes because improving the slow nodes does not help optimize the average performance of individual clients in the system.

We develop Lsync, a low-latency one-to-many file transfer system for wide-area distributed systems. The completion time metric drives us to examine new optimization opportunities that may not be advisable for other systems. For instance, Lsync aggressively uses available bandwidth in the server for aiding slow nodes at runtime, because the bandwidth would remain unused otherwise. We deploy Lsync on PlanetLab and compare it against a file synchronizer, CDN/P2P systems including commercial systems, and gossip-based systems. The results of the experiments demonstrate that Lsync drastically reduces latency compared to the tested systems under various scenarios. We also show that Lsync provides stable performance in the presence of unexpected bandwidth fluctuation in the remote nodes, which is common in wide-area systems. Lsync's file transfer policy is not tied to a specific protocol but designed to be easily pluggable into many systems. We integrate Lsync into existing data transfer systems [6, 16, 56] to improve their latency.

The design principles that we discovered from Lsync are summarized as follows:

- Scheduling slow nodes earlier can mask the effects of the bottleneck nodes on the completion time.
- Late-binding the selection of target nodes significantly improves the completion time.
- Using an overlay does not always help reduce completion time, so it should be used only for the nodes that benefit.
- Runtime policy switching not only improves completion time but also provides stable performance.

1.3 Dissertation Overview

This dissertation is structured as follows: Chapter 2 describes the analysis of PlanetLab resource usage. We analyze per-experiment characteristics using six years (2005 to 2010) of usage logs that we collected through CoMon. PlanetLab provides fair sharing between experiments running on the same node. We examine the effectiveness of alternative resource allocation schemes that have been proposed for PlanetLab-like federated testbeds. Our simulation results show that they can address only a small percentage of total usage. We also examine the workload imbalance problem in PlanetLab and show that failures are the main cause for the different popularity of nodes.

Chapter 3 describes Lsync, a low-latency one-to-many file transfer system for wide-area systems. We show that existing systems are suboptimal for completion time because they are not favorable to slow nodes, though those nodes typically become bottleneck during a file transfer. We describe a new file transfer policy that gives preference to the slow nodes, adaptively uses an existing overlay, and dynamically switches policies at runtime to address unexpected performance problems in the overlay. We deploy Lsync on PlanetLab and compare it against a variety of wide-area file transfer systems.

Chapter 4 summarizes the lessons learned and concludes the dissertation.

Chapter 2

PlanetLab Resource Usage

2.1 Introduction

In this chapter, we analyze resource usage in PlanetLab and discuss its design implications for emerging federated testbeds. We note that PlanetLab itself is a federated platform. The nodes in PlanetLab are managed by a trusted intermediary named PlanetLab Central (PLC), but each site retains ultimate control over its own nodes. Since it was launched in 2002, PlanetLab has tried to balance fairness and the utility of the system without imposing strict resource controls [14, 57]. Thus, we believe that understanding resource usage in PlanetLab can help shape the policy decisions of future testbeds that have similar design requirements. Since planned testbeds such as GENI have architectures similar to PlanetLab, the lessons we have learned from our analysis can be generalized beyond PlanetLab to many federated systems that need to control shared resources donated by autonomous organizations.

Characterizing PlanetLab's resource usage is challenging because it is highly dynamic and evolves with changes in the underlying platform. For example, some experiments are active year-round and consume an almost constant amount of resources while many other experiments show heavy and bursty demand over short time peri-

ods. As a result, large-scale, long-term analysis is necessary to capture usage patterns and their evolution.

To address this challenge, we have collected detailed statistics on every online PlanetLab node and the active experiments running on the node since August 2004 through the PlanetLab monitoring system CoMon [55]. The collected datasets have detailed information about both node-centric and experiment-centric data at a five minute granularity. In addition to passively recording OS-provided metrics, CoMon also actively gathers information about each node's status by periodically running a set of test programs. In this chapter, we analyze six years of PlanetLab usage, from 2005 to 2010. Our three main observations follow:

No tragedy of the commons Conventional wisdom suggests that network testbeds should suffer from a tragedy of the commons, and this belief has led to much development on PlanetLab, including two deployed resource reservation schemes [44, 72], two deployed resource discovery systems [3, 5], and papers investigating resource allocation and migration [29, 53]. This belief has even shaped the requirements of testbeds like GENI, which are devoting much attention and software development cost to resource reservation systems [30].

However, we observe no indication of the tragedy of the commons on PlanetLab, and we find several measurements indicating that these kinds of network testbeds are unlikely to suffer such effects. Unlike compute clusters where users try to utilize every available resource, most PlanetLab users are not aggressive in using resources in the testbed. While PlanetLab hosts some long running services, most PlanetLab experiments have bursty resource consumption, and this resource consumption is tied to network activity. As a result, the resource consumption shows bimodal distributions along many axes. The primary reason for the non-aggressive behavior of PlanetLab users is that the main utility of PlanetLab comes from its wide network vantage points, not the aggregate amount of resources.

Limitations of market-based resource allocation Using data-driven analysis, we explore the effectiveness of two representative resource allocation schemes proposed for PlanetLab-like federated systems: pair-wise bartering and market-based banking. We find that the bartering and banking systems can account only for 3% and 14% of the total resource usage, respectively, because most resource usage is from sites that use more resources than they donate. Since the remaining 83% of the resources need to be allocated, market-based allocation approaches are not sufficient for network testbeds, and some mechanism must be employed to ensure that the bulk of the testbed's resources are used appropriately.

Improving utility of PlanetLab We examine factors that degrade the overall utility of PlanetLab, and discuss how to mitigate their impact. We find that the workload is persistently unbalanced among PlanetLab nodes, resulting in high resource contention in overloaded nodes as well as inefficient resource usage. Several factors are responsible for this imbalance, ranging from users staying with known-good nodes to node utility being degraded due to DNS failures, node unreliability, bandwidth limitations, and other reasons. We also find unstable experiments consume a disproportionately high share of the resources, typically dwarfing stable long-running services. We simulate pruning the problematic experiments to measure their impact on other well-behaved experiments in PlanetLab.

The rest of this chapter is structured as follows. In Section 2.2, we describe some background on PlanetLab and the CoMon datasets. We analyze per-slice characteristics in Section 2.3, and examine several resource allocation systems in Section 2.4. We examine the workload imbalance problem in Section 2.5, and discuss policing of slices in Section 2.6. We compare our observations with related work in Section 2.7.

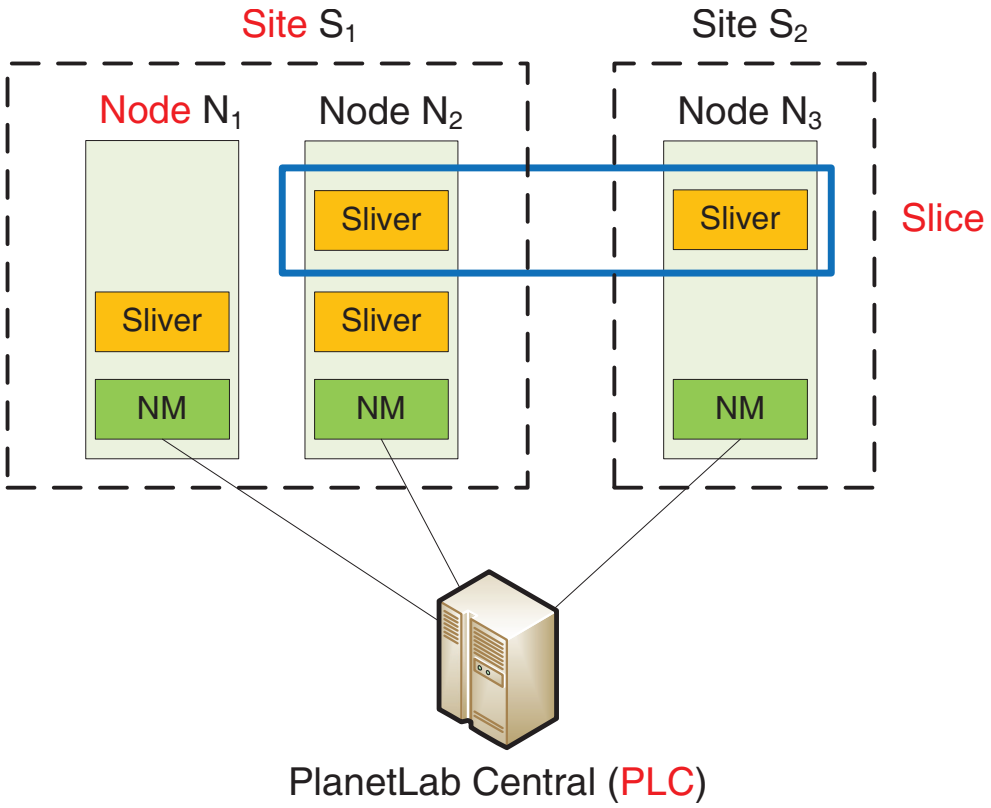


Figure 2.1: Overview of PlanetLab Architecture. PlanetLab consists of nodes distributed at different sites. PlanetLab Central (PLC) manages user accounts and creates virtual machines called slivers for PlanetLab users. Multiple slivers can run at a node at any given time, and the set of slivers assigned to one account is called a slice.

2.2 Background and Datasets

To better understand the analysis in this chapter, some background on PlanetLab and its terminology is provided here. Figure 2.1 illustrates the architecture of PlanetLab and its components. When organizations join PlanetLab, they host physical servers at one or more locations. Each location is called a *site*, and the servers are called *nodes*. All account creation and node management is handled by a centralized database, called PlanetLab Central (PLC). Users create accounts on one or more PlanetLab nodes to perform their experiments. The nodes host one virtual machine per account, and users can run any number of processes within their own slivers. The virtual machines are called *slivers*, and the set of virtual machines assigned to one account

is called a *slice*. PlanetLab is a shared testbed, so multiple slivers are running on the same node at any given time.

We classify a sliver as an *in-memory* sliver if it contains at least one instantiated process, regardless of whether the process is running or blocked. An in-memory sliver is called a *live* sliver if it uses more than 0.1% of the CPU per day.[1] A slice that has at least one live sliver is called a live slice. We say that a slice *uses* a node if the slice has in-memory slivers on the node. A node is considered to be live if it responds to CoMon requests.

In this chapter, we analyze six years (2005 to 2010) of data from CoMon, a scalable monitoring system for PlanetLab. Since August 2004, CoMon has collected and reported statistics on PlanetLab nodes to help PlanetLab users monitor their services and spot problems. CoMon runs daemons on every PlanetLab node to gather values that are provided by operating systems, and values that are actively measured by means of test programs running on the nodes. A central CoMon server collects data from all PlanetLab nodes every 5 minutes.

CoMon monitors and collects *node-centric* and *slice-centric* data. The node-centric datasets consist of 51 fields that represent node health and aggregate resource consumption, including CPU utilization, memory usage, timing behavior, DNS resolver behavior, bandwidth consumption, etc. The slice-centric data contains information about each sliver's resource usage on its node, which is an aggregate resource used by all processes within the sliver in the node. The measured metrics include CPU usage, memory consumption, and transmit/receive bandwidths.

Table 2.1 shows the basic statistics about our datasets. The size of the datasets has increased over time because PlanetLab's node count has increased and more metrics have been added to CoMon over time. The slice-centric datasets contain resource usage of each sliver, and can be aggregated into slices as needed. Since CoMon

[1] 0.1% is the minimum CPU time that CoMon measures for a sliver's CPU usage at any given time.

Year	Nodes	Slices	LiveSlices	Size
2005	354 (62)	215 (14)	106 (8)	164.8 GB
2006	433 (33)	278 (38)	136 (18)	232.9 GB
2007	438 (77)	371 (54)	133 (21)	260.7 GB
2008	474 (85)	254 (66)	139 (25)	291.2 GB
2009	613 (55)	349 (103)	145 (19)	430.1 GB
2010	683 (67)	421 (62)	158 (15)	503.9 GB
Total				1883.6 GB

Table 2.1: Summary statistics for CoMon datasets. Each row contains means and standard deviations of online nodes, in-memory slices, and live slices per day in each year. The size of CoMon logs has increased over time as PlanetLab has expanded itself in its size and user base.

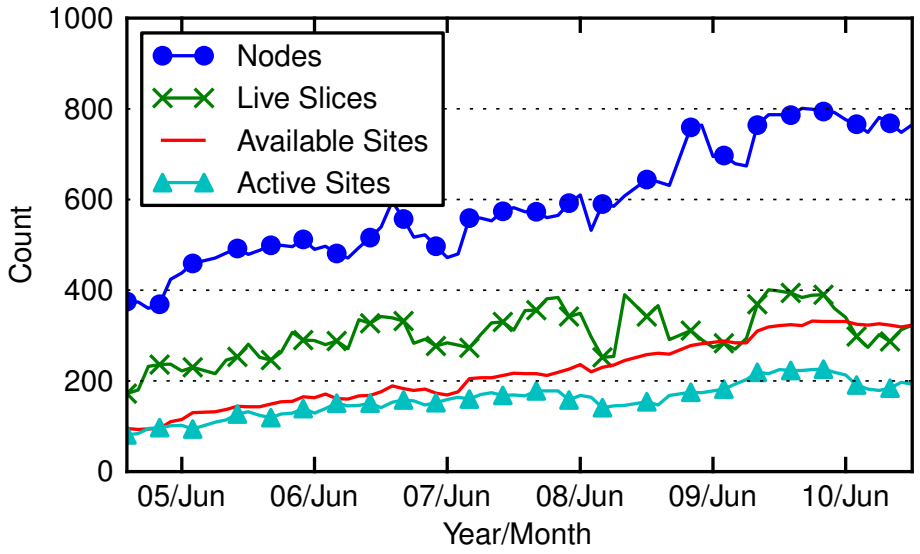


Figure 2.2: The scale of PlanetLab over time. PlanetLab nodes and live slices have increased as more researchers have joined the testbed in general.

fetches data from all PlanetLab nodes in parallel, the aggregated values estimate the total amount of resources that a slice uses across multiple nodes at a given time. We associate the two kinds of datasets to study the effect of an experiment's behavior on a node's status, and vice versa.

CoMon's task has grown over time as the testbed itself has expanded, since CoMon tries to monitor information about every sliver in the system. Figure 2.2 presents the numbers of online nodes, live slices, sites running online nodes (labeled as "Available

Sites"), and sites having live slivers in other remote nodes ("Active Sites") per month. We find that the scale of the testbed has persistently increased over the period (2005 to 2010) that we examine. The number of available sites and their nodes has increased by 179% and 82%. The active site count has increased at a slower rate (95%) because most PlanetLab users intermittently run their experiments in the testbed, and their usage of the testbed is spread over time. Similarly, the number of live slices has increased by 48%. In particular, it is notable that live slice count has been fluctuating over time, which implies that resource demands on PlanetLab are synchronized to some degree with external events such as conference submission deadlines. Sliver count has increased at a faster rate (123%) than slices because slivers counts grow as a result of slice growth and node growth. Most slices only create slivers on a fraction of the nodes, but some slices, particularly those related to infrastructure, are typically created on every node in the system.

2.3 Slice Resource Usage

Examining the slice resource usage in PlanetLab allows us to determine how experiments are using the system, and the patterns of resource consumption on the testbed. We find that PlanetLab experiments are typically bursty along several dimensions, and that most use relatively few resources at any given time. This likely stems from the network-centric experiments on PlanetLab – their resource consumption is tied to their network activity, rather than the total resource pool on PlanetLab.

2.3.1 Active Periods

Since PlanetLab slices share nodes, we begin our per-slice analysis by examining how long slices tend to run and actively use resources. We define a sliver's *active period* to be the number of hours during which the sliver is continuously live on its node.

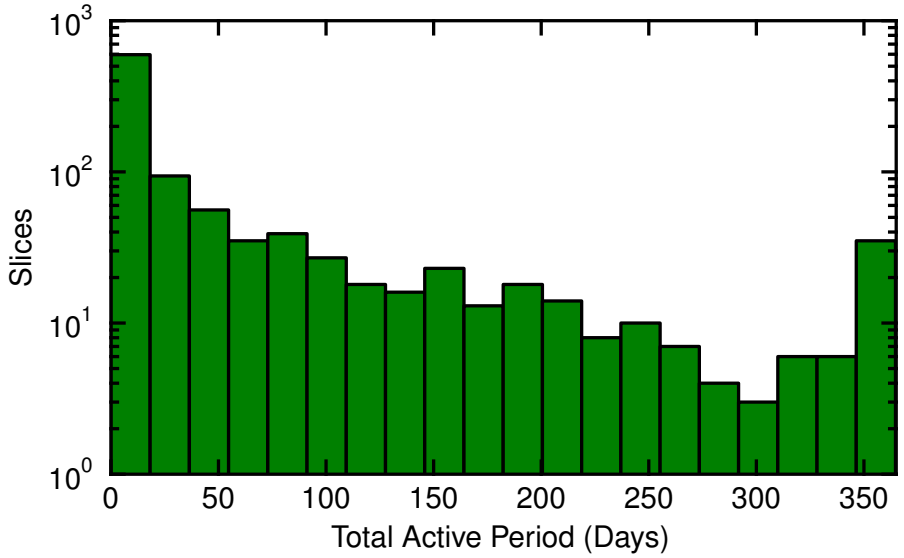


Figure 2.3: The distribution of slice total active period. While most slices are short-lived, a number of slices were active for an entire year.

Experimenters typically leave slivers instantiated on nodes for long periods of time, and only use the slivers when actively performing experiments, resulting in multiple active periods separated by idle periods in CoMon datasets. We consider a slice active if any of its slivers are active, even if sliver count changes over time.

We find that slice activity is largely bimodal, with a great many short-lived slices and a number of very long-lived slices, as shown in Figure 2.3. The number of short-lived slices is not surprising, since many classes use PlanetLab for hands-on measurement projects and short assignments. It is also notable that there are 26 slices that were active longer than 360 days in 2010. These slices include 6 management slices (e.g., root and SliceStat [73]), and 20 long-running services that run on PlanetLab [17, 18, 28, 36, 56].

One possible observation from this data is that short-lived slices are an important aspect of PlanetLab usage, and that it may serve as a training facility for future developers. As such, all of the setup overhead on PlanetLab may be an issue for this class of user, who has to perform these tasks and then amortizes that effort over a

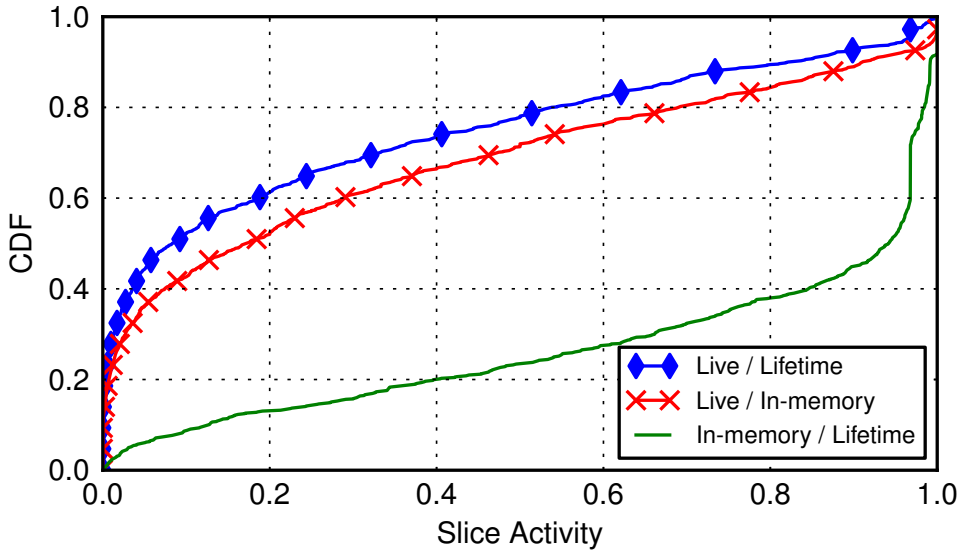
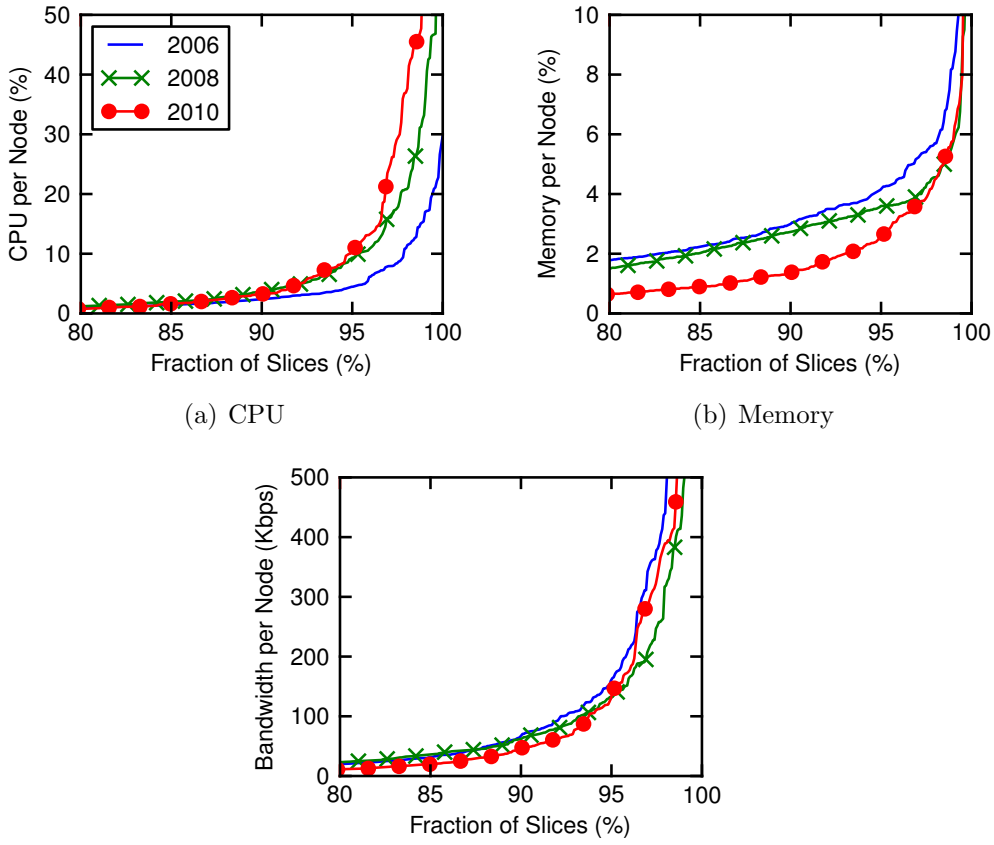


Figure 2.4: CDFs of the relative activity of slices over their lifetimes. Versus their lifetimes, most slices are active for relatively short periods of time (Live/Lifetime). The ratio of activity is even low when compared to the time a slice is instantiated (Live/In-memory).

relatively short usage. Testbed designers may be well advised to focus on simplicity as a way of gaining usage.

The other implication of this result is that most slices do not use a significant amount of resources, even when they are active. Shown in Figure 2.4 is that the total active periods of the slices is often spread over a much longer slice lifetime (Live/Lifetime), so many of those slices are idle for most of their lifetimes. The ratio of activity is even low when compared to the time a slice is instantiated (Live/In-memory). Slices often tend to also disappear and re-appear over time, with large gaps in time when they are not present at all on the testbed. Any attempt at introducing heavyweight resource allocation mechanisms would therefore have two side effects – it would burden the users of most slices, and it would often require resource overcommitment anyway in order to ensure that the resources are being used.



(a) CPU

(b) Memory

(c) Bandwidth

Figure 2.5: Per-sliver usage of CPU, memory, and bandwidth for slices in 2006, 2008, and 2010. Most slices have low resource consumption except for the heaviest 5% of slices. The CPU usage shows the heaviest slices gaining a larger share over time, while memory usage shows flatter curves. The heaviest bandwidth consumers typically provide services to large external user populations.

2.3.2 Local Resource Consumption

Understanding the distribution of slice resource usage and its change over time provides insight into the workload profiles of PlanetLab experiments. We focus on three resources – CPU, memory, and network bandwidth, and examine them on a per-sliver basis when the sliver is active. We do not include disk usage in our analysis because PlanetLab disk space is partitioned into per-sliver quotas (5 GB), and not shared by multiple users in a node.

Since most slivers have fairly low resource consumption, we focus only on the heaviest quintile of resource consumers, shown in Figure 2.5. The three graphs have similar characteristics, in that the aggregate resource consumption is a few percent at most, and increases sharply as we approach the heaviest 5% of slices. However, important differences are apparent when examined in closer detail. The CPU usage, for example, shows the heaviest slices gaining a larger share over time, while memory usage shows the exact opposite.

We believe these differences are a result of PlanetLab policy and the changes in hardware over time. PlanetLab uses a modified CPU scheduler [57], which allocates CPU evenly across slices (not threads/processes), and then allocates any unused CPU on demand. Over time, as more machines enter the PlanetLab testbed, and more powerful machines enter the system, the aggregate CPU in the system increases. As a result, more slices can have their CPU demands met, and the remaining CPU is used by the heaviest consumers. However, these heavy CPU consumers are not heavyweight long-running services, but instead are classified as spin-loop slivers that consume many CPUs but do not generate any network traffic, which we cover in more detail in Section 2.6. This result suggests that CPU contention in PlanetLab has been decreasing over time.

The CDF of memory consumption differs in that the curves are much flatter, and that the opposite effect occurs over time, with the heaviest consumers using less of the testbed. One reason that partly explains both features is that PlanetLab's policy for memory allocation is that when a node runs out of swap space, the heaviest consumer of physical memory is killed on that node. As a result, slices have a tendency to police their own memory usage to avoid being the heaviest consumer, leading to a flatter memory consumption profile among slices. Over time, as the memory capacities of the nodes have increased, the self-policing behavior introduced by the memory-killing policy results in slices consuming a smaller fraction of memory over time.

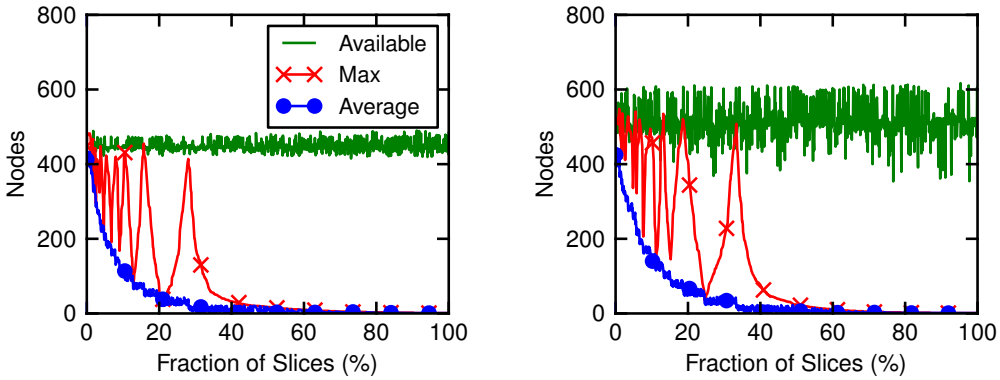
Bandwidth consumption is related to the testbed itself – for those experiments without a large user population, bandwidth consumption is driven by the experiment itself and whatever bandwidth caps the nodes have been assigned. The heaviest bandwidth consumers typically involve large external (non-PlanetLab) user populations, such as content distribution networks or peer-to-peer systems, and the consumption of these systems is not captured on this graph.

2.3.3 Slice Sizes and Dynamics

As we have seen that PlanetLab's resource distribution and slice distribution has many bimodal properties, it is worth investigating whether PlanetLab is monopolized by only a handful of researchers, or whether it has a broader utility to the community. As we believe that PlanetLab's main differentiator versus other testbeds is its network reach, one measure of this utility would be to see how different slices use PlanetLab's scale.

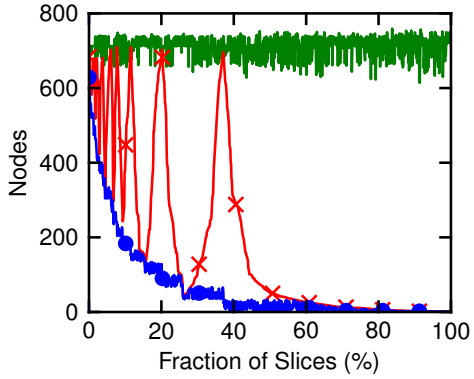
To visualize the range of sliver usage within slices, we want to view the average and maximum daily sliver counts for the slices. However, viewing this data sorted by only one of these values results in garbled images since the average and maximum values are not necessarily correlated. To address this problem, we divide slices into 20 groups by their daily average sliver counts. We then sort slices within each group by their maximum sliver counts. The sort order alternates between ascending and descending for the different groups for aesthetic appeal. The results for 2006, 2008, and 2010 are shown in Figure 2.6.

These graphs show a number of interesting features – the first is that most slices have a relatively low average number of slivers, with 75% of the slices using less than 7% of the available nodes on average, and only 4% use more than half the available nodes on average. However, we also see that the maximum number of nodes used approaches the total size of PlanetLab for virtually every range of average node



(a) 2006

(b) 2008



(c) 2010

Figure 2.6: The average and maximum daily sliver counts for slices in 2006, 2008, and 2010. Most slices have a low average number of slivers, but a large number of them have relatively high maximum sliver counts in their lifetimes.

counts. This result suggests that while much of the development of experiments may happen at low sliver counts, many researchers are in fact expanding their experiment to a large fraction of the available nodes at some point in the slice's lifetime.

This usage pattern becomes more apparent when we aggregate the slices into the sites that created them, and then examine network reach, as shown in Figure 2.7. For each site, we combine the locations of all slivers created by researchers from that site, and examine the locations of those slivers, by site and continent. We find that 105 sites were purely donating resources in 2010, and did not run any slices of their own on the rest of the network. At the same time, 132 sites (35%) used more than 200 remote sites (53%). Taking the inactive sites into account, this result implies that

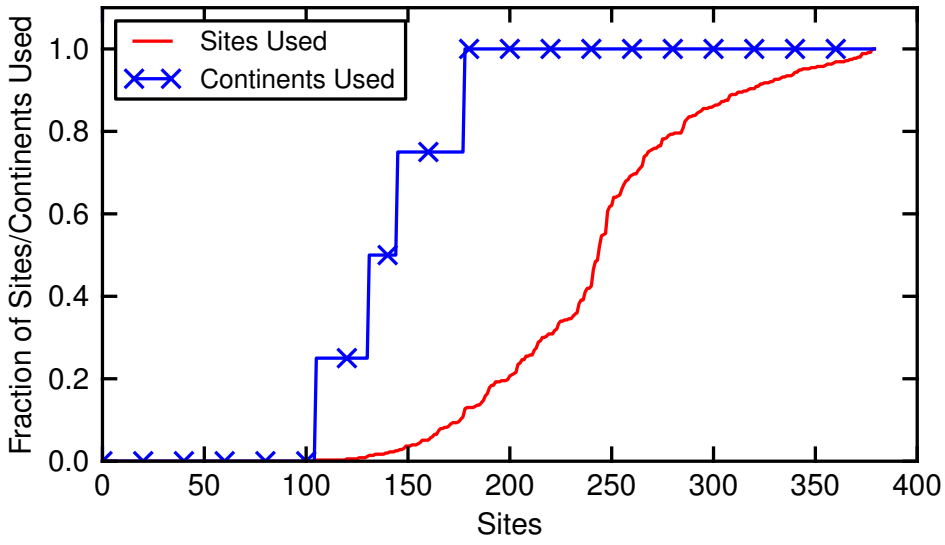


Figure 2.7: The distribution of network reach that PlanetLab sites used in 2010. More than 50% of all active sites used remote nodes in half of all available sites and every accessible continent in PlanetLab.

50% of all active sites used more than half of all available sites for their experiments, which would be impossible without PlanetLab-like global testbeds. Similarly, 73% of all active sites used nodes in every accessible continent (North America, South America, Europe, and Asia) in PlanetLab.

Combined, these two results demonstrate the main utility of PlanetLab – it allows researchers much larger network reach than they would have from just their own sites. Snapshots over small time periods are likely to understate this usage, since most experiments run on a small number of nodes for most of their lifetimes, but a large number of them expand to over half the testbed at some point in their lifetimes. This kind of utilization of the network is not likely to be captured by examining CPU or memory resources, as would be appropriate for computer clusters.

2.3.4 No Tragedy of the Commons

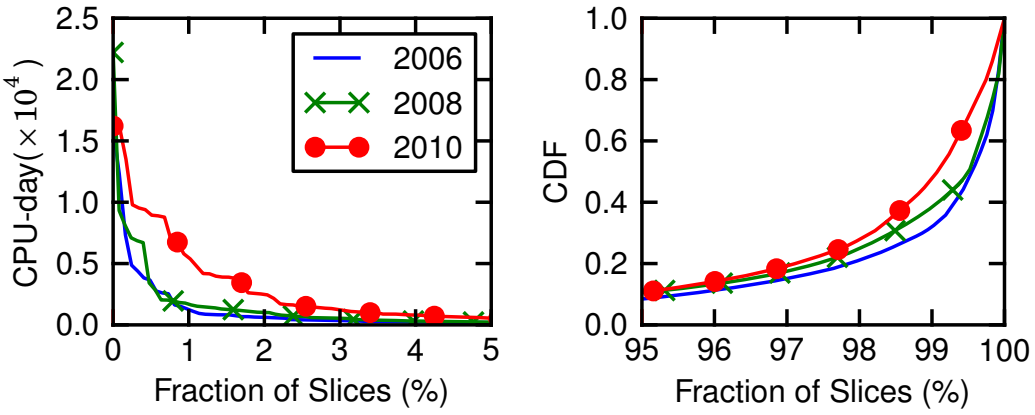
From the details presented earlier in this section, we find no measurement-based support for the idea that there is a tragedy of the commons on PlanetLab. Most

experiments are relatively small most of their lifetimes, and use a large fraction of the testbed in a bursty manner. Likewise, resource consumption is relatively low for most slices, and the fact that the largest CPU consumers are runaway processes suggests that many more slices could get more CPU if needed.

The related question is that if PlanetLab appears to have excess capacity, why is that capacity not being (even surreptitiously) tapped by non-networking researchers? We believe that several PlanetLab policies make it unappealing for compute-intensive researchers. The first is that PlanetLab is organized with a small number of nodes (typically 2) per site, and a large number of sites. This model is different from compute grids, which have a large number of nodes per site connected with high-bandwidth local-area networks. In comparison, the external bandwidth capacity at many PlanetLab sites is in the range of 1-10 Mbps, so the ratio of CPU to bandwidth is much different than compute grids. Not only does PlanetLab have a worse bisection bandwidth than LANs, but the latency between nodes is much higher due to the physical distance. The other issue is the available memory – a large memory footprint increases the chances of a sliver being killed, so memory-intensive compute applications are not well-matched to PlanetLab. This combination of high CPU and low bandwidth is typical of certain bag-of-tasks parallel applications, such as SETI@home [70], but volunteers can provide far more CPUs than are available on PlanetLab.

2.4 Resource Allocation

Since the usage behavior of PlanetLab experiments is very different from compute-intensive testbeds, decisions on resource allocation policy are likely to also be impacted. In this section, we use the historical usage data to examine the resource allocation systems that have been proposed for federated network testbeds.



(a) Per-slice CPU usage

(b) CDF of total CPU usage

Figure 2.8: Total CPU consumptions by slices in 2006, 2008, and 2010. Only 3% of all slices can account for 80% of all CPU usage in PlanetLab.

2.4.1 Total Resource Consumption

To understand the impact of resource allocation proposals, we must first understand resource consumption, which has many dimensions, such as node count, length of running time, resource consumption per node, etc. To capture the different usage patterns and to reduce the dimensions of the problem, we focus on the contended resources, CPU and memory, and aggregate per-slice usage across time and across the entire testbed.

We represent a slice's total resource usage in units of *CPU-day* and *MEM-day*. A CPU-day means the total CPU time that a single CPU core provides per day. Likewise, we define a MEM-day as the total memory space that a node provides per day. If a sliver uses 20% of CPU time and 10% of memory space in 10 nodes for 2 days, its total resource usage is 4 CPU-days and 2 MEM-days.

Figure 2.8(a) presents the distributions of per-slice total CPU usage in decreasing order. The top 1% of slices use more than 10^3 CPU-days while the medians are below 0.1 CPU-days in the years that we examined. PlanetLab slices consumed in total 0.9×10^5, 1.2×10^5, 2.2×10^5 CPU-days in 2006, 2008, and 2010 respectively. Figure 2.8(b) presents a detail from the CDFs of the total CPU usage. We find that

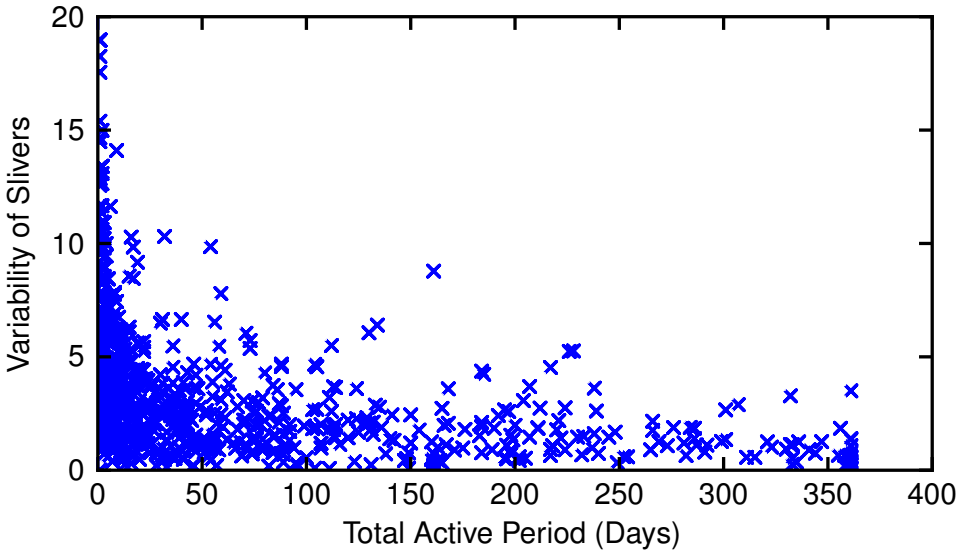


Figure 2.9: The distribution of each slice's total active period and coefficient of variation in its sliver count over time. Long-running slices show relatively lower variability than short-lived slices.

Slice group name	Slice count	Percentage
Short	718	56
Medium	370	29
Long-intermittent	171	13
Long-continuous	16	2
Total	1275	100

Table 2.2: The distribution of slice groups. The majority of slices are in the short or medium slice groups.

only 3% of all slices can account for more than 80% of all CPU usage in PlanetLab. Memory usage has a similar pattern, with 4% of all slices account for more than 80% of all memory usage.

2.4.2 Resource Usage by Experiment Type

To understand what kinds of slices are creating resource demands and the extent of their demands, we categorize slices into several groups and compare their aggregate resource usages.

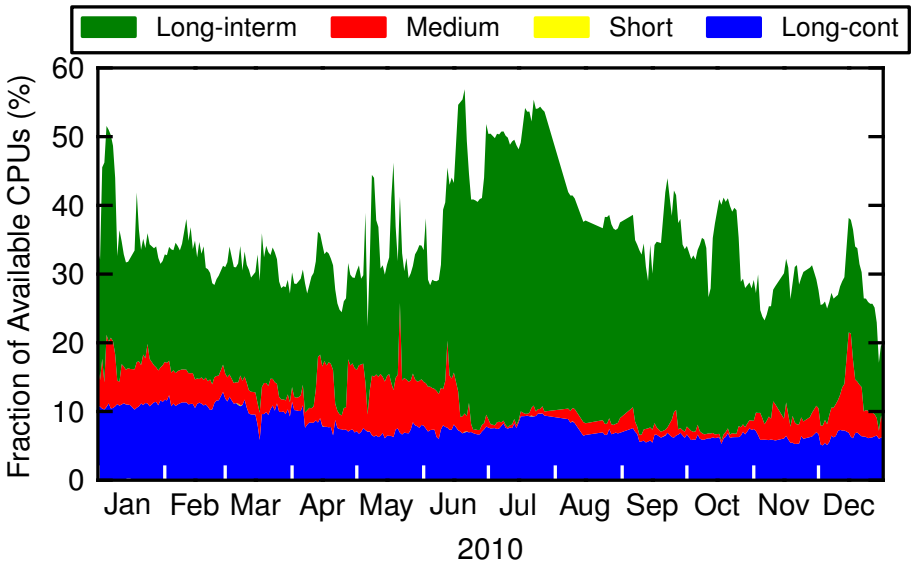


Figure 2.10: Time series of the distributions of CPU usage by each slice type in 2010. The y-axis represents the fraction of available CPUs consumed by slices per day. The Long-intermittent slices consume the largest amount of the resources with high variation.

We first divide slices along two axes, the variability in the number of slivers and the lifetime of the slice. Using these groupings, we would expect an infrastructure service to have a long live and a stable sliver count, while a bursty experiment would have a short life and a variable sliver count. Figure 2.9 plots each slice's total active period and variability of its slivers. The variability of slivers is measured as the coefficient of variation in a slice's sliver count over time. We find that long-running services show relatively lower variability than short-lived slices in general. For simplicity, we pick some break points, with Short slices having less than a week of total activity per year, Medium slices having between one week and 100 days of activity, and Long slices having more than 100 days. We further divide the long slices into continuous or intermittent based on the normalized deviation of sliver count, with continuous slices having a deviation of less than 0.25. The number of slices in each of these groups is shown in Table 2.2.

Daily aggregate CPU usage across the testbed, divided by the slice types, is shown in Figure 2.10. We first calculate the total CPU usages by all slices per day. The

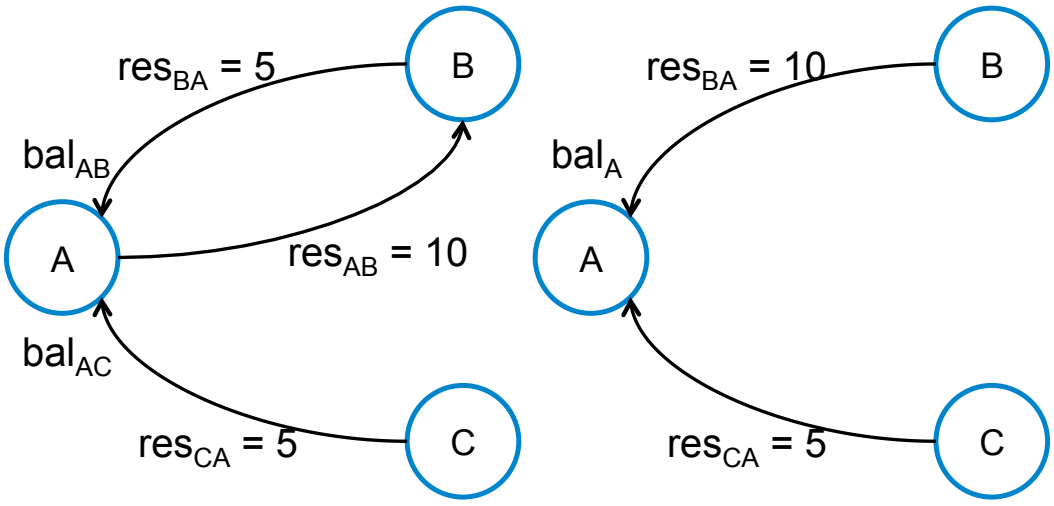
Short slices, despite being over half of all slices, show virtually no CPU usage, while the Long-continuous slices, at 2% of all slices, consume roughly 5-12% of all available CPUs over time. The Medium and Long-intermittent slices are much more bursty in their CPU usage, with the Medium slices showing the least activity during the summer months and the start of the academic year. This behavior would correlate with PlanetLab being used for coursework and projects during the academic year.

Upon closer inspection of the slice groups, we find that the slices in the Long-continuous group run infrastructure services. The slices provide package management [18], monitoring [45, 55, 73], or scalable file distribution services [28, 56, 78]. These slices, while often heavy bandwidth consumers, are surprisingly not a huge impact on PlanetLab's CPU, presumably because a production-quality service that has external users must take some care to run stably. On the other hand, the Long-intermittent slices are the primary source of the fluctuation in the workloads of the testbeds. For example, we find that many spin-loop slices, described in Section 2.6, belong to this group.

2.4.3 Resource Allocation Systems

We extend our analysis to explore the effectiveness of alternative resource allocation schemes in PlanetLab. Among the various resource allocation systems proposed for federated platforms, we focus on two representative approaches in this section: *pair-wise bartering* and *centralized banking*. Other schemes, such as chaining resources among a subset of the nodes [29], would fall between these two extremes. We examine how well the PlanetLab workload could be addressed by the alternative resource allocation systems if they were widely deployed in PlanetLab.

The two schemes we have selected represent the envelope of resource allocation schemes, since pair-wise barter is the most restrictive and banking is the most permissive. The two schemes assume that users trade their resources with each other,



(a) Bartering (b) Banking

Figure 2.11: Balance accounts in bartering and banking. In bartering, each site has separate balance accounts for the other sites. In banking, each site has a single balance account managed in a centralized bank.

or bid their resources to reserve remote resources in other sites. In bartering, a site A grants certain units of A's resources to site B in exchange for access to the same units of B's resources. This peering enables the sites to trade their resources without central agreement. In the central banking system, a site earns virtual currency budgets based on the amount of its donated resources, and obtains remote resources by spending its balance.

To calculate the amount of the PlanetLab workload that could be addressed by the resource peering schemes, we break each site's total resource usage into four categories: Self, Barter, Bank, and Slop. If a slice from site A runs on nodes owned by the same site A, we classify the slice's resource usage as *Self*.

In bartering, each site keeps separate balances for the other sites. Figure 2.11(a) shows three sites A, B, and C. A has two balance accounts, bal_{AB} and bal_{AC}, for site B and C, respectively. bal_{AB} increases when B uses resources at A's nodes. res_{BA} represents the amount of resources that B uses at A's nodes. Likewise, the balance decreases when A uses B's nodes. Given resource usage between PlanetLab sites, we

calculate the amount of the bartering usage as

$$Barter = \sum_{A,B \in S} \min(res_{AB}, bal_{AB}) \tag{2.1}$$

for every pair of sites in all PlanetLab sites S. In Figure 2.11(a), *Barter* is 5 while the total CPU usage is 20.

In banking, each site has a single balance across the testbed. In Figure 2.11(b), A's balance bal_A increases when other sites use resources at A's nodes. A can use B's resources by spending its balance. First, we calculate the total resources that A uses at other sites as

$$res_A = \sum_{B \in S, B \neq A} res_{AB} \tag{2.2}$$

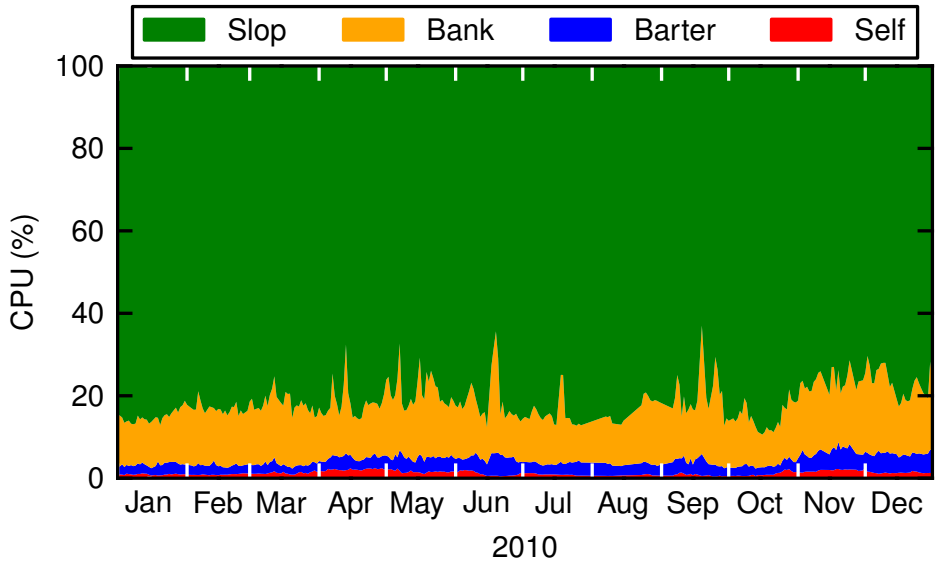
Then we compute the sum of the usages of all sites that can be addressed by each site's balance account.

$$Bank = \sum_{A \in S} \min(res_A, bal_A) \tag{2.3}$$

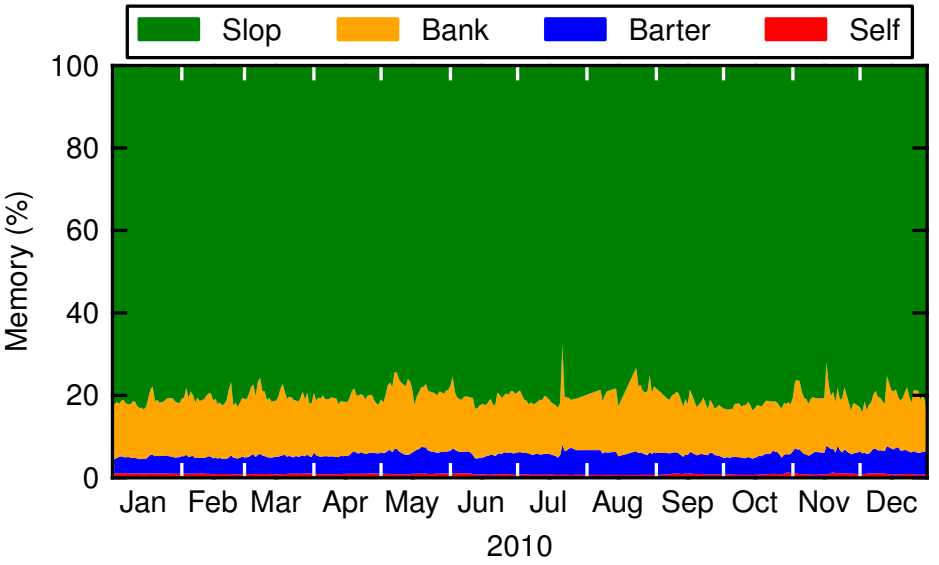
In Figure 2.11(b), A uses C's resources by spending the balance that it earned from B. The *Bank* is 5 while the total CPU usage is 15 in this example.

Lastly, the remaining resource usage is called *Slop*, which is the amount of resources that sites used beyond what they contributed.

For simplicity and generality, we make a number of assumptions about the resource allocation schemes, but these do not lead to a loss of generality in the results. For bartering schemes, we assume a uniform exchange rate, rather than a dynamic exchange rate proposed by some systems [10, 20, 44], since the dynamic rate would only restrict some of the exchanges we observe. Similarly, in central banking, we do not impose any upper limit on resource balances, which allows us to capture all exchanges that could be performed in a central banking model. We consider the effect of resource limits later in this section.



(a) CPU usage



(b) Memory usage

Figure 2.12: Time series of the distributions of CPU/Memory usage that could be addressed by several resource peering schemes. Barter and Bank can account for only 17% of the total CPU usage on PlanetLab while most CPU usage is from Slop. Memory usage shows a similar distribution. Bank and Barter schemes show slightly higher percentages (19% total) than for CPUs, but most memory usage still comes from Slop.

Figure 2.12 presents the amount of testbed-wide CPU/Memory usage addressed by each category. As expected, the percentage of Self usage is low, since sites would

have little reason to join PlanetLab and use solely their own machines. The Barter approach handles on average less than 3% of the CPU usage on PlanetLab, and the Bank approach handles an additional 14% beyond Barter. The vast majority of the CPU usage, however, cannot be handled by any of these approaches, and is allocated from Slop. This implies that, no matter the underlying exchange rate mechanism, there is not much demand for resources that the resource peering schemes are able to handle. It also means that there has to be a policy to handle allocation when bartering or banking fails since most resources will be allocated via Slop. For memory usage, the Bank and Barter schemes show slightly higher percentages (19% total) than for CPUs, but the results are consistent with CPU usage in that most memory usage also comes from Slop. The underlying cause of these results is that CPU demand is unbalanced, with most sites using a large network reach but relatively little CPU. Schemes that attempt to allocate resources must contend with the fact that, for the vast majority of users, CPU is in relatively low demand.

We extend our simulation of the pair-wise bartering scheme to examine the effectiveness of resource routing in PlanetLab, which enables sites to access resources in more remote sites through some transitive ticket redemption paths. If site A has a ticket to claim on B's resources and B has a ticket to claim on C's resources, A could request C's resources using the chained redemption of its ticket to B ($A \rightarrow B \rightarrow C$). For conservative evaluation, we assume that every site has the global knowledge of the distribution of tickets at a given time, and we set no limit on the length of the path that can be used to claim resources.

We find that such resource routing enables PlanetLab sites to find 3.2 times more CPUs than with the pair-wise bartering scheme, but that the total usage addressable by chained approaches is still less than 7% of total CPU usage. Even with unlimited chain lengths, the chained bartering schemes have a result that is much closer to pair-wise bartering than to central banking, which suggests that the chains are likely

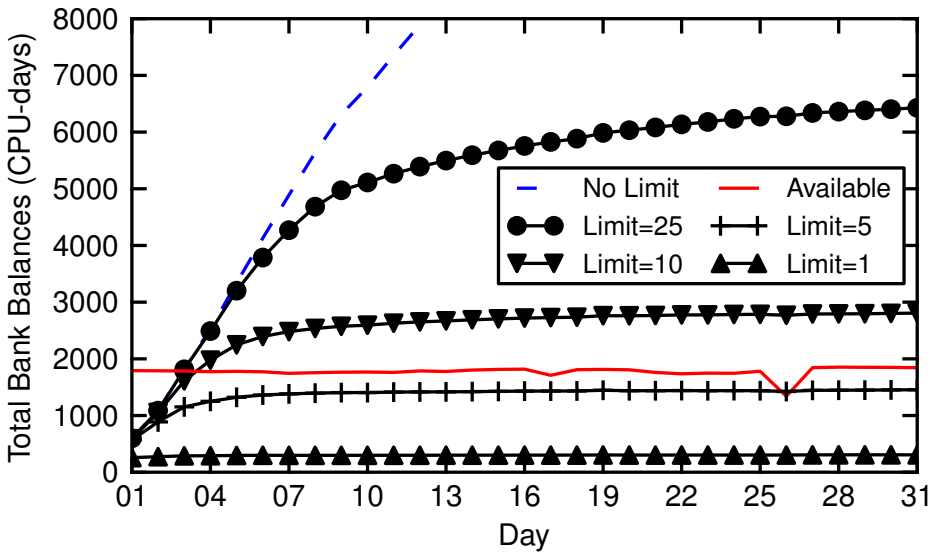
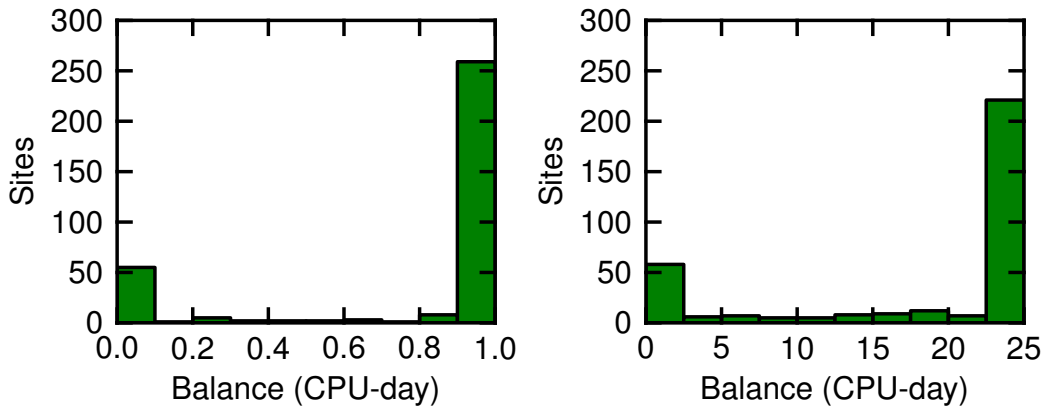


Figure 2.13: The total balances amassed at all sites. Without balance limits, the total balances will exceed the daily capacity of PlanetLab within 3 days, leading to inflation of virtual currency.

to be fairly short. Indeed, we find that the maximum length of the used path was 18 hops, but most of the CPU addressed (85% of the 7%) in bartering was found in sites within 4 hops.

Despite the Bank approach addressing only 14% beyond Barter, any feasible implementation would achieve even less since some limit has to be placed on the balances each site can accumulate. Using the data for January 2010, we examine the bank balance growth in Figure 2.13, and show the effect of various limits. With no limit on balances, the total bank balances grow quickly and exceed the daily capacity of PlanetLab ("Available") within three days. Beyond this limit, the virtual currency becomes inflated, as more currency is accumulated than can ever be spent. With balance limits enforced, the total balance converges at certain points. We find that five CPU-days limit (or lower) can prevent the total balances from growing beyond what PlanetLab can serve. Since most nodes in PlanetLab are dual-core or quad-core machines (82%), the five CPU-days correspond to two physical machines, which is what each site typically provides.



(a) 1 CPU-day Limit

(b) 25 CPU-day Limit

Figure 2.14: The distribution of bank balances among sites. There are bimodal distributions with most sites being near the limits.

Even if the total balance is bounded, the utility of banking and resource auctions depends on users willing to outbid each other in order to get access to resources. However, we find that bank balances tend toward bimodal distributions, as shown in Figure 2.14. The majority of sites hit the balance limit, and a fraction of sites are constantly at a zero balance, with similar patterns at balance limits of 1 CPU-day and 25 CPU-days.

If banking is intended to solve the problem of demand before external events, such as conference submission deadlines, this bimodal wealth distribution suggests that auctions will fail, since all sites have the same amount of currency to bid on the same resource, and the utility of the resource presumably drops to zero after the paper submission deadline.

Conversely, when no external deadline exists, the bank balances provide little benefit, since PlanetLab already has most of its resources being allocated via Slop. Therefore, a banking scheme would allow all of the current demand to be satisfied, without providing any additional benefit beyond what is currently present.

If banking were employed to reduce the resources being allocated via Slop, then a policy decision has to be made regarding the testbed. From our earlier examina-

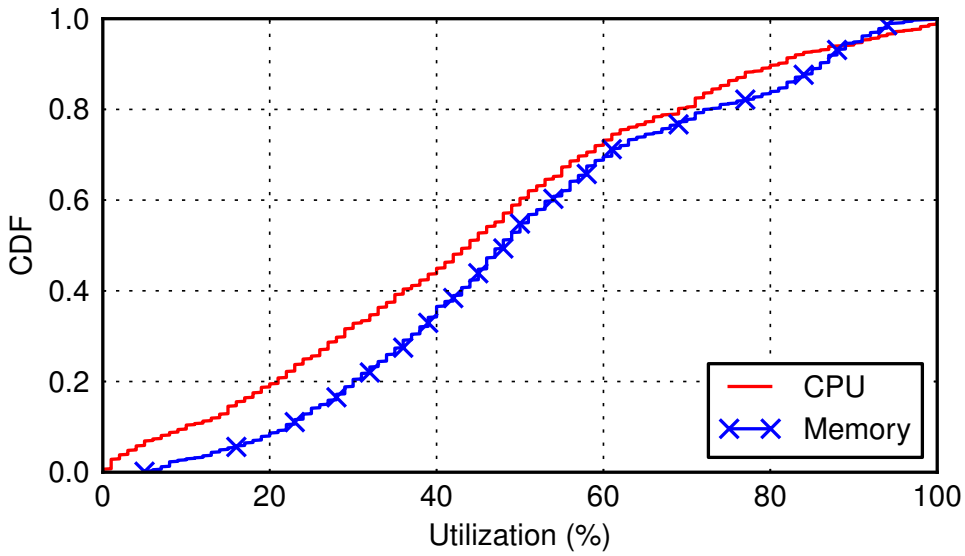


Figure 2.15: CDFs of average CPU/Memory utilization of all PlanetLab nodes in 2010.

tion of consumption, the Long-continuous slices were responsible for far less CPU consumption than the Long-intermittent slices. One may decide that Slop should be allocated preferentially to continuously-running services in order to increase the visibility (internally and externally) of PlanetLab. In any case, the decision becomes how to allocate the Slop, not how to use banking, suggesting that banking by itself provides little utility.

2.5 Workload Imbalance

Despite the availability of resources on PlanetLab, some level of contention does occur from the testbed being shared, leading to workload imbalance. In this section, we examine the workload distribution in 2010, and analyze the degree of imbalance in PlanetLab and its effects on the system. We analyze some reasons for the distribution, and explore solutions based on these observations.

By measuring average CPU and memory utilization by node, we can see a persistent difference during the year, as shown in Figure 2.15. The effects of this imbalance

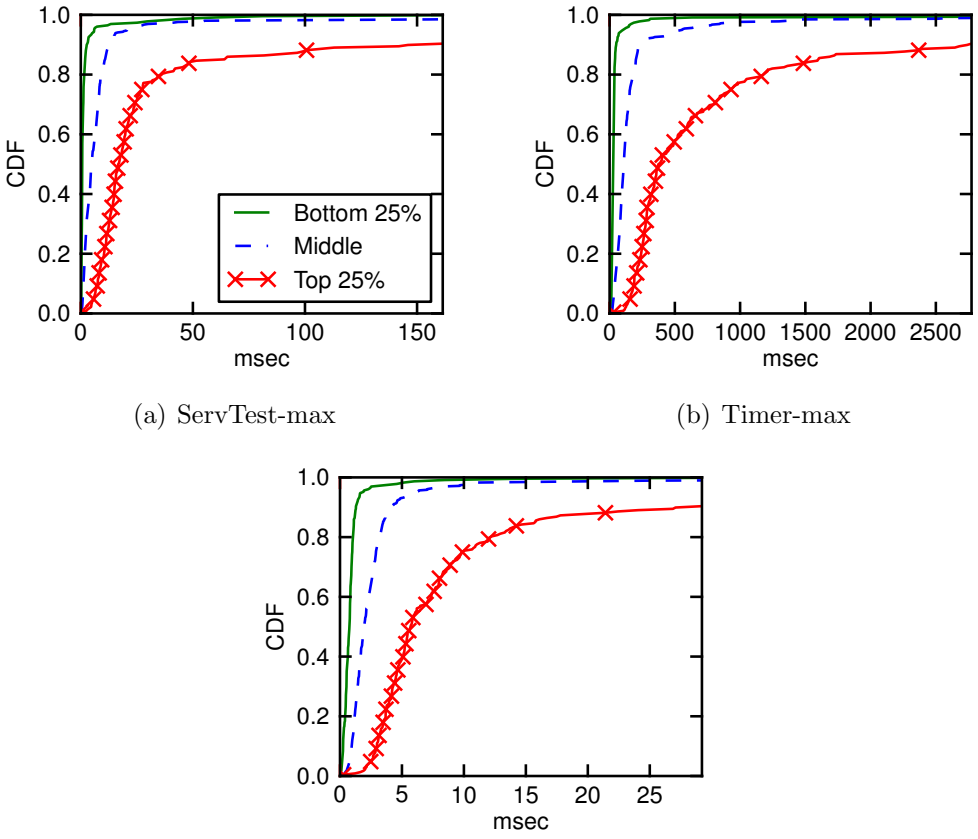
Field	CPU-Low	Med	High
ServTest-max	2.71	12.95	150.57
ServTest-avg	0.29	0.62	3.66
Timer-max	38.02	282.75	3247.94
Timer-avg	10.35	10.62	15.01
SleepLoop	1.28	4.15	27.72
SpinLoop	0.49	0.98	7.48

Table 2.3: The 90^{th} percentile values (in milliseconds) of the system lag metrics in nodes with low, medium, and high CPU load.

are also quantifiable, using CoMon's metrics regarding system lag and timing. These metrics are related to the responsiveness of networked systems, so load-induced timing problems would degrade PlanetLab's overall utility. The metrics used are

- **ServTest** – Measure latency to make a loopback connection and receive a byte from it. Calculate maximum and average values over previous 60 runs. Used to measure connectivity responsiveness.
- **Timer** – Measure latency in wake-ups from 10 msec sleeps. Calculate maximum and average values over last 60 runs. Used to measure load on the scheduler.
- **SleepLoop** – Run 11 spin-loops with 10 msec sleeps in-between. Measure gaps between the loops and calculate (max - min) of the gaps. Used to simulate the behavior of low-rate measurement activity.
- **SpinLoop** – Run 11 spin-loops without sleeps. Calculate a diff value like SleepLoop. Used to simulate the behavior of high-rate measurement activity.

We divide PlanetLab nodes into three groups based on their CPU load shown in Figure 2.15: top 25%, bottom 25%, and the remaining 50% of all nodes. Then we compare the values of the lag-related metrics between the groups (Figure 2.16). It is noticeable that the nodes with high CPU load show two orders of magnitude of increase in the metrics compared to other lightly loaded nodes. This latency may degrade responsiveness of network services or add measurement noise to running



(a) ServTest-max

(b) Timer-max

(c) SleepLoop

Figure 2.16: CDFs of system lags in nodes grouped by their average CPU load.

experiments. Table 2.3 summarizes the 90^{th} percentile values of all the metrics in each group. We observe similar results in the nodes grouped by their memory load. Such timing increases complicate experiment design and add noise to measurements. On the positive side, long-running services will have to develop mechanisms to deal with these issues, which are also likely to occur in the real world, thereby making their services more robust outside of testbed environments.

2.5.1 Origins of Imbalance

While one may expect that a certain amount of workload imbalance is naturally to be expected in a large testbed, we believe that the imbalance on PlanetLab has other more identifiable causes. Identifying these causes can help researchers in improving

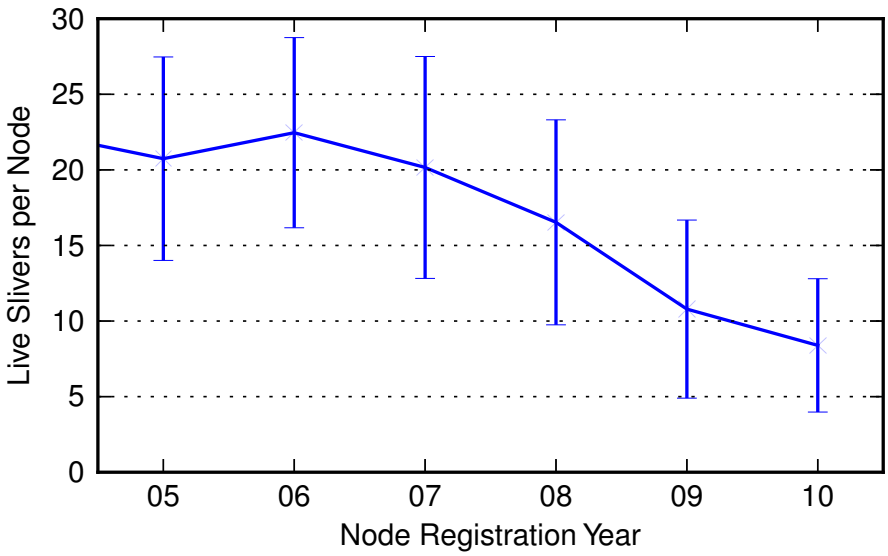


Figure 2.17: The number of live slivers per node in 2010. The recently registered nodes serve lower number of live slivers than older nodes. The error bars represent standard deviations.

the resources their experiments use, and it can help future testbed operators determine policies that would help alleviate imbalance.

One of our observations regarding PlanetLab nodes is that newer, more powerful nodes are often very lightly used, and that older, less capable nodes are in heavy demand, which is the opposite behavior of what one would expect from researchers seeking out the most available CPU resources. We can quantify this behavior by examining the sliver counts on different nodes. Figure 2.17 shows the number of live slivers per node, broken down by the year the node entered PlanetLab. We see that the older nodes have more than twice as many slivers as the newer nodes, and this metric understates the difference, since these sliver counts include many of the long-running services.

The same behavior is evident if we examine the type of machine involved, since older nodes would tend to have fewer CPU cores, and newer machines would have more cores. Figure 2.18 plots the numbers of slivers in nodes, based on the number of CPU cores per node. The trend is clear – not only do fewer slices run on the more

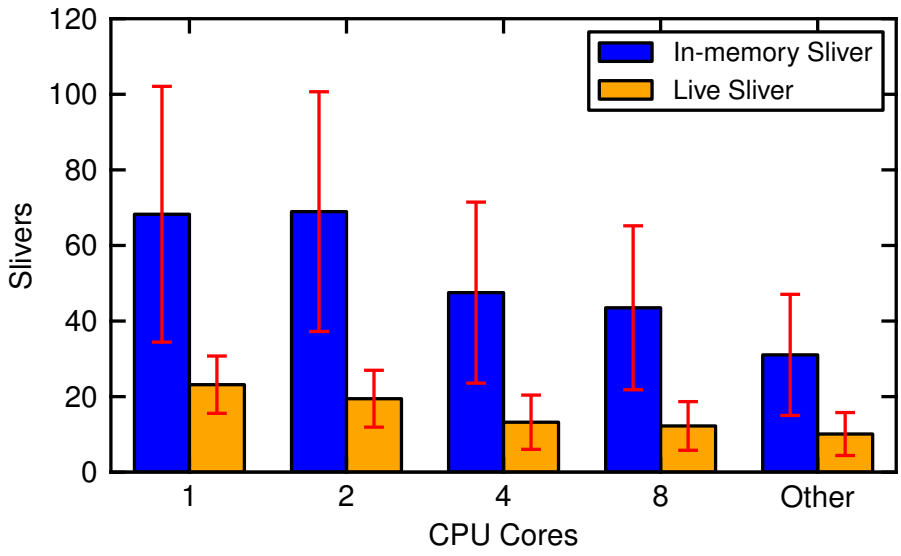


Figure 2.18: The distribution of slivers in nodes based on the number of CPU cores per node. The number of in-memory and live slivers shows a generally decreasing trend as the number of cores increases, which is responsible for some of the measured workload imbalance. The error bars represent standard deviations.

powerful machines, but even if we look at just the instantiated slices (the in-memory slivers), fewer of those exist on the more powerful machines. Even the most powerful nodes (labeled as "Other") are the least popular among the nodes. This breakdown is even more apparent when calculated the slivers per core, with the 8-core nodes serving only 1.53 slivers per core at a given time while single-core machines were busy with running 23.16 slivers.

We observe similar patterns in memory usage when compared to node memory size. Figure 2.19 shows the average memory usage of nodes, grouped by the memory size of the node. We see that nodes with more memory see relatively little extra usage of that memory. We believe that the reason that more memory is not used is because experiments that are deployed across multiple nodes have to plan for the lowest common denominator, and therefore restrict their memory usage to avoid being killed. At the same time, the reason we see any growth in memory usage on larger nodes is likely due to more memory-intensive experiments avoiding the smaller nodes,

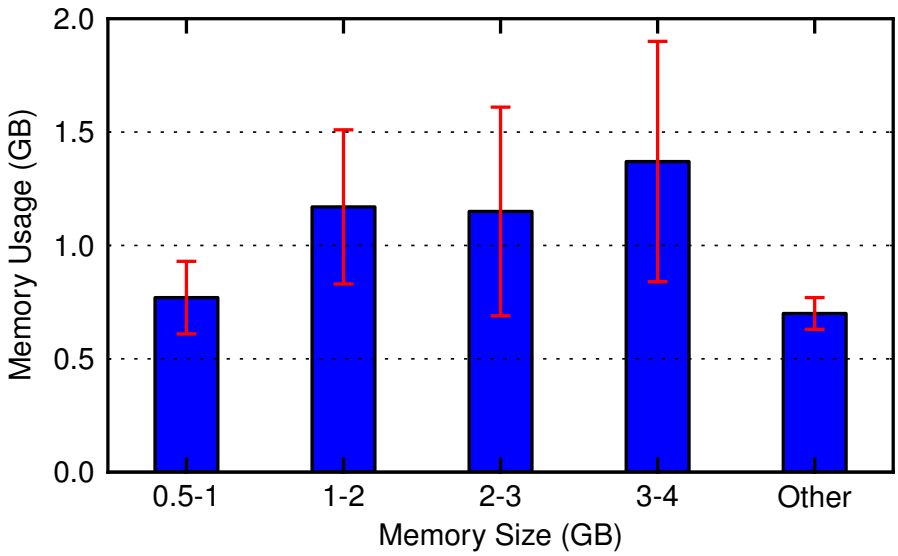


Figure 2.19: The memory usage by node memory size. The nodes with more memory see relatively little extra usage of that memory. The error bars represent standard deviations.

perhaps as a result of having found their slivers killed on those nodes. Regarding live sliver counts, we see some of the same trends observed for the node's CPU cores. The nodes with the largest memory space ("Other") host only 6.64 live slivers on average while the most memory-constrained nodes ("0.5 - 1GB") are busy with running 20.14 live slivers.

Several possible explanations can explain this behavior, ranging from the history of PlanetLab to human nature. One may expect that users obtain a list of working nodes, and do not regularly update their lists, leading to a bias against newly-introduced nodes. Older nodes may also reflect more established hosting sites that joined PlanetLab earlier since they had more active network research groups. These sites may have a larger user population keeping their nodes well-maintained, and may be connected to the Internet using better-quality links. Users may also flock to more busy nodes precisely because other users have found them desirable – knowing nothing else about two nodes, the one with more active users may actually be the better

node to use, because other users have already found the node to be more useful for their experiments.

2.5.2 Nodes with Failures

While it may be argued that there is little harm letting users flock to known-good nodes, it can also be useful to explore why users avoid other nodes. From a policy perspective, the testbed itself has reasons to encourage a flatter load balance, because it can increase the capacity of the testbed, reduce experimental variance, and alleviate congestion. The testbed operators may also want to ensure that the participating sites are really contributing resources of value, instead of pro-forma resources that are useless to the rest of the testbed.

Since resource pressure on a node is not a main concern for users, we focus on the types of failures that could significantly limit network experiments on the node. CoMon records a set of metrics about node health. Among them, we selected four fields that we believe users are likely to correlate with the stability and the quality of network connections that a node provides. They are DNS failure rates, provisioned bandwidth, system uptime, and availability of each PlanetLab node.

We define several failure modes for nodes. A node is considered to have a non-working DNS system if its DNS failure rate is over 90% on average in the node. A node has low bandwidth if its 90^{th} percentile of the achieved bandwidth in the node is less than 1 Mbps. For stability, we define a node to be unstable if its average system uptime is less than a week. Lastly, a node's availability is classified as low if the node was online for less than a month in total during 2010. These choices are meant to be conservative, in that users may still prefer higher quality values than our thresholds.

We classify PlanetLab nodes based on each failure mode that we define, and compare their popularity using sliver counts as a proxy. Figure 2.20 presents the average number of live slivers in both failed nodes and healthy nodes. In contrast

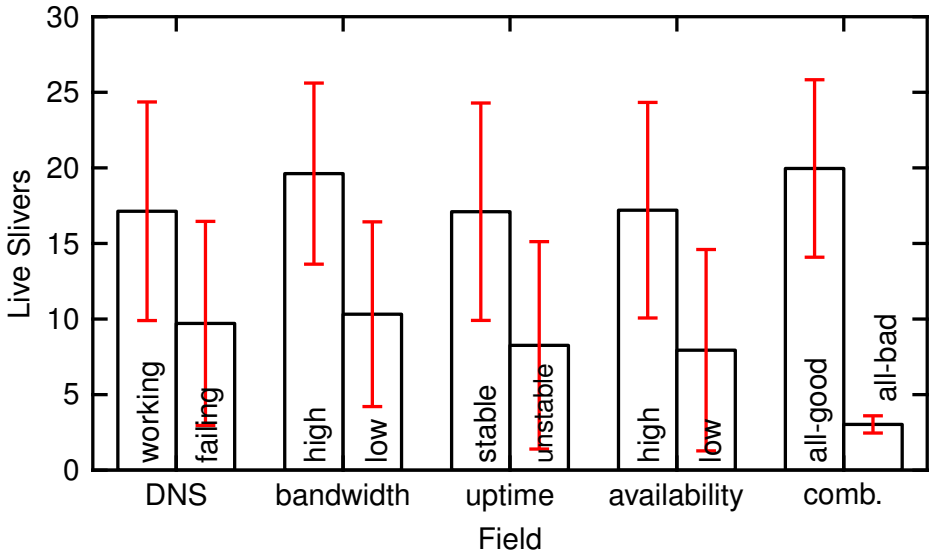


Figure 2.20: Popularity of nodes based on failure modes. PlanetLab users avoid nodes with high DNS failures, low bandwidth, and unstable operation. The error bars represent standard deviations.

to resource availability, we find that PlanetLab users do react to the failures in the nodes. We find that healthy nodes have twice as many live slivers compared to failing nodes, and since many of these slivers are due to infrastructure services, the difference in experimental slivers is likely to be even higher. The "all-bad" nodes in the combination failures (labeled as "comb.") represent nodes that exhibit all the four failures. Those nodes had only 3 live slivers while other healthy nodes hosted 18.5 live slivers on average. Figure 2.21 plots the distribution of node popularity measured as live slivers per node and in-memory slivers per node. We find that the all-bad nodes are in the bottom 3% of the unpopular nodes. Roughly 65% of nodes were "all-good" nodes that do not have any failures. Those nodes serve 20 live slivers on average, which is in the 64^{th} percentile in node popularity.

2.5.3 Alternative Experiment Placement

In the previous sections, we showed that PlanetLab users seem to stay with the nodes they have been known to work well over time while avoiding non-working nodes for

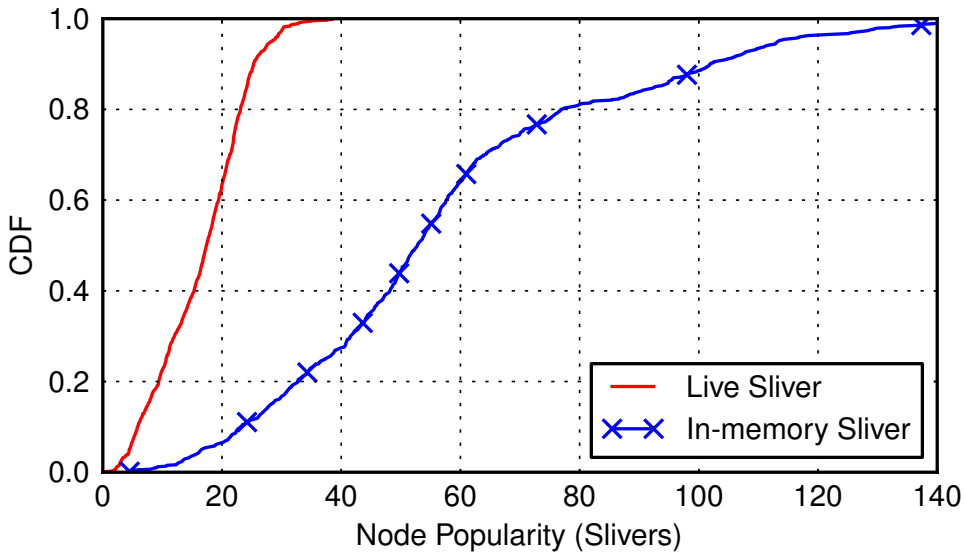


Figure 2.21: CDFs of node popularity. Each node's popularity is measured as its sliver count.

their experiments. However, this conservative placement strategy can collectively lead to inefficient resource allocation and undesirable system lags in the popular nodes. Also, the manual deployment prevents a new participant's nodes from being adopted by existing users, which can hamper the growth of the testbed in the long term.

We examine how the workload would change if researchers were to deploy their systems in a different manner. In PlanetLab, heavily loaded nodes are easily avoidable because there are available services to help users identify such nodes. CoMon provides a set of interfaces to allow users to pick lightly loaded nodes based on web-based queries. Similarly, CoMon provides interfaces to filter out failing nodes in using various metrics. Some available execution management systems [3, 5] locate resources based on high-level queries in XML given by users.

We consider a "what if" scenario that all PlanetLab users query CoMon-like monitoring systems to find the set of lightly loaded *all-good* nodes when they deploy their experiments. We simulate the scenario using the CoMon data of 2010. We assume that a sliver is placed on the node selected by a monitoring service only when it

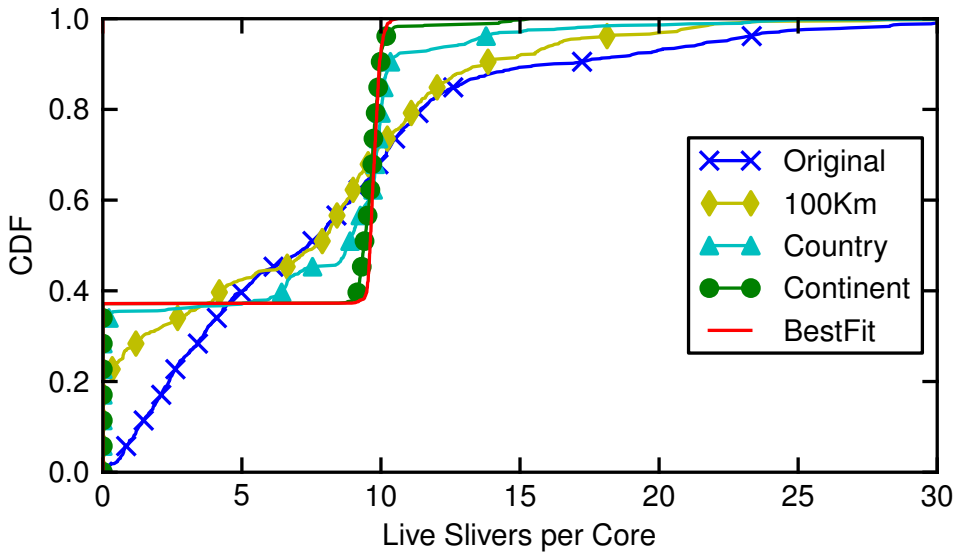


Figure 2.22: CDFs of live slivers per core in simulations of alternative node placement policies. Since lightly loaded and all-good nodes are selected, the workload is well balanced among nodes while any undesirable failures are avoided.

is started, and that continuously live slivers do not migrate between nodes in our simulation.

Figure 2.22 plots the distribution of slivers with several alternative service-placement policies. *Original* plots the number of live slivers per core that we observed in the datasets. *BestFit* represents a policy that places a new sliver in the least loaded all-good node regardless of its location. However, it is not always desirable to deploy services in this way, because users may want to deploy their experiments at a certain range of network vantage points. *Continent* and *Country* represent policies that find nodes from the same continent or the same country as its original node in the dataset, respectively. Lastly, since some countries have a large number of nodes, we simulate a policy to find a node for a sliver within a configurable distance, 100 Kilometers, from its initial node. We compare against an ideal policy, BestFit, in which all good nodes host an equal number of slivers, and any nodes that exhibit failures (201 in total) are avoided entirely.

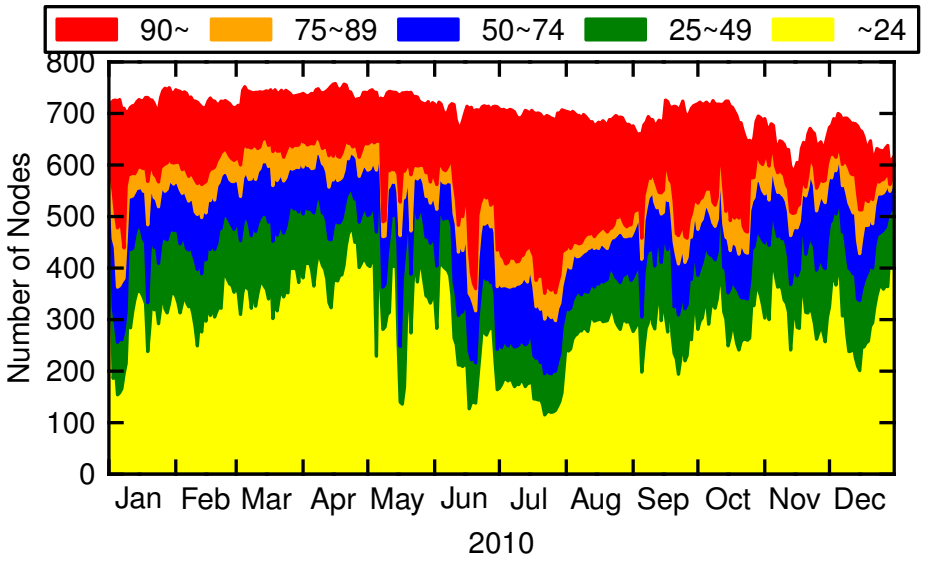


Figure 2.23: The distribution of per-day CPU loads on PlanetLab nodes in 2010. Each day, nodes are divided into five categories according to their per-day CPU usage. The nodes with more than 90% CPU usage account for up to 49% of all PlanetLab nodes (July 22).

Our simulation result shows that the dynamic experiment placement could greatly improve the load balancing in PlanetLab while still largely avoiding problematic nodes. The 90^{th} percentile of per-node live slivers decreases from 16.1 to 13.7 (in 100 km) and 10.3 (in Country). As the placement restrictions are loosened, not only does it become possible to more evenly distribute load, but it also becomes more likely that the bad nodes are avoided entirely. The 100-km policy still allocates a substantial fraction of nodes from the bad set, whereas continent-level placement avoids them almost entirely. If a user wants to use a set of specific nodes, she should be able to deploy her service on the nodes, but many other users could benefit from the intelligent service placement. The primary reason for the improved load-balancing is that most experiments are short-lived (Section 2.3), so the dynamic experiment placement can help spread well the load over available nodes.

2.6 Policing of Slices in PlanetLab

In this section, we show that PlanetLab's resources are affected by a few problematic slices. We examine the impact of those slices on PlanetLab and consider the implications for policing of resource usage in the testbed. We also explore how PlanetLab's workloads would change if stricter form of policing is introduced in PlanetLab.

2.6.1 Spin-loop Slices in PlanetLab

PlanetLab is a shared infrastructure, and while it provides some mechanisms to prevent experiments from interfering with each other, it also relies on the cooperation of researchers. For example, local tests and incremental rollouts are recommended before a new service is fully deployed in PlanetLab [74]. We observed that most slices follow this practice well in Figure 2.6. To minimize centralized control, PlanetLab typically does not police resource usage of a slice unless there are significant risks of system crashes or security concerns that need to be addressed immediately.

However, some experiments may behave poorly because of design or implementation flaws, and negatively affect many other well-behaved experiments. We analyze the problem using CoMon data collected in 2010. We divide nodes into several categories according to their per-day CPU usage, and observe the distribution of the nodes over time. Figure 2.23 presents the distribution of per-day CPU loads on PlanetLab nodes in 2010. In this analysis, we define a node to be *overloaded* on a given date if its per-day CPU usage is over 90%, and up to 49% of all live PlanetLab nodes are overloaded. The number of overloaded nodes varies over time, but we did not find any noticeable correlation between the numbers of overloaded nodes and live slices a day (the graph is omitted for brevity). This implies that the overloaded nodes are not caused by the increase of active users.

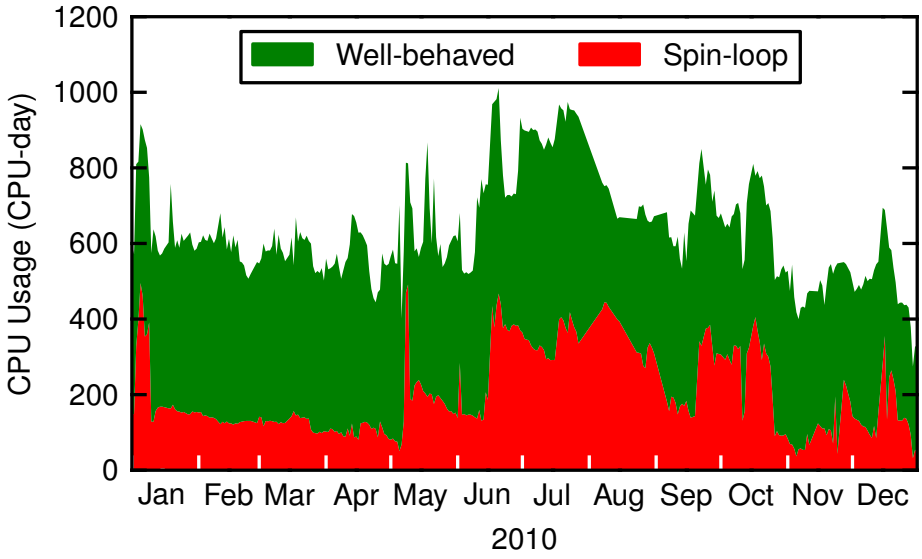


Figure 2.24: The CPU consumption of spin-loop slices in 2010. Although there are only a few spin-loop slices (4.8 slices among 152.7 live slices per day), the average CPU consumption of the spin-loop slices accounts for 31% of the total CPU usage of all slices.

The largest testbed-wide CPU consumers are often a few slices that use many aggregate CPU cycles without generating network traffic. We define a *spin-loop sliver* as a sliver that consumes more than 20% of a CPU and has an average bandwidth consumption (Tx + Rx) below a minimum value, 1 Kbps. We classify a slice as a *spin-loop slice* if the majority of its slivers are spin-loop slivers.

We believe that most of the spin-loop slices are caused by unintentional mistakes in their designs or implementations, but it is also possible that some slices process meaningful jobs without generating network traffic. We found that a few slices used PlanetLab for running simulators or data analysis on the nodes such as social network data analysis. However, PlanetLab is designed for network experiments, not resource-intensive computations. In our analysis, we classify the slices as spin-loop slices because the simulation experiments or grid computations are not consistent with the testbed's purpose.

Figure 2.24 presents the CPU consumption of the spin-loop slices in 2010. We find that the problematic slices accounted for on average 31% of the total CPU across all

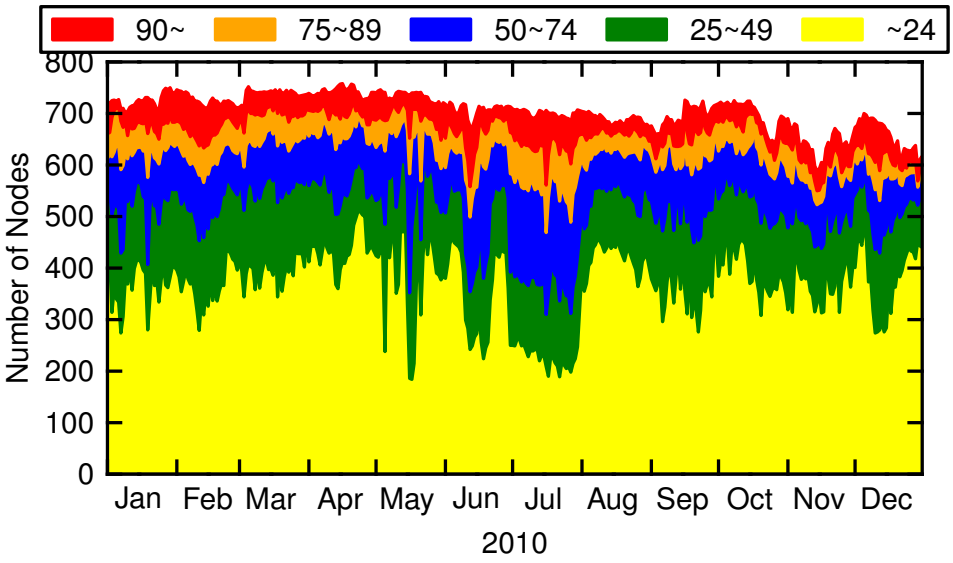


Figure 2.25: Time series of the updated distributions of per-day CPU loads after pruning spin-loop slices. The number of overloaded nodes is reduced by 71% (150 to 43) on average.

of PlanetLab. Since spin-loop slices average 4.8 per day among the 152.7 live slices per day, the 31% resource consumption is significantly large. Also, we see the CPU consumption of the spin-loop slices is the main cause for the fluctuation of the overall workload in the testbed.

2.6.2 Pruning Spin-loop Slices

We simulate pruning the spin-loop slices to measure their impact on other slices. For a spin-loop slice, we subtract its sliver's CPU consumption from the node's CPU usage. If a spin-loop sliver had been pruned by PlanetLab, more CPUs might have been available to non-spinloop slivers in the node. Therefore, for each non-spinloop sliver, we select the larger of its CPU usage in a day or the median of its CPU usage throughout the year, to recompute the node's CPU usage after pruning.

Figure 2.25 plots the updated distribution of CPU load after spin-loop slices are pruned by PlanetLab. We find that the number of overloaded nodes is reduced from 150 to 43 nodes by policing only 4.8 spin-loop slices a day on average. It is notable

that there are still some overloaded nodes even after pruning all identified spin-loop slices on PlanetLab nodes. We examine the nodes in order to understand what other factors made them remain overloaded. We find that 56% of the overloaded nodes are single core machines, which represent only 13% of all PlanetLab nodes.

2.7 Related Work

Several resource management frameworks have been proposed for PlanetLab-like federated distributed computing infrastructures. In Sharp [29], multiple autonomous parties can exchange their resources using tickets. A ticket represents the holder's claim over a certain amount of resources in other peers, which can be issued, delegated, and redeemed in a cryptographically secure manner. Millennium [21], Mirage [20], Tycoon [44], Amoeba [75], and Bellagio [10] propose market-based mechanisms for trading resources in an economically efficient way. The systems focus on maximizing the values delivered to users by providing a means to express their valuation of resources. In Bellagio, participating users receive virtual currency budgets based on their resources that they contributed, and submit their preferences in the form of auction bids. The market-based systems are not widely deployed in PlanetLab because user valuation of resources is useful in the systems where resource demand exceeds resource supply (e.g., sensor network testbed). We expect that our analysis results will provide insights into user behavior in the federated infrastructures, which is required for designing similar economic resource allocation models.

Previous studies in traditional cluster resource management largely focused on resource utilization for compute-intensive applications in time-sharing or batch-queue systems. The job scheduling and resource allocation in the systems are designed to improve performance metrics such as throughput and mean response time. Distributed batch queue systems (Condor [47], Matchmaking [67]) provide resource shar-

ing across loosely coupled pools of distributively owned machines. Load balancing systems (MOSIX [13], LSF [62]) balance CPU load across nodes in cluster by actively migrating processes across cluster machines. However, these cluster resource management systems do not address non-aggressive user behavior in wide-area network testbeds that are much different from compute clusters.

Our work relates to research projects that help PlanetLab users monitor and locate resources in the testbed. SWORD [5] is a resource discovery service deployed in PlanetLab. In SWORD, users describe desired resources such as per-node characteristics in XML and submit the queries, and then the service locates an appropriate set of resources for the user based on the given specification. CoMon [55] provides a comprehensive view of statistics about every node and slice in PlanetLab. It also provides a mechanism to select nodes based on queries provided by users. Also, several execution management systems [3, 9, 18] are available to provide GUI interfaces to help users deploy and monitor their systems across multiple remote nodes.

Chapter 3

Lsync: Low-latency File Transfer System

3.1 Introduction

In the previous chapter, we showed that most PlanetLab experiments are short-lived but often expand to a large fraction of available nodes in the testbed. Given the usage pattern, reducing deployment delay is required for improving programmer productivity in wide-area testbeds. Besides testbeds, low-latency file transfer is essential for many wide-area systems that want to maintain a close control over their remote nodes. These systems frequently distribute new configurations [71], coordinate task lists among multiple endpoints [79], and optimize system performance under dynamically changing network conditions [52]. All of the scenarios involve *latency-sensitive synchronization*, where the enforced synchronization barrier can limit overall system performance. 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.

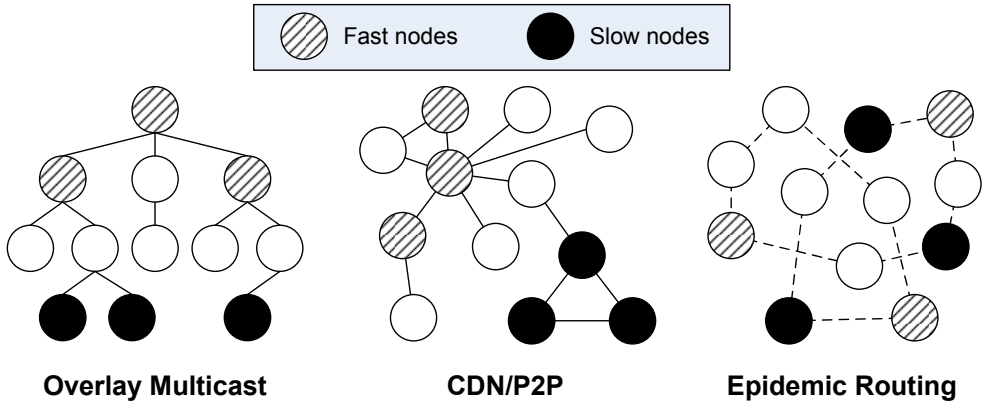


Figure 3.1: Slow Nodes in Overlay – Peering strategies in scalable one-to-many data transfer systems are not favorable to slow nodes.

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 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 transfer [15, 19, 28, 43, 56, 60], they largely ignore the latency issue because resource 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, existing systems are not favorable to slow nodes (Figure 3.1). For instance, many overlay multicast systems attempt to place well-provisioned *fast nodes* close to the roots of their multicast trees while pushing slow nodes down the trees. Likewise, the peering strategies used in CDN/P2P systems prevent slow nodes from peering with fast nodes. The random gossiping in the epidemic routing protocols helps slow nodes peer with fast nodes, but only for short-term periods. These peering

strategies produce long completion times with high variations because they make slow nodes download more slowly. However, it is common that existing services rely on one of the data transfer schemes for coordinating their remote nodes.

In this chapter, we explore general file transfer policies for latency-sensitive synchronization with the goal of minimizing completion time in a *fixed 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 performance problems in the overlay at runtime. The server allocates its bandwidth to slow nodes in a manner favorable to slow nodes while synchronizing other nodes through the existing overlay. This server-assisted synchronization reduces the tail latency caused by slow nodes without sacrificing scalable data transfers in the overlay, which drastically improves the completion time and achieves stable file transfer.

We design and develop Lsync, a low-latency one-to-many file transfer system. Lsync can be used as a 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 nodes, 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.

Lsync implements the techniques that we develop for reducing latency in this work, including: (1) unified node selection with synchronization that outperforms other approaches; (2) dynamic node scheduling to give preference to bottlenecked nodes; (3) one-to-many file distribution service characterization using a black-box

approach; and (4) adaptively choosing the best synchronization mechanism, which depends heavily on the environment and the data to be synchronized.

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 [2], Lsync makes most PlanetLab nodes full synchronized throughout the experiment.

The rest of this chapter is organized into following sections. In Section 3.2, we describe the model and assumptions for our problem. Based on the model, we discuss how to allocate the server's bandwidth to slow nodes in Section 3.3. In Section 3.4, we describe how to divide nodes between the server and the overlay in a way to minimize completion time. We describe the implementation in Section 3.5 and evaluate it on PlanetLab in Section 3.6.

3.2 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 3.2. 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 3.4.5, 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 3.5.

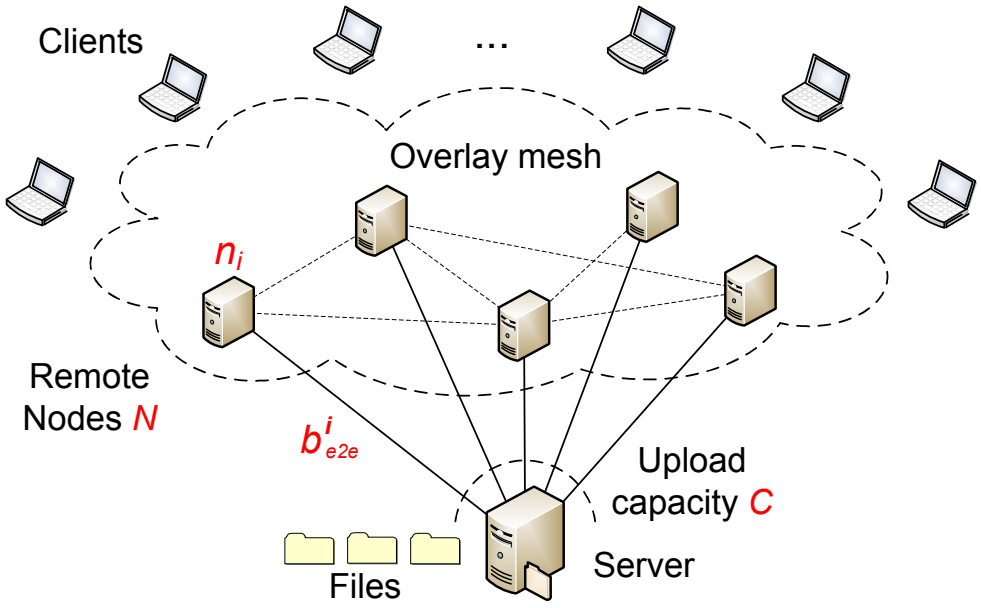


Figure 3.2: Synchronization Environment – the server has files to transfer to remote nodes with low latency. The remote nodes construct an overlay mesh for providing a scalable data transfer service to external clients.

Many distributed systems construct overlay meshes 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 partition the 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.

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

Examples of current systems where our intended approach may be applicable include distributed caching services [6, 52], WAN optimization appliances [22, 68], distributed computation systems [70], edge computing platforms [24], wide-area testbeds (PlanetLab, OneLab), and managed P2P systems [58, 59, 77]. 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 Lsync uses different file transfer strategies for different 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. However, it is not always feasible to wait until 100% of nodes are synchronized. In wide-area systems, it is common that some nodes are left disconnected for extended periods of time. Therefore, Lsync should also support partial update scenarios. Users can specify a certain fraction of nodes that need to be synchronized quickly. We name the input parameter *target synchronization ratio* denoted by r for the rest of this chapter. Given r, Lsync will automatically pick the best set of nodes that can deliver the lowest latency to users, and focus available network resources on the selected nodes while synchronizing remaining nodes in the background. Our evaluation results show that this late-binding of nodes significantly reduces latency. This partial update is useful for interactive development in wide-area testbeds, incremental rollouts of new configurations, and maintaining quorum in wide-area systems. For global updates, users can simply set r to 1. Likewise, when users want a partial update for a specific set of remote nodes, they can set r to 1 for the manually selected nodes.

Parallel transfer control The server limits the number of simultaneous transfers in order to avoid self-congestion and associated problems stemming from server overload [4, 65]. While many existing tools have static configuration for maximum con-

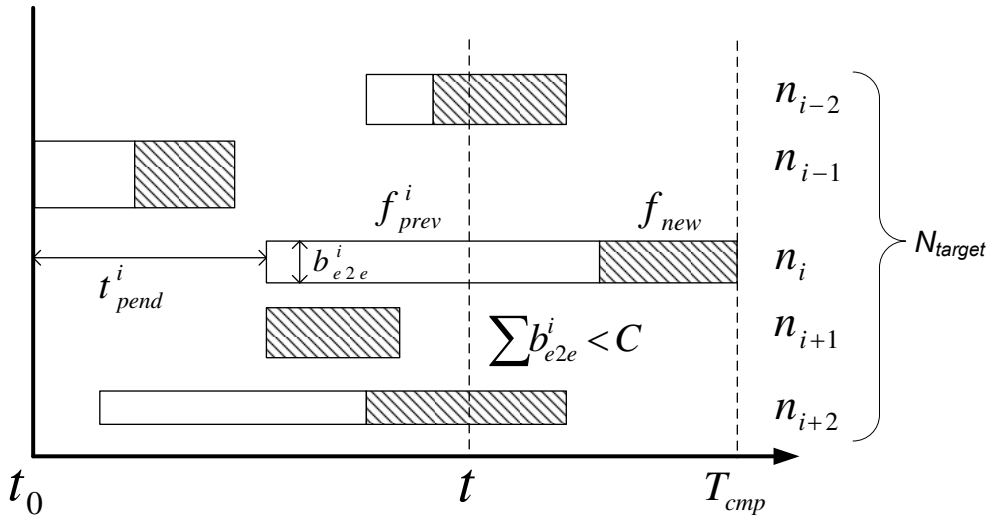


Figure 3.3: End-to-End Synchronization – The completion time T_{cmp} is determined by the node with the latest finish time among the target nodes N_{target}. The server can control t^i_{pend} and N_{target} with node scheduling and node selection policies.

nections, when beneficial, we expect to dynamically adjust the number of end-to-end transfers such that the total bandwidth demand does not overwhelm the upload capacity of the server.

3.3 Server Bandwidth Allocation

Lsync's file transfer policy combines multiple factors that contribute to the overall latency reduction. Lsync exploits the server's available bandwidth to speed up synchronizing remote nodes. In this section, we begin by examining the effects of the server's bandwidth allocation policies on completion time. This end-to-end synchronization scenario serves as a useful starting point both because of its simplicity as well as the fact that many tools operate in this manner [3, 54, 76]. We will later extend our discussion to incorporate more scalable data distribution systems in Section 3.4.

Figure 3.3 shows an example of end-to-end synchronizations where 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 $|f_{new}|$, 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 [4].

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

$$T_{cmp} = t_0 + \max_i \left(t^i_{pend} + \frac{|f^i_{prev}| + |f_{new}|}{b^i_{e2e}} \right) \tag{3.1}$$

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 on PlanetLab to examine their effects on latency.

3.3.1 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. The proof of NP-hardness is simple. We reduce the bin packing problem to the problem of minimizing completion time in node scheduling in Figure 3.3. The bin packing problem is defined as follows: Given a set of items of different sizes, determine the minimum number of bins of capacity V needed to pack all the items. Let s_i be the size of a given item. We set C, f_{new} and f^i_{prev} to V, 0 and

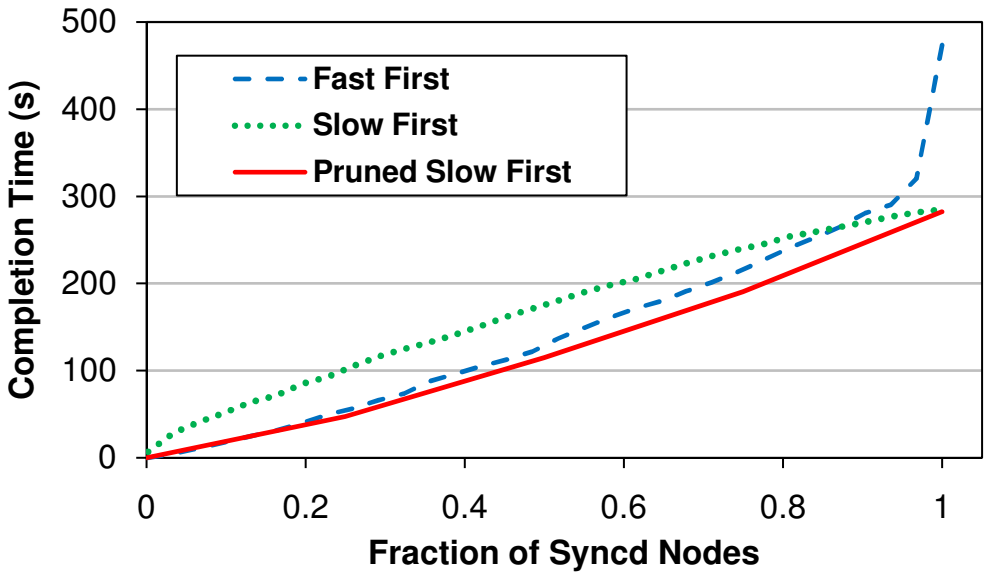


Figure 3.4: Node Scheduling and Node Selection – Pruned Slow First captures both the initial speed advantage of Fast First, as well as the total overall advantage of Slow First.

s_i, respectively. Then the NP-hard bin packing problem is reduced to the problem of finding the scheduler that can minimize the completion time T_{cmp} in Figure 3.3. Hence the node scheduling problem is NP-hard.

We compare two basic scheduling heuristics, Fast First and Slow First. The *Fast First* policy is similar to Shortest Remaining Processing Time (SRPT) scheduling in that the server's resource is allocated to the node that has the shortest expected completion time. SRPT is known to be optimal for minimizing mean response time [12, 33]. The *Slow First* policy is the opposite of the Fast First policy, and selects the nodes with the longest expected completion time when the server has available bandwidth.

We measure the performance of 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 3.4. 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 the 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 Slow First scheduling can mask their effects on latency. Another implication of the result is that offline optimization will provide little benefit to Lsync in this scenario. When Slow First is used, the server's bandwidth is underutilized only for last 0.5 seconds. This means that no scheduler can outperform the Slow First policy 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 address.

3.3.2 Node Selection

In this section, we examine the effect of node selection on latency. 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 the best set of nodes that can deliver the lowest latency to users.

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 their 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 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 3.4 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 instance, configuration management systems [71] continuously generate new updates for remote nodes. Also, many distributed systems expect all nodes to share global state, including those where all nodes are expected to know each other [15, 25] and the systems that periodically distribute monitoring results [49, 79]. 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 an *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 3.5 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 0

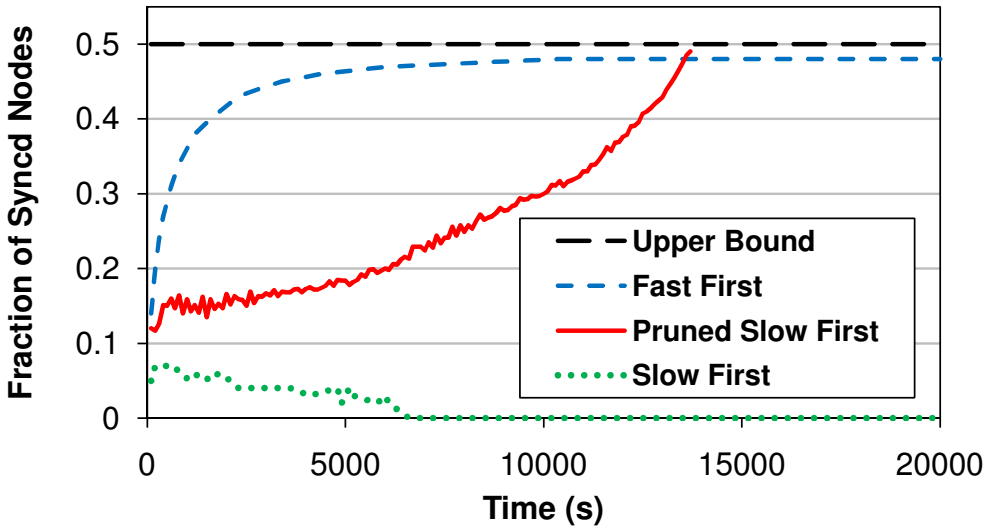


Figure 3.5: Synchronizing Frequent Updates – While Fast First synchronizes quickly at first, Pruned Slow First actually reaches the upper bound more quickly.

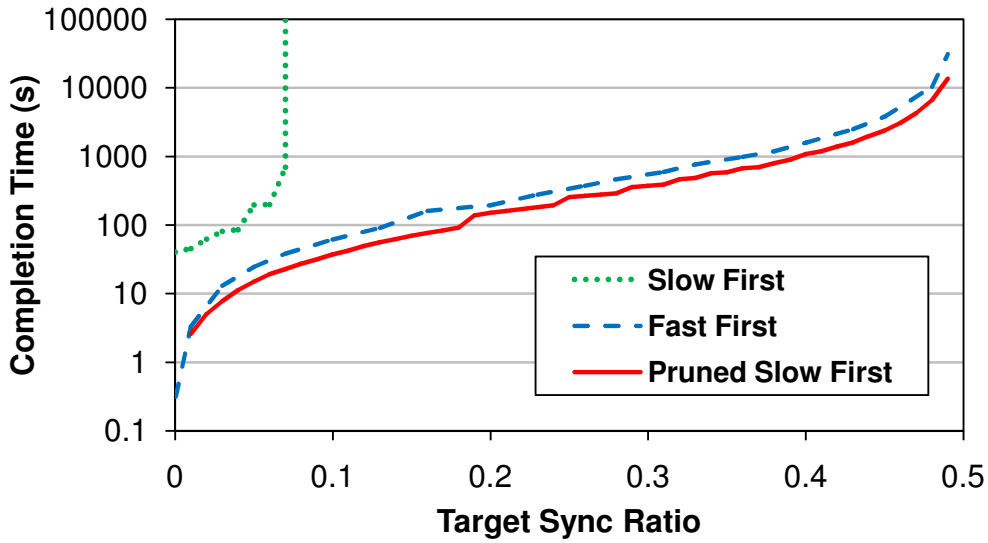


Figure 3.6: Synchronization Latency for Frequent Updates – While Slow First leads to failure, integrating node selection with the Slow First scheduling reduces latency for all target ratios (y-axis is in log-scale).

near 6800 seconds because it gives high priority to the nodes that can never be fully synchronized. 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 3.5, 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 3.6, we show the completion time of each policy for a range of feasible r values. The Pruned Slow First policy 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.

3.4 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 call CDN/P2P systems for the rest of the chapter. Many large-scale distributed services construct overlay meshes for scalable data transfers to external clients, and often use the overlay for internal data dissemination to remote nodes as well [71]. In this section, we explore how to leverage the overlay mesh to reduce synchronization latency without changing its behavior.

3.4.1 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, CDN/P2P systems typically have diverse peering strategies and dynamic routing

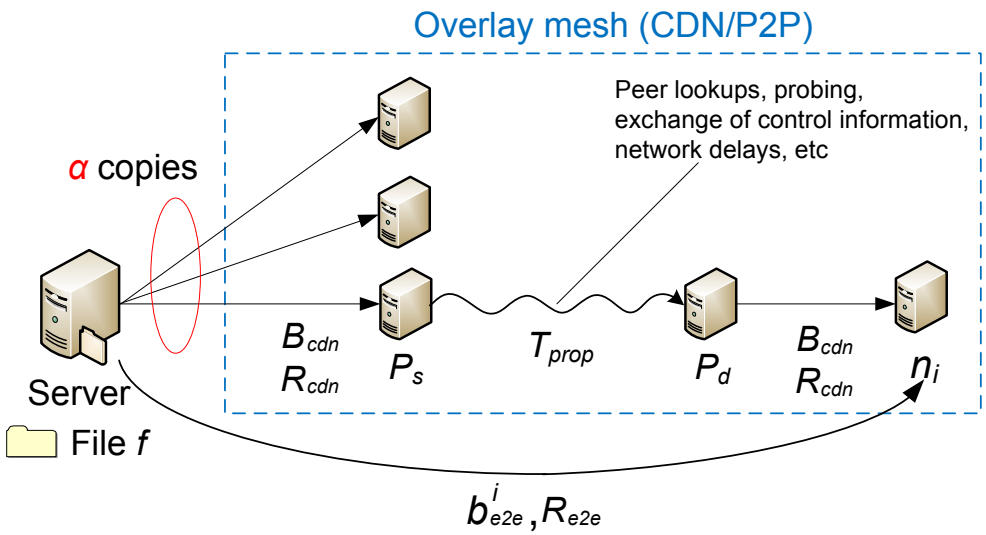


Figure 3.7: Startup Latency in CDN/P2P – To leverage a given overlay system, Lsync estimates the startup latency for fetching a new file f from the server and propagating to the remote nodes n_i in the overlay.

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 3.7 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 in real deployments 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 3.6.2. 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.

3.4.2 Completion Time Estimation

To partition the workload across the server and the overlay, 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

$$\delta = 2 \cdot R_{cdn} + T_{prop} \tag{3.2}$$

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,

$$T_{cdn} = R_{e2e} + \delta + \frac{f_s}{B_{cdn}} \tag{3.3}$$

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

$$T_{e2e}(r) = R_{e2e} + \frac{N_s \cdot r \cdot f_s}{C} \tag{3.4}$$

where C is the server's upload capacity.

3.4.3 Selective Use of Overlay Mesh

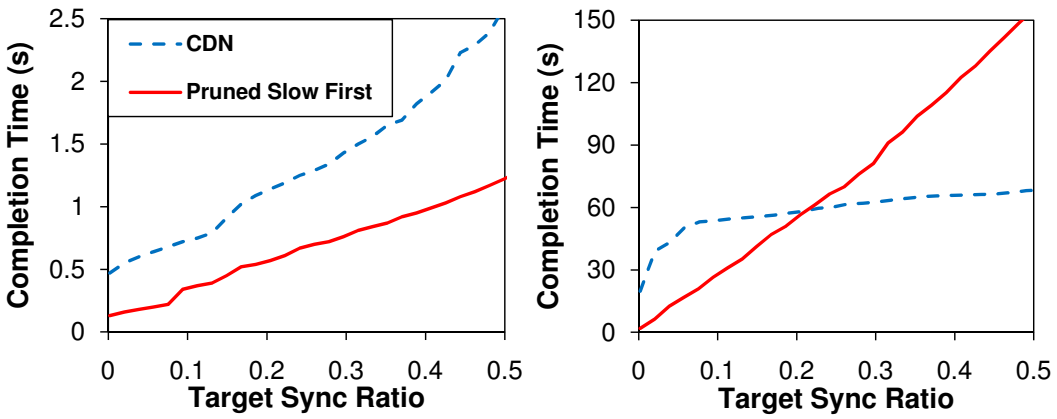
If a given CDN/P2P system has high startup latency, it will outperform end-to-end transfers only when its bandwidth efficiency can outweigh the cost. After the server estimates T_{cdn} and $T_{e2e}(r)$, the server can dynamically choose between end-to-end transfers and the overlay mesh to get better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

To get a more accurate estimation of the completion time, we extend T_{cdn} in (3.3) 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

$$T_{cdn}(r) = R_{e2e} + \delta^r + \frac{f_s}{B^r_{cdn}} \tag{3.5}$$

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.

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 fol-



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

Figure 3.8: 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.

lowing conditions is met.

$$f_s < \delta^r \cdot \frac{C \cdot B^r_{cdn}}{N_s \cdot r \cdot B^r_{cdn} - C} \tag{3.6}$$

$$B^r_{cdn} < \frac{C}{N_s \cdot r} \tag{3.7}$$

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 the CoBlitz CDN [56] on PlanetLab. Beyond PlanetLab, CoBlitz has been used in a number of commercial trial services [23], and we believe CoBlitz represents one of the typical CDN services currently available. For end-to-end transfers, we use the Pruned Slow First policy for allocating the server's bandwidth.

Figure 3.8 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 3.8 imply that Lsync will need different strategies for different file sizes to address the tradeoffs between the startup latency and the bandwidth efficiency of the overlay mesh.

3.4.4 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 additional 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 (3.5), 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 a *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 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

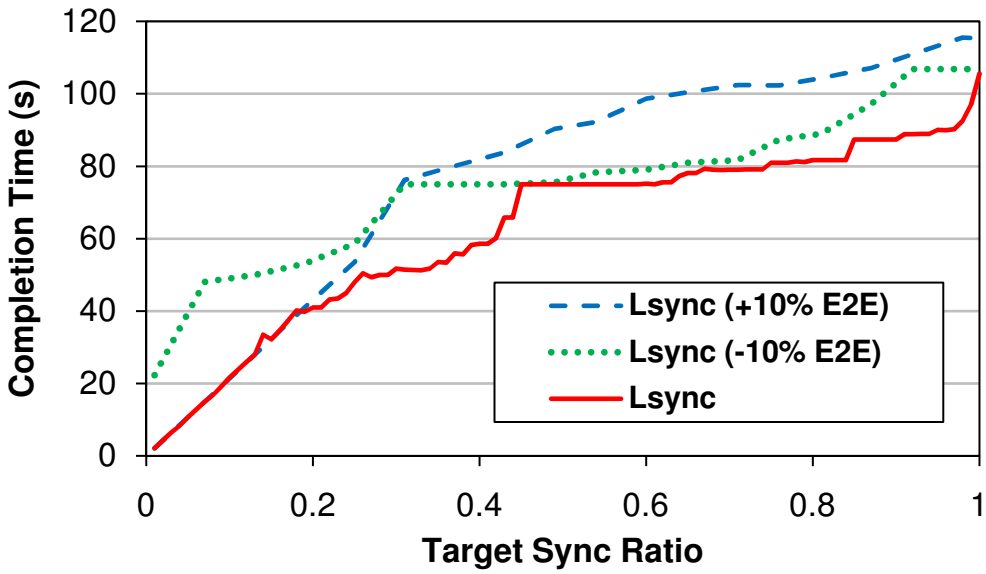


Figure 3.9: Optimality of the Division – The split between overlay and E2E is not improved by moving some nodes to the other mechanism, suggesting that Lsync's split is close to optimal.

bandwidth, the end-to-end completion time is

$$T_{e2e}(r_{e2e}) = R_{e2e} + \frac{N_s \cdot r_{e2e} \cdot f_s}{C - B_{load}} \tag{3.8}$$

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 3.9 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.

3.4.5 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.

3.5 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. The Lsync daemon implements the file transfer policy described in the previous sections.

Lsync configuration is relatively simple, with the most basic case consisting of the user providing one or more local directories to be synchronized, and a set of remote nodes that will receive the directories. The node list can simply be a long list of nodes and one synchronization ratio, or can be broken into multiple node groups with individual synchronization ratios. In this manner, simple policies such as "half of all nodes" can easily be specified. If a particular node or set of nodes must always

be in the synchronization group, it can be placed in a group with a synchronization ratio of 1, allowing manual node selection.

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 [66] 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 end-to-end transfer, or by copying the log file to a Web-accessible directory and informing the remote daemons to grab the file via the overlay. Once the remote daemon receives a log file, it applies the necessary changes to its local copies of the file.

3.6 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.

3.6.1 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 ratio of 0.98 to account for nodes that may become unreachable during our experiments. Lsync attempts to achieve the given target ratio quickly while leaving other available nodes synchronized via overlay in the background.

Since Lsync runs in the background, it can store the history of its activity to use when performing calculations. This history is used to determine parameter values for its models, especially for end-to-end and overlay bandwidths. Lsync uses an Exponentially Weighted Moving Average (EWMA) for updating the parameters because history-based prediction is much more accurate than formula-based prediction when there is a short history of recent TCP transfers along the same path [34]. For the experiments, we use passive bandwidth measurements from repeated network transfers by the clients, which provides enough history of recent TCP transfers.

3.6.2 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 for the typical workload in existing CDN/P2P systems.

Using our black-box model described in Section 3.4.1, we measured parameters, δ^r and B^r_{cdn}, for four existing systems. We used two academic CDNs running on Plan-

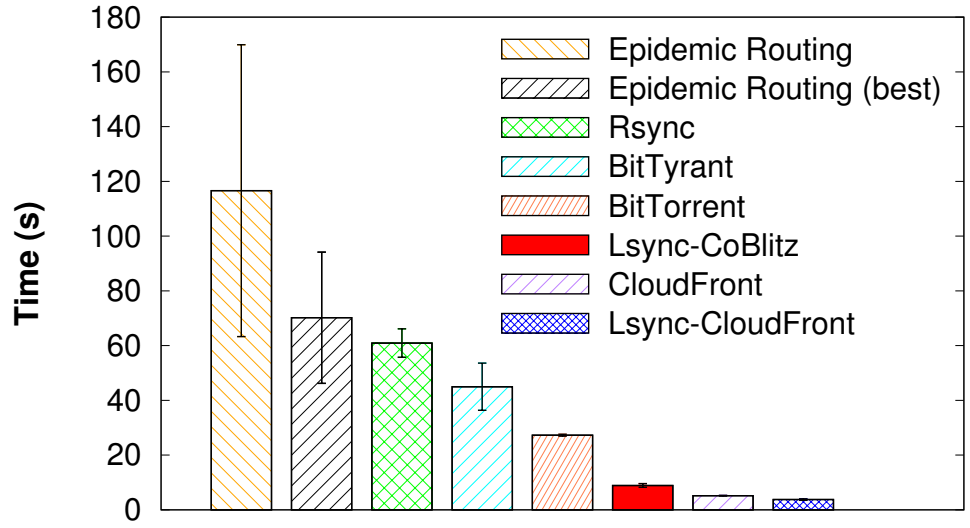
Systems	δ^r	B^r_{cdn}	Division Ratio	
			E2E	Overlay
CoBlitz	1.4	1.2	0.24	0.76
Amazon CloudFront	2.9	6.4	0.06	0.94
Coral	7.6	0.6	0.52	0.48
BitTorrent	29.7	4.7	0.30	0.70

Table 3.1: 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.

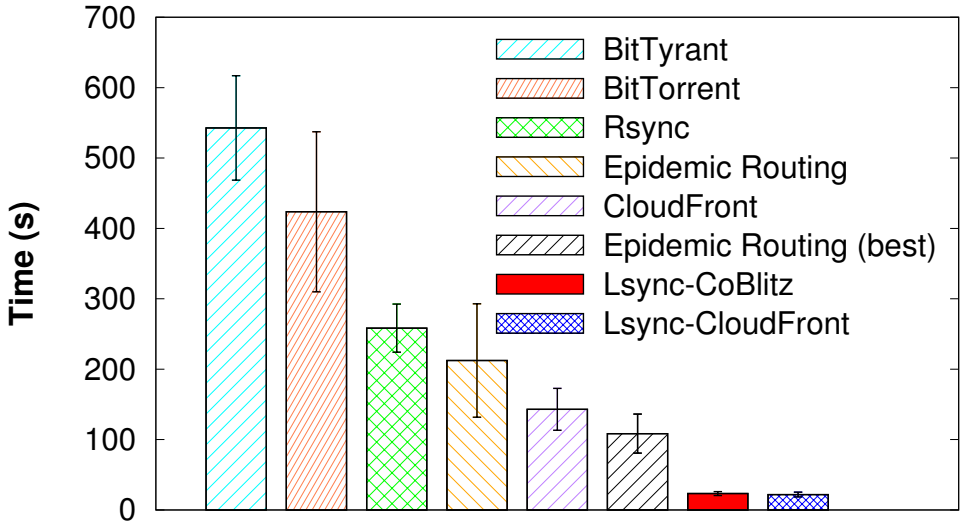
etLab, CoBlitz [56] and Coral [28], and a commercial CDN, Amazon CloudFront [6]. We also deployed BitTorrent [16] on PlanetLab nodes for the measurement.

Table 3.1 shows the setup costs and bandwidths of the four 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 four 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 ratio 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. CoBlitz assigned 76% of the target nodes to the overlay group. For Coral, Lsync assigns nodes roughly equally between the overlay group and the end-to-end group. Although BitTorrent has the highest latency, it shows a similar distribution to CoBlitz because nodes have high overlay bandwidth after the startup latency. For CloudFront, Lsync assigns most of the nodes to the overlay group to exploit the high bandwidth in the well-provisioned CDN nodes. However, for small file transfers, Lsync still opts to use direct end-to-end transfers.



(a) Target Sync Ratio 0.50



(b) Target Sync Ratio 0.98

Figure 3.10: 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).

We select CoBlitz for the rest of our experiments, mostly due to its low latency, but we will also integrate Lsync with CloudFront from some comparisons.

3.6.3 Comparison with Other Systems

We compare Lsync with different types of data transfer systems. We transfer the CoBlitz web proxy executable file (600 KB) to all PlanetLab nodes using each of the systems, and measure completion times (Figure 3.10). 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* [76] 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 the 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* [16] and *BitTyrant* [60] as examples of P2P systems. Although BitTorrent has a high setup cost (Table 3.1), it performs better than end-to-end copying tools (Rsync) for the target ratio of 0.5 because the majority of nodes have high throughput. However, BitTorrent ends up with a 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 slow nodes finish more slowly than in other systems. We see a similar trend with BitTyrant, but its performance is slightly worse than BitTorrent. This means that slow nodes are in a worse position if other peers behave strategically in the swarm.

Epidemic routing We also implemented an *epidemic routing* (or gossip) protocol [15] to measure its latency in the WAN. In the protocol, each node picks a random subset of remote nodes periodically, and exchanges file chunks. To evaluate it in the best possible scenario, 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 [27].

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 [25] used the gossip protocol for rapid file dissemination in the WAN and outperformed Bullet [43] and SplitStream [19] in terms of completion time in the evaluation. Our implementation showed better performance (141 seconds) than CREW's result (200 seconds) with a similar environment (60 nodes, 600 KB file, 200 Kbps bandwidth). As the topology is not specified in the CREW experiments, we randomly selected 60 PlanetLab nodes, and averaged over 10 repeated experiments. We used a user-level traffic shaper, Trickle [26], for setting bandwidth on the selected PlanetLab nodes.

"Epidemic Routing (best)" in Figure 3.10 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 outper-

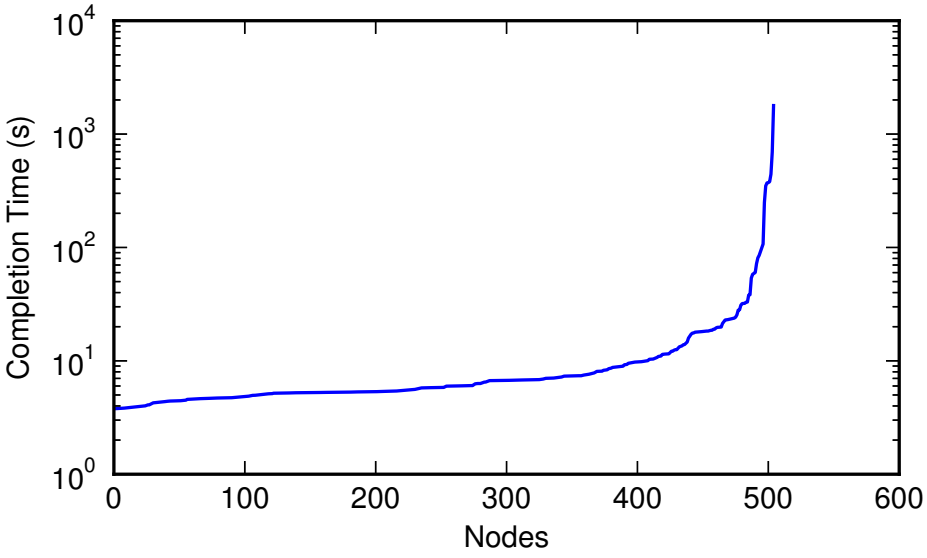


Figure 3.11: Distribution of CloudFront first fetch latency in all PlanetLab nodes – while most nodes have low latency, more than 10% of nodes are slow in fetching the uncached file via the overlay.

forms 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 ratios 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 the 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 the 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. CloudFront is faster than the other systems for the low target ratio tests due to its well-provisioned CDN nodes.

However, we observe that some nodes are still slow in fetching new file from the CDN for the high target ratio. Figure 3.11 shows the distribution of first fetch latency measured in all PlanetLab nodes. Most nodes have low startup latency, but 11% of nodes spend more than 20 seconds downloading the file via the CloudFront overlay. The nodes with poor CDN connectivity 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.

3.6.4 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 [71]. 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 the overlay mesh only (Overlay) 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

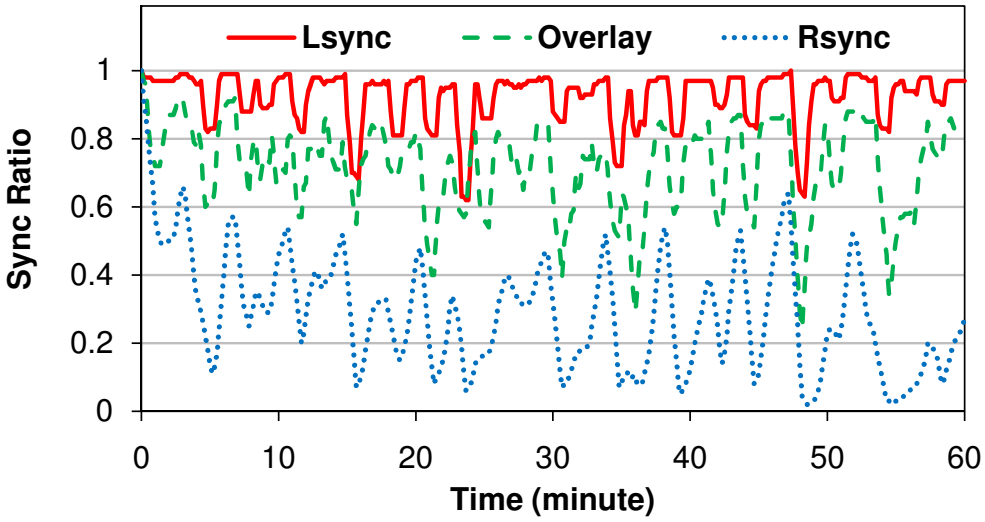
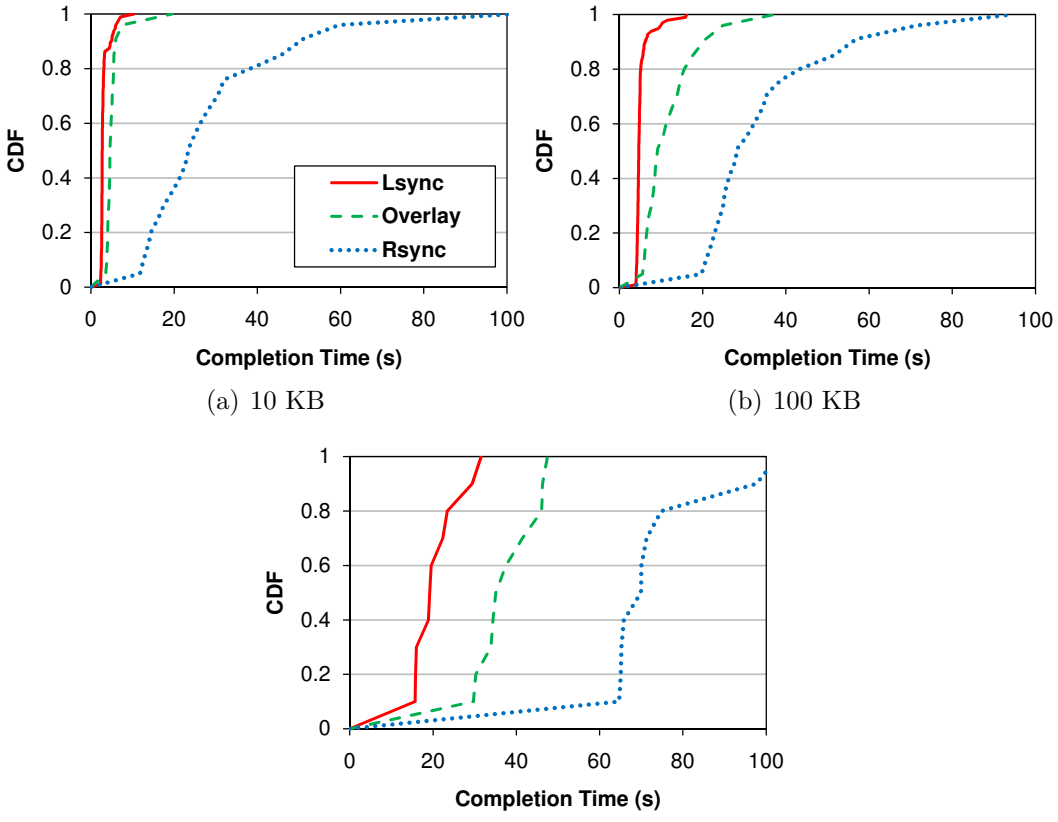


Figure 3.12: Frequently Added Files – Lsync makes most nodes fully synchronized during the entire period of the experiment.



(a) 10 KB

(b) 100 KB

(c) 1 MB

Figure 3.13: 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.

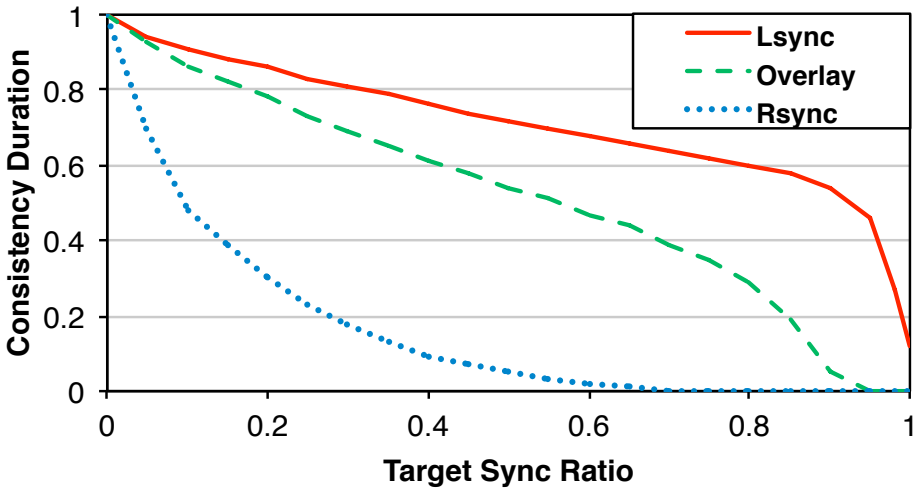


Figure 3.14: Consistency Duration – low-latency synchronization enables Lsync to achieve high consistency duration across all synchronization ratios.

over one minute. Figure 3.12 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 3.13 shows the distributions of the completion latency for 10 KB, 100 KB, and 1 MB files.

We also calculate the total duration of time when a given target synchronization ratio is satisfied, *consistency duration*, during the experiment. In Figure 3.14, we plot the duration for every target synchronization ratio. Lsync shows higher consistency duration than Rsync and CoBlitz because of its low-latency synchronization of individual files. If a system requires 95% of nodes to be full synchronized for its consistent views, Lsync can satisfy the consistency requirement for 50% of the time while the other two approaches cannot reach the ratio at all.

3.6.5 Lsync Contributing Factors

Lsync combines various techniques discussed in the chapter, 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

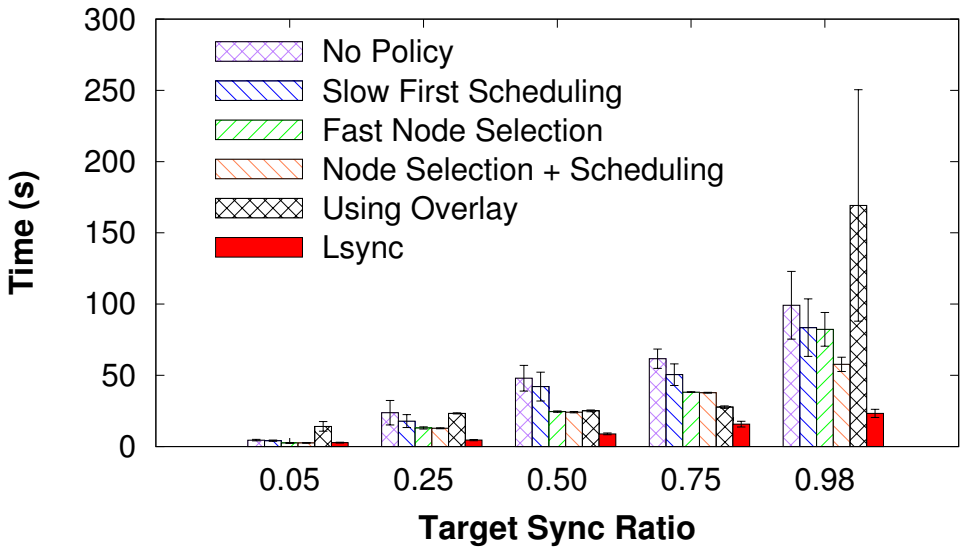


Figure 3.15: Lsync Contributing Factors – We see the individual contributions of node selection, node scheduling, and using the overlay. Each component contributes to the overall time reduction. Slow First scheduling improves the completion time for every target ratio, but that intelligent node selection is more critical at lower ratios. Using the overlay is slow for high target ratio because some nodes have very high startup latency.

transfer the CoBlitz web proxy executable file (600 KB) to all remote nodes as before. The results of these tests are shown in Figure 3.15. 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. Using the overlay is slow at low target ratios, and becomes comparable to the best end-to-end synchronization latency at

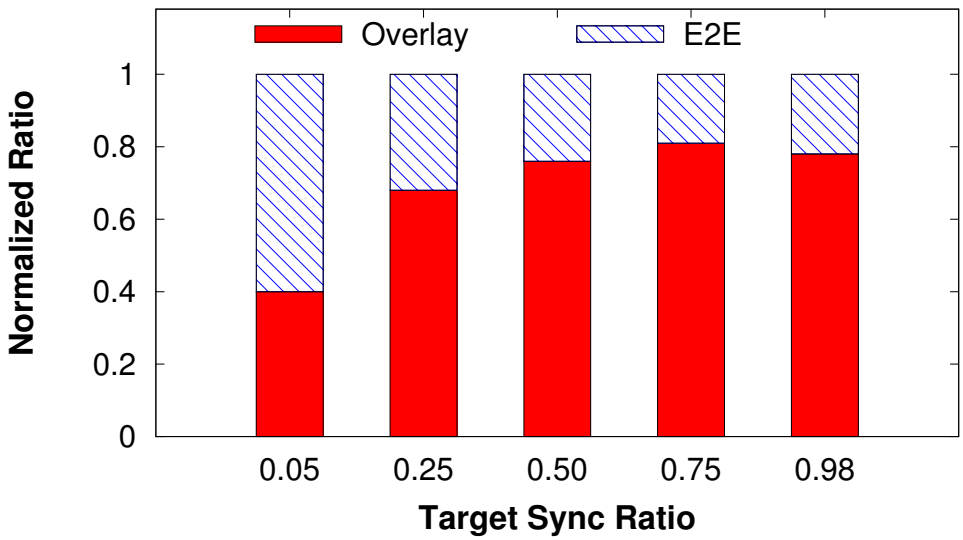


Figure 3.16: Division of Nodes in Lsync – We see that the fraction of nodes served by overlay mesh changes across target ratios, and that the fraction is not monotonically changing with target ratio.

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

3.6.6 Nodes Division and Adaptive Switching

To see how Lsync partitions the workload across the server and the overlay, 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 a file transfer at the same time. In Figure 3.16, 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.

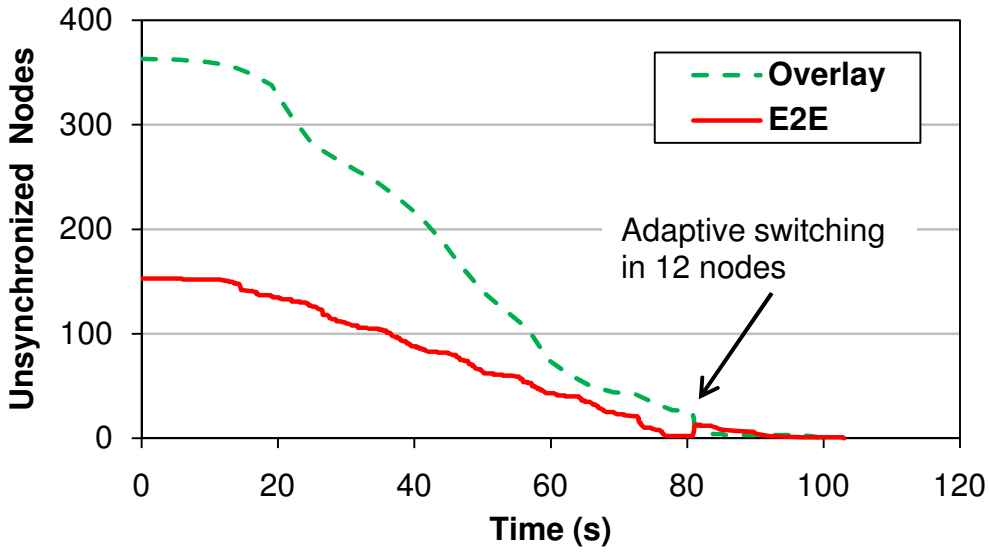


Figure 3.17: Adaptive Switching in Lsync – At 80 seconds, 12 nodes dynamically switch to end-to-end connections and finish downloading from the origin server.

Lsync makes adjustment at runtime, as can be seen in Figure 3.17, 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.

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 3.18 plots the variations of the

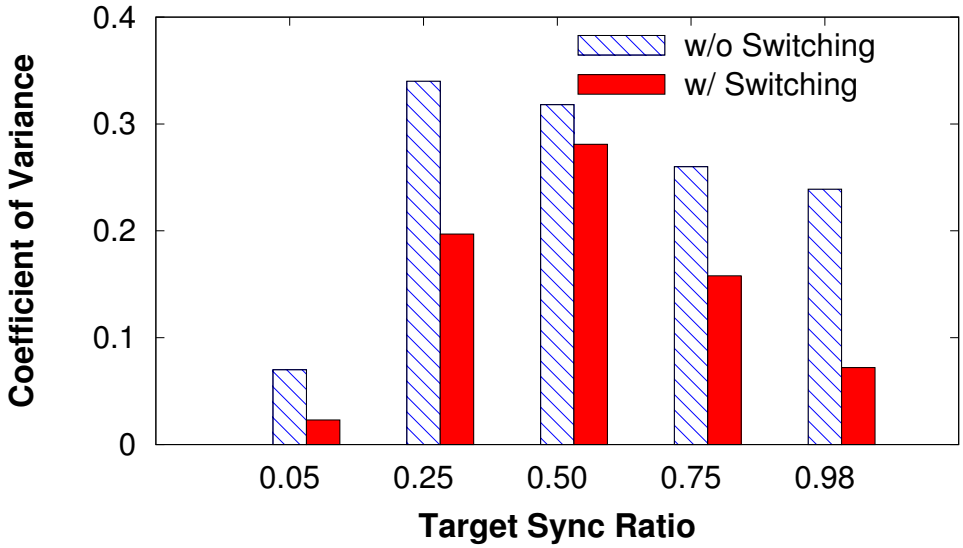


Figure 3.18: Stable File Transfers in Lsync – Adaptive switching in Lsync lowers variance of the latency.

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.

3.7 Related Work

CDNs and P2P systems [16, 28, 51, 52, 56, 60, 69] scalably distribute the content to a large number of clients through dynamic content replication, overlay construction, and request routing. 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 [59] 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 [4] 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 [7] uses similar techniques to improve job completion time in Map-Reduce clusters. The monitoring system detects outlier nodes and performs restarting or duplicating the straggling tasks based on resource constraints. For our target environments, it is not allowed to drop the slow nodes because some nodes may have persistent poor connections but still need to be synchronized to serve clients close to the nodes. Lsync's approach gives preference to the slow nodes suffering from either persistent or transient performance problems, keeping the overall latency low.

Lsync aggressively uses available resources as suggested in *buffet principle* [48]. When CDN/P2P systems are used, the server's spare bandwidth can be greedily exploited because it would otherwise remain unused. Lsync uses the spare bandwidth to assist nodes that are not covered well by the overlay mesh or that are experiencing unexpected performance problems. Our measurements demonstrate that using the spare bandwidth can significantly reduce latency as well as its variation. Another example is *dsync* [64] 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.

Lsync uses content fingerprinting [66] to find new chunks on file updates, which is widely adopted by network file systems and file transfer systems [8, 35, 50, 64]. Since these systems use end-to-end connections or overlay mesh to exchange chunks, Lsync can be applied in the systems for fast synchronization of particular chunks.

Execution management systems are developed for easing system development and maintenance in wide-area testbeds. AppManager [9] and PlMan [63] provide GUI-based interfaces that users can use for deploying files on remote nodes. Plush [3] takes high level specification of planned tasks, finds available resources, and monitors the

execution of the tasks on the remote nodes. Stork [18] provides an efficient software package management in PlanetLab by enabling slices to share the packages in a secure manner. These systems need rapid system deployment because most experiments on PlanetLab are short-lived and repeated with different configurations. We believe that Lsync can be easily integrated into the systems and improve the productivity of the users using the systems.

EdgeComputing [24] uses distributed Akamai CDN nodes as a platform for client-facing applications. The system allows service providers to deploy parts of their web application logic to the edge servers close to end users. The approach improves response time for interactive applications by processing user requests at the edge locations without traversing to the origin server over the public Internet. Likewise, NaDa [77] is a managed P2P system that uses ISP-controlled home gateways for storage and computing services. This decentralized approach reduces the infrastructure cost compared to large-scale centralized datacenters. Since software modules are distributed to the edge locations, these systems require a close control over their remote nodes. Lsync will improve the overall responsiveness of wide-area distributed systems by reducing the completion time of dissemination of new configuration or software modules.

File synchronizer tools provide an efficient way to replicate a shared folder across multiple machines. Rsync [76] uses a delta encoding to transfer only changed parts of the file over a low bandwidth link. Rsync server does not use any bandwidth allocation policy for sending the computed differences to multiple remote nodes. Unison [32, 61] provides safe two-way synchronization between two replicas based on a formal specification, but does not provide scalable transfer to multiple clients. Shotgun [42] is a set of extensions to Rsync to enable the server to efficiently synchronize a large number of remote clients. Shotgun runs Rsync in batch mode to locally compute the differences of old and new files, then the server uses Bullet [43], an overlay system to

disseminate the changes. Lsync adaptively uses the overlay because using the overlay does not always help improve completion time in the WAN.

Gossip-based broadcast [15, 27] 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. Similarly, overlay-based multicast systems [11, 19, 38] typically optimize for network topology and put slow nodes at the bottom of the multicast trees. This topology can improve aggregate bandwidth utilization, but lead to long synchronization latency in the system.

Chapter 4

Conclusion

Developing wide-area systems requires an infrastructure that allows system developers to deploy and test new services under realistic network conditions in the WAN. The next generation of network testbeds have been under active development recently, but very little is known about resource usage and user behavior in federated testbeds, leading to the development of conservative resource allocation policies.

In this dissertation, we conducted an extensive and in-depth analysis of six years of PlanetLab measurements to understand and characterize PlanetLab's resource usage. Based on the data-driven analysis, we discussed its design implications for future federated network testbeds. We found that the usage is much different from shared compute clusters, that conventional wisdom does not hold for PlanetLab, and that several properties of PlanetLab as a network testbed are largely responsible for this difference. We found that experiments typically utilize PlanetLab to expand their network reach, and that this metric of utility is a far better indicator of PlanetLab's effectiveness than compute-oriented metrics like CPU utilization.

We also showed that approaches focusing on compute-oriented metrics are likely to be inapplicable to PlanetLab-like workloads. In particular, we found that resource usage is very bursty, and that the vast majority of experiments consume very few

resources over much of their lifetimes. Based on the measurement of total resource usage, we explored the effectiveness of several resource allocation systems, and found that both pair-wise bartering and centralized banking systems can address only a small percentage of total resource usage. This result implies that resource management systems in federated network testbeds still need policies to fairly allocate available resources for the vast majority of the workload. We examined some policies for better resource discovery, node management, and pruning of runaway experiments.

From the measurement study of PlanetLab usage, we found that most PlanetLab experiments are short-lived but expand to a large fraction of available nodes in the testbed. This means that developing systems in the wide-area testbed is not interactive because long deployment delays will hamper every develop/deploy/test cycle during the short period of the usage.

In the second part of this dissertation, we presented Lsync, a low-latency one-to-many file transfer system for wide-area distributed systems. Low-latency data dissemination is an essential service for many wide-area platforms including federated testbeds, but latency has been largely ignored in existing data transfer systems. We measure the latency as the completion time of file transfer to all target remote nodes. We found that existing systems are suboptimal for this metric because they are not favorable to bottleneck nodes during a file transfer.

To solve this problem, we have identified the sources of the latency and described techniques to reduce their impact on the latency. We demonstrated that slow-first node scheduling and late-binding of the nodes can greatly reduce latency under various scenarios. The measurements on running CDN systems showed that using overlays can be slow and unpredictable when the file is not cached in the overlay. We developed techniques to adjust workload across a server and an overlay and dynamically switch policies to address unpredictable variation in overlay performance. Lsync integrates all of these techniques in a manner to minimize latency based on information available

at runtime. In addition to stand-alone applications, we expect that Lsync is useful to many wide-area distributed systems that need to coordinate the behavior of remote nodes with minimal latency.

Bibliography

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

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

[3] Jeannie Albrecht, Ryan Braud, Darren Dao, Nikolay Topilski, Christopher Tuttle, Alex C. Snoeren, and Amin Vahdat. Remote control: Distributed application configuration, management, and visualization with Plush. In *Proceedings of USENIX Large Installation System Administration Conference (LISA)*, 2007.

[4] Jeannie Albrecht, Christopher Tuttle, Alex C. Snoeren, and Amin Vahdat. Loose synchronization for large-scale networked systems. In *Proceedings of USENIX Annual Technical Conference (ATC)*, 2006.

[5] Jeannie R. Albrecht, David L. Oppenheimer, Amin Vahdat, and David A. Patterson. Design and implementation trade-offs for wide-area resource discovery. *ACM Transactions on Internet Technology (TOIT)*, 8(4), September 2008.

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

[7] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the outliers in Map-Reduce clusters using Mantri. In *Proceedings of USENIX Operating Systems Design and Implementation (OSDI)*, 2010.

[8] Siddhartha Annapureddy, Michael J. Freedman, and David Mazières. Shark: Scaling file servers via cooperative caching. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2005.

[9] AppManager. `http://appmanager.berkeley.intel-research.net/`.

[10] Alvin AuYoung, Brent N. Chun, Alex C. Snoeren, and Amin Vahdat. Resource allocation in federated distributed computing infrastructures. In *Workshop on Operating System and Architectural Support for the Ondemand IT Infrastructure (OASIS)*, 2004.

[11] Suman Banerjee, Bobby Bhattacharjee, and Christopher Kommareddy. Scalable application layer multicast. In *Proceedings of ACM SIGCOMM*, 2002.

[12] Nikhil Bansal and Mor Harchol-Balter. Analysis of SRPT scheduling: investigating unfairness. In *Proceedings of ACM SIGMETRICS*, 2001.

[13] Amnon Barak and Oren Laadan. The MOSIX multicomputer operating system for high performance cluster computing. *Future Generation Computer Systems*, 13(4-5), March 1998.

[14] Andy Bavier, Mic Bowman, Brent Chun, David Culler, Scott Karlin, Steve Muir, Larry Peterson, Timothy Roscoe, Tammo Spalink, and Mike Wawrzoniak. Operating system support for planetary-scale network services. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2004.

[15] Kenneth P. Birman, Mark Hayden, Oznur Ozkasap, Zhen Xiao, Mihai Budiu, and Yaron Minsky. Bimodal multicast. *ACM Transactions on Computer Systems*, pages 41–88, May 1999.

[16] BitTorrent. `http://bittorrent.com/`.

[17] J. Cappos, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Seattle: A platform for educational cloud computing. In *Proceedings of ACM Technical Symposium on Computer Science Education*, 2009.

[18] Justin Cappos, Scott Baker, Jeremy Plichta, Duy Nyugen, Jason Hardies, Matt Borgard, Jeffry Johnston, and John H. Hartman. Stork: Package management for distributed VM environments. In *Proceedings of USENIX Large Installation System Administration Conference (LISA)*, 2007.

[19] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. SplitStream: High-bandwidth multicast in cooperative environments. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[20] Brent N. Chun, Philip Buonadonna, Alvin AuYoung, Chaki Ng, David C. Parkes, Jeffrey Shneidman, Alex C. Snoeren, and Amin Vahdat. Mirage: A microeconomic resource allocation system for sensornet testbeds. In *Proceedings of IEEE Workshop on Embedded Networked Sensors*, 2005.

[21] Brent N. Chun and David E. Culler. User-centric performance analysis of market-based cluster batch schedulers. In *Proceedings of IEEE International Symposium on Cluster Computing and the Grid*, 2002.

[22] Cisco Wide Area Application Services (WAAS). `www.cisco.com/`.

[23] CoBlitz Inc. `http://www.verivue.com/`.

[24] Andy Davis, Jay Parikh, and William E. Weihl. EdgeComputing: Extending enterprise applications to the edge of the internet. In *ACM International Conference on World Wide Web (WWW)*, 2004.

[25] Mayur Deshpande, Bo Xing, Iosif Lazardis, Bijit Hore, Nalini Venkatasubramanian, and Sharad Mehrotra. CREW: A gossip-based flash-dissemination system. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2006.

[26] Marius A. Eriksen. Trickle: A userland bandwidth shaper for unix-like systems. In *Proceedings of USENIX Annual Technical Conference (ATC)*, 2005.

[27] P. Th. Eugster, R. Guerraoui, S. B. Handurukande, P. Kouznetsov, and A.-M. Kermarrec. Lightweight probabilistic broadcast. *ACM Transactions on Computer Systems*, pages 341–374, November 2003.

[28] Michael J. Freedman, Eric Freudenthal, and David Mazières. Democratizing content publication with Coral. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2004.

[29] Yun Fu, Jeffrey S. Chase, Brent N. Chun, Stephen Schwab, and Amin Vahdat. SHARP: an architecture for secure resource peering. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[30] GENI Aggregate Manager API. `http://groups.geni.net/geni/wiki/GeniApi/`.

[31] GENI: Global Environment for Network Innovations. `http://www.geni.net/`.

[32] Michael B. Greenwald, Sanjeev Khanna, Keshav Kunal, Benjamin C. Pierce, and Alan Schmitt. Agreeing to agree: Conflict resolution for optimistically replicated data. In *International Symposium on Distributed Computing (DISC)*, 2006.

[33] Mor Harchol-Balter, Bianca Schroeder, Nikhil Bansal, and Mukesh Agrawal. Size-based scheduling to improve web performance. *ACM Transactions on Computer Systems*, 2003.

[34] Qi He, Constantine Dovrolis, and Mostafa Ammar. On the predictability of large transfer TCP throughput. In *Proceedings of ACM SIGCOMM*, 2005.

[35] Sunghwan Ihm, Kyoungsoo Park, and Vivek S Pai. Wide-area network acceleration for the developing world. In *Proceedings of USENIX Annual Technical Conference (ATC)*, 2010.

[36] Tomas Isdal, Michael Piatek, Arvind Krishnamurthy, and Thomas Anderson. Privacy-preserving P2P data sharing with OneSwarm. In *Proceedings of ACM SIGCOMM*, 2010.

[37] Thomas Jackson, Ray Dawson, and Darren Wilson. Case study: evaluating the effect of email interruptions within the workplace. In *Proceedings of the 6th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, 2002.

[38] 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 USENIX Operating Systems Design and Implementation (OSDI)*, 2000.

[39] Ethan Katz-Bassett, Harsha V. Madhyastha, Vijay Kumar Adhikari, and Colin Scott. Reverse traceroute. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2010.

[40] Wonho Kim, Kyoungsoo Park, and Vivek S Pai. Server-assisted latency management for wide-area distributed systems. In *Proceedings of USENIX Annual Technical Conference (ATC)*, 2012.

[41] Wonho Kim, Ajay Roopakalu, Katherine Y Li, and Vivek S Pai. Understanding and characterizing PlanetLab resource usage for federated network testbeds. In *Proceedings of ACM Internet Measurement Conference (IMC)*, 2011.

[42] Dejan Kostic, Ryan Braud, Charles Killian, Erik Vandekieft, James W. Anderson, Alex C. Snoeren, and Amin Vahdat. Maintaining high bandwidth under dynamic network conditions. In *Proceedings of USENIX Annual Technical Conference (ATC)*, 2005.

[43] Dejan Kostic, Adolfo Rodriguez, Jeannie Albrecht, and Amin Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[44] Kevin Lai, Lars Rasmusson, Eytan Adar, Li Zhang, and Bernardo A. Huberman. Tycoon: An implementation of a distributed, market-based resource allocation system. *Multiagent and Grid Systems*, 1(3), August 2005.

[45] Sung-Ju Lee, Puneet Sharma, Sujata Banerjee, Sujoy Basu, and Rodrigo Fonseca. Measuring bandwidth between PlanetLab nodes. In *Proceedings of Passive and Active Measurement Conference (PAM)*, 2005.

[46] Harry C. Li, Allen Clement, Edmund L. Wong, Jeff Napper, Indrajit Roy, Lorenzo Alvisi, and Michael Dahlin. BAR gossip. In *Proceedings of USENIX Operating Systems Design and Implementation (OSDI)*, 2006.

[47] Michael J. Litzkow, Miron Livny, and Matt W. Mutka. Condor - a hunter of idle workstations. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 1988.

[48] Ratul Mahajan, Jitendra Padhye, Ramya Raghavendra, and Brian Zill. Eat all you can in an all-you-can-eat buffet: A case for aggressive resource usage. In *Proceedings of ACM Hot Topics in Networks (HotNets)*, 2008.

[49] Ratul Mahajan, Ming Zhang, Lindsey Poole, and Vivek Pai. Uncovering performance differences among backbone ISPs with Netdiff. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2008.

[50] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[51] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: A read/write peer-to-peer file system. In *Proceedings of USENIX Operating Systems Design and Implementation (OSDI)*, 2002.

[52] Erik Nygren, Ramesh K. Sitaraman, and Jennifer Sun. The Akamai network: A platform for high-performance internet applications. *ACM Operating Systems Review*, 2010.

[53] David Oppenheimer, Brent Chun, David Patterson, Alex C. Snoeren, and Amin Vahdat. Service placement in a shared wide-area platform. In *Proceedings of USENIX Annual Technical Conference (ATC)*, 2006.

[54] Parallel SSH. `http://www.theether.org/pssh/`.

[55] KyoungSoo Park and Vivek S. Pai. CoMon: A mostly-scalable monitoring system for planetlab. *ACM Operating Systems Review*, 2006.

[56] KyoungSoo Park and Vivek S. Pai. Scale and performance in the CoBlitz large-file distribution service. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2006.

[57] Larry Peterson, Andy Bavier, Marc E. Fiuczynski, and Steve Muir. Experiences building PlanetLab. In *Proceedings of USENIX Operating Systems Design and Implementation (OSDI)*, 2006.

[58] Ryan S. Peterson and Emin Gun Sirer. Antfarm: Efficient content distribution with managed swarms. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[59] Ryan S. Peterson, Bernard Wong, and Emin Gun Sirer. A content propagation metric for efficient content distribution. In *Proceedings of ACM SIGCOMM*, 2011.

[60] Michael Piatek, Tomas Isdal, Thomas Anderson, Arvind Krishnamurthy, and Arun Venkataramani. Do incentives build robustness in BitTorrent? In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2007.

[61] Benjamin C. Pierce and Jérôme Vouillon. What's in Unison? a formal specification and reference implementation of a file synchronizer. Technical Report MS-CIS-03-36, University of Pennsylvania, 2004.

[62] Platform Computing Load Sharing Facility (LSF). `http://www.platform.com/`.

[63] PlMan. `http://www.cs.washington.edu/research/networking/cplane/`.

[64] Himabindu Pucha, Michael Kaminsky, David G. Andersen, and Michael A. Kozuch. Adaptive file transfers for diverse environments. In *Proceedings of USENIX Annual Technical Conference (ATC)*, 2008.

[65] Lili Qiu, Yin Zhang, and Srinivasan Keshav. On individual and aggregate TCP performance. In *Proceedings of IEEE ICNP*, 1999.

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

[67] Rajesh Raman, Miron Livny, and Marvin H. Solomon. Matchmaking: Distributed resource management for high throughput computing. In *Proceedings of IEEE International Symposium on High Performance Distributed Computing (HPDC)*, 1998.

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

[69] Brad Karp John Kubiatowicz Sylvia Ratnasamy Scott Shenker Ion Stoica Sean Rhea, Brighten Godfrey and Harlan Yu. OpenDHT: a public DHT service and its uses. In *Proceedings of ACM SIGCOMM*, 2005.

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

[71] Alex Sherman, Philip A. Lisiecki, Andy Berkheimer, and Joel Wein. ACMS: The Akamai configuration management system. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2005.

[72] Sirius: A Calendar Service for PlanetLab. `https://snowball.cs.uga.edu/~dkl/pslogin.php/`.

[73] SliceStat: Slice monitoring sensor on PlanetLab. `http://codeen.cs.princeton.edu/slicestat/`.

[74] Neil Spring, Larry Peterson, Andy Bavier, and Vivek Pai. Using PlanetLab for network research: Myths, realities, and best practices. *ACM Operating Systems Review*, 40(1), January 2006.

[75] Andrew Tanenbaum, Sape Mullender, and Robert Van Renesse. Using sparse capabilities in a distributed operating system. In *Proceedings of IEEE International Conference on Distributed Computing Systems (ICDCS)*, 1986.

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

[77] Vytautas Valancius, Nikolaos Laoutaris, Laurent Massoulie, Christophe Diot, and Pablo Rodriguez. Greening the internet with nano data centers. In *Proceedings of ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2009.

[78] Limin Wang, KyoungSoo Park, Ruoming Pang, Vivek S. Pai, and Larry Peterson. Reliability and security in the CoDeeN content distribution network. In *Proceedings of USENIX Annual Technical Conference (ATC)*, 2004.

[79] Ming Zhang, Chi Zhang, Vivek S. Pai, Larry Peterson, and Randolph Y. Wang. PlanetSeer: Internet path failure monitoring and characterization in wide-area services. In *Proceedings of USENIX Operating Systems Design and Implementation (OSDI)*, 2004.