TOWARDS HIGHLY RELIABLE AND SCALABLE DISTRIBUTED SYSTEMS

KYOUNGSOO PARK

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

APRIL 2007

Abstract

Over the past decades, we have observed the development of a number of Internet-scale distributed systems such as the Domain Name System (DNS) and Content Distribution Networks (CDNs), which support the Internet access of millions of users. Though they have become indispensable infrastructure in the Internet, their operating dynamics have not been well studied so far. This dissertation focuses on the reliability and scalability of such large-scale distributed systems and explores the design principles for highly available and scalable systems.

One key principle for highly reliable services is distributed management of available resources based on autonomous monitoring. We implement CoDeeN, a latency-sensitive public CDN service with purely decentralized control, and demonstrate that its service reliability is greatly improved using careful resource monitoring, even in a highly unreliable environment. CoDeeN has been operating over three years, handling 5.8+ billion successful HTTP requests and serving over 14 million users, and has been one of the most stable long-running services on PlanetLab.

The second principle is that intelligent composition of temporarily unreliable resources can provide better reliability than any of the underlying resources. Using this principle, we build CoDNS, which has failure rates that range from one-tenth to one-hundredth of that of the existing DNS services. By aggregating the unreliable services, CoDNS improves availability by an extra '9', from 99% to over 99.9%, and in some cases achieves over 99.99% or less than 8 seconds of downtime per day. The utility of this service has also been proved in practice by providing more predictable and reliable name lookup service to the CoDeeN CDN service.

Finally, we find that the scalability of a large-scale distributed system can be immensely improved by independent and asynchronous node peering strategy and effec-

tive request distribution. CoBlitz, a scalable large-file transfer CDN atop CoDeeN and CoDNS, achieves downloading performance 27-48% higher than BitTorrent, while reducing the origin load by a factor of 7 more than the previously best known research system.

Acknowledgments

This dissertation could not have been possible without the help from a number of people around me. I cannot thank enough my advisor, Professor Vivek S. Pai, for his invaluable advice on countless events. I have always been inspired by his brilliant guidance, diligence and earnest attitude toward research. Throughout my Ph.D. program, he has provided me with more than enough support, encouragement and enthusiasm to finish this work. I would also like to thank Professor Larry Peterson who initiated the PlanetLab project and has made this world-wide distributed testbed come into life, where I could deploy and test all of our services. He has not only provided numerous valuable comments on our Co* systems but also helped remove many administrative obstacles in running our systems.

I am especially grateful to Professor Jennifer Rexford for many of her very insightful comments on my thesis, which helped greatly improve the quality of this thesis. I thank Professor Brian Kernighan and Andrea LaPaugh for serving non-readers on my thesis committee. They gave me many valuable suggestions and comments on my work.

I would like to thank the PlanetLab staffs, including Andy Bavier, Marc Fiuczynski, Mark Huang and Steve Muir for their support and patience during my experiments. I also thank the IT staffs of the Princeton computer science department, especially Scott Karlin and Chris Tengi, for their advice, support and tolerance with many of my resource-heavy experiments on the departmental computing infrastructure.

I cannot forget my off-campus industrial experience during two of my summer internships. While at IBM Research, Kang-Won Lee helped me familiarize with interesting real-world computing issues around the corporate environments. At Intel, I have learned a lot about the emerging virtualization techniques from many discussions with Mic Bowman. I thank them both for broadening my sight beyond the academia.

My life at Princeton has been very delightful due to my nice friends. I thank Jack Reilly and Dotty Westgate for tutoring English in my early years at Princeton, and my host family, Shahla and Eberhard Wunderlich who helped me settle down when I first arrived at Princeton. I would like to thank my graduate student friends at Princeton, Jaehyung Hwang, Sunghwan Ihm, Berk Kapicioglu, Changhoon Kim, Peter Kwan, Christine Lv, Ruoming Pang, Lindsey Pool, Yaoping Ruan, Limin Wang, Yi Wang and Doogab Yi, for their support and encouragement. Especially, Christine helped me figure out thesis printing problems in the last minute.

I could not have come this far if there were not encouragement by my parents. I thank my father, Duk-Hyun Park, and my mother, Kyoung-Soon Kim, for their everlasting love and care.

Finally, this dissertation is dedicated to my better half, Hyejin Huh, who has firmly believed in my work even when I am not sure and supported me with everything she could.

This work was supported in part by NSF grants ANI-0335214, CNS-0439842, CNS-0520053 and DARPA contract F30602-00-2-0561.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the explosive growth of the Internet, everyday Internet activities are increasingly dependent on the performance and reliability of large-scale distributed systems. Commercial content distribution networks (CDNs) transparently cache and deliver popular Web content to millions of end users on behalf of thousands of content providers, and the Domain Name System (DNS) hierarchically distributes over 70 million name resolutions [21], enabling ubiquitous access to the resources on the Internet via human-understandable names. These systems have become essential components of today's Internet infrastructure, playing an indispensable role in making the entire Internet scale.

However, understanding the dynamics of large-scale distributed systems has been a difficult task. Access to commercial systems is not easily granted to outsiders, and some systems are not owned by a single organization and could not have been monitored at more than a few places. Fortunately, the recent development of large-scale networking testbeds, such as PlanetLab [62], has brought opportunities for researchers to develop and deploy Internet-scale wide-area network projects subjected to real traffic conditions. Researchers can now observe the various aspects of their own deployed systems and care-

fully measure the behavior across an unprecedented number of geographically distributed vantage points.

Two important properties that we will discuss in this dissertation are reliability and scalability of large-scale distributed systems. Reliability refers to how consistently a given system performs in the presence of unpredictable failures. With the sheer complexity of typical large-scale distributed systems, often made worse by heterogeneous environments on which they operate, delivering high reliability is a difficult task. On the other hand, scalability refers to whether the system can provide similar reliability and performance as the system grows. Designing a small-scale system is relatively easy, but when the number of entities involved increases by many orders of magnitude, maintaining comparable performance is no longer straightforward.

We address reliability in two ways. One is to efficiently monitor the unreliable resources and adjust future usage. By avoiding the entities that are to exhibit a problem with a high probability, we show that we can dramatically improve the reliability of the entire system. The other approach is to intelligently multiplex unpredictable services to create a highly reliable service. We demonstrate that we can achieve an order of magnitude better availability by aggregating unreliable services with minimal resource consumption. In terms of scalability, we focus on the autonomous decision making of each participating node in the distributed system. Instead of resorting to high-cost consensus algorithms, independent decision making based on asynchronous monitoring provides surprisingly good scalability in practice.

This dissertation demonstrates these principles via two real deployed systems, a Web content distribution network and a Domain Name System resolver. We discuss how the monitoring infrastructure in a CDN can improve the service reliability in a non-dedicated environment where heavy resource contention may exist. We exercise the second prin-

ciple in designing a highly available name lookup service. We identify the origins of some DNS service instability, and demonstrate how to provide a fast and reliable name lookup service. Finally, we consider the scalable large-file distribution problem using these systems. We investigate the design issues in an efficient large-file transfer service on a regular HTTP CDN.

1.1 CDN in a Resource-contending Environment

The goal of a Web content distribution network is to improve the end user's Web experience by distributing, caching and serving content near the clients. Commercial CDNs operate by geographically distributing their servers and redirecting clients to one of the nearby replicas. They typically encode the URLs of embedded objects such as image or sound files so that resolving the domain name of the URL finds a server near the client. Their DNS servers pick a lightly loaded replica among the candidate servers, providing load balancing in the CDN system.

Commercial CDNs usually operate on an exclusive, dedicated environment where they can carefully provision and control the resources. However, when the environment becomes diverse, as in peer-to-peer systems, or when there is non-trivial resource contention in the environment, maintaining the proper level of service reliability becomes much harder. Providing latency-sensitive service in a non-dedicated environment poses new challenges which have not been previously researched.

Two traditional solutions in improving service reliability in the request-and-response framework are (a) to use parallel requests and respond with the fastest response and (b) to retry with a different replica when the timeout of the previous request expires (fail-over). Unfortunately, neither of these are appropriate for HTTP requests/responses, since

multiple requests may break the response idempotency because whether a request is a CGI or not cannot be determined *a priori*. Moreover, the fail-over mechanism would incur too much overhead in user-perceived delay for the CDN service to be useful.

Our approach is to have each node independently monitor the status of the necessary resources and services, and to proactively avoid near-term problems. The necessary resources include node-specific metrics such as available global file descriptors, physical memory and free CPU cycles as well as service-specific metrics such as application ping time and dummy request service time. DNS lookup performance and availability is another critical building block, as is network path stability. Assessing the reliability of each component and predicting and avoiding the problematic ones in the near future greatly improves the overall reliability of the CDN service.

The benefit of this approach is that the monitoring infrastructure can be reused in other environments like peer-to-peer systems or any shared environment with unreliable resources. In addition, by making the system autonomously react to failures without requiring a central authority, the management of the system becomes truly decentralized.

1.2 Highly Available DNS

The Domain Name System (DNS) provides the translation service between human readable domain names and machine friendly identifiers, called Internet Protocol (IP) addresses. With the enormous success of the HTTP protocol, the DNS service has become one of the most fundamental services in keeping the Internet connected.

A typical DNS name resolution works as follows. Assume you want to resolve "www.abc.com" from a Web browser. The Web browser calls a function in its resolver library, and the resolver issues a DNS query to its local nameserver. The local nameserver

goes to one of the root nameservers asking for the address of the ".com" nameservers, and looks up the ".abc.com" nameserver from the ".com" nameserver. Finally, it sends a DNS request for "www.abc.com" to the ".abc.com" nameserver and receives the IP address for "www.abc.com". The results of all the queries are cached at the local nameserver, and the final result is sent back to the resolver library and then to the Web browser.

As seen in the example above, DNS is a large-scale distributed system with its service reliability depending on various entities not owned by a single organization. Through caching and replication, it has maintained relatively good performance, but it has not been free of problems. Most problems in the early days of DNS deployment were related to buggy server implementations, but over time, more problems were found in the misconfiguration by the DNS operators. Though many implementation bugs have been fixed, misconfiguration problems continue to persist even these days.

Most DNS research in the past focused on analyzing the problems in the server-side infrastructure, where nameservers interact with each other. Our focus, however, is the client-side behavior between the name lookup client and its local nameserver, which has been largely ignored in the research community. With the local nameservers' cache hit rate approaching 90% these days, even a slight DNS lookup instability at the local nameservers may produce a large number of performance problems, affecting the reliability of other services depending on the DNS service.

We trace the origin of client-side problems and identify four common causes: (a) temporary nameserver overloading where local nameserver gets affected by bursty traffic, (b) packet loss in the path between the client and the local nameserver, (c) heavy regularly-scheduled tasks (e.g., cron jobs) on the nameserver machine, which temporarily forces the nameserver to compete for the resources, and (d) maintenance problems such as misconfiguration or having no backup machine for outages.

Our approach to these problems is to selectively utilize temporarily unreliable resources to produce a highly reliable service. We simply accept that the temporary failures can happen at any time, but instead of trying to fix the root cause completely, which may not be always possible, we use some other available peer DNS service at the time of failure. This is an intelligent fail-over mechanism, and it works well with DNS because DNS lookups are designed to produce no harmful side-effects, even if their responses are not idempotent.

Addressing the client-side problems this way provides direct benefit to the application without requiring access to the privileged nameserver itself. Also, our approach does not incur much overhead because the service just maintains contact with the peer group in the normal case and uses the local DNS service most of the time. Only when there is a problem with its local name lookup service, does the system forward the lookup query to one of its peers chosen by a deterministic algorithm to improve the query locality.

1.3 Scalable Large-file Distribution

The final piece of this dissertation is about how to design a scalable large-file distribution service using a regular HTTP CDN. Large files are increasingly common, ranging from high quality movies and podcast videos to on-line software packages such as Linux distribution CD/DVD ISOs. The typical size of such files is many hundred megabytes to a few gigabytes. Distributing popular large files bears a similarity to flash crowds in regular CDNs, but since large-file distribution has a different resource usage profile, it brings with it new challenges as well.

Two important factors in the scalability and performance of a large-file distribution service are the physical memory utilization and the request service time. Many large

files are bigger than the physical memory size of a CDN node, and applying the whole-file caching model, as when caching the common Web objects would produce frequent thrashing due to the enlarged working set. The situation is aggravated by the fact that the service time is many orders of magnitude longer than the time needed for traditional Web objects of size 10 KB or less. Consequently, serving popular large files based on the whole-file caching model would overload the CDN node itself, slowing down all downloads from that node.

One solution to the problems is careful provisioning of the resources, which is often used by commercial CDN vendors. By provisioning enough replicas and distributing the clients based on some sort of admission control, the CDN can avoid performance degradation due to frequent disk access or node overloading. One obvious drawback, however, is inefficient resource utilization. Also, the solution does not easily scale with lots of popular large files.

Our key observation is that we can reduce the large-file distribution problem to the small-file regular CDN problem. By splitting the large files into smaller chunks and distributing and caching them, we can take advantage of the aggregate memory without severely taxing any single node. When the system needs to serve the whole file, it can dynamically merge and deliver the chunks fetched from multiple CDN nodes. This design removes the thrashing problem by efficient use of memory, achieves load balancing among the CDN nodes, and reduces the average service time of an individual node.

One challenge in this scheme is how to deterministically map each chunk request to a CDN node regardless of network instability or CDN node churn. Improving the request locality by mapping the same chunk request to the same node is critical to good performance because multiple cache hits on the same node would make the chunk content available from its physical memory. A cache miss would force the node to fetch the chunk

from the origin server, so it would not only increase the response latency but too many cache misses would make the origin server easily overloaded.

Consistent hashing [44, 45] and Highest Random Weight (HRW) [82], which will be discussed in Chapter 2, would solve the node churns, but these algorithms assume that each node shares the same view on the other nodes, which means it does not take into consideration the possibility that partial network connectivity may create different views among the nodes. Structured peer-to-peer systems such as Chord [79], Pastry [72] and CAN [66] do not require the full knowledge of the nodes, but they cannot cope with partial network connectivity either. In practice, however, full network connectivity is often hard to achieve either because of temporary network partitions and failures or because of structural problems as in Internet2 and CANARIE which cannot route packets to the commerical Internet [32].

Instead of depending on any consensus algorithm, which would incur lots of overhead in practice, we let each node peer with others purely based on its own monitoring. This is done without consensus. Then, we route the chunk request using the HRW algorithm as follows. Each node calculates the HRW hash values for its own peers, picks the best replica based on the ordering, and forwards the request. The subsequent hops repeat the same process until there is no better node than the node itself. The final node is responsible for the chunk fetch in case of a cache miss and all the nodes in the route cache the chunk while delivering it to the client. This forwarding process converges quickly because of the global ordering property of the HRW algorithm. It is similar to routing in peer-to-peer systems, but the number of hops is usually much smaller because it does not need to route to the globally best node in case of a cache miss, whereas in peer-to-peer systems, it cannot avoid routing to the responsible node.

This unstructured HRW-based multi-hop routing, in addition to the independent peer-

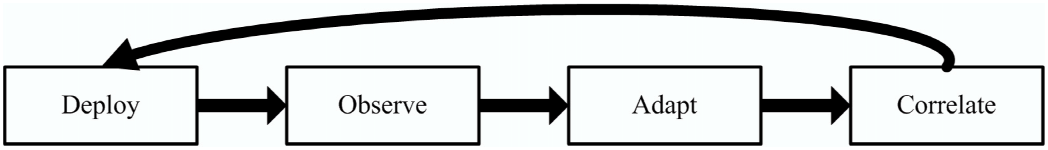


Figure 1.1: Incremental development model for system improvement

ing strategy, greatly improves the scalability of the system because new nodes can be added to the system without any provisioning, and the nodes will automatically distribute the load more evenly. Moreover, because each node does not depend on other nodes' potentially incorrect monitoring, it can avoid any temporary network failures or partitions and the request routing does not oscillate.

1.4 Contributions

The goal of this dissertation is to understand the essential factors in reliability and scalability of large-scale distributed systems, and provide guiding principles for designing highly reliable and scalable systems. In the process, we have designed and implemented real systems and observed the operational behavior in their interaction with the real-world clients. Three systems we have built for this purpose include CoDeeN, a content distribution network on a non-dedicated platform, CoDNS, a highly available DNS service, and CoBlitz, a scalable large-file transfer CDN on top of CoDeeN and CoDNS. By running them on PlanetLab and allowing public access to the services over three years, we have gained insight about how real-world systems behave under load, resource contention, and various failures, and we have developed design principles in producing highly reliable and scalable systems. These are summarized as follows.

1. Autonomous monitoring plays a critical part in improving the reliability of dis-

tributed systems. Any resources or services which may fail should be monitored and avoided in case of failure in order to reliably provide latency-sensitive service on a non-dedicated platform.

2. Intelligent composition of unreliable services produces a cost-effective and highly available service.

3. Independent decision-making is one of the key factors in achieving high scalability. Consensus and synchronization of large entities is not only hard to achieve but also harmful to system scalability.

Our general mode of system operation is shown in Figure 1.1, and consists of four steps: (1) deploy the system, (2) observe its behavior in actual operation, (3) determine how the underlying algorithms, when exposed to the real environment, cause the behaviors, and (4) adapt the algorithms to make better decisions using the real-world data. This model has been applied to all our systems and has greatly improved the development process.

1.5 Dissertation Overview

Chapter 2 describes the background techniques of the dissertation. We review the basic concept of consistent hashing and Highest Random Weight (HRW), and focus on the trade-offs in each scheme. HRW is the underlying request redirection algorithm in CoDeeN and CoBlitz, and distributes the load better than consistent hashing but at the cost of more CPU consumption. We also review the monitoring and peering strategies used by all three systems, which help the overall system to scale and to be robust to var-

ious failures. We focus on the general monitoring and peering issues in this chapter, and the specifics in each system will be discussed in their respective chapters.

Chapter 3 describes the monitoring system in the CoDeeN content distribution network. Over the three-year period it has been deployed on PlanetLab, CoDeeN has handled over 5.8 billion HTTP requests from over 14 million clients around the world. The operation of CoDeeN would not have been possible without autonomic monitoring, which provides service reliability in a resource-contending environment. The monitoring system measures the reliability of essential resources and services and avoids near term problems by detecting unreliable components. Chapter 3 discusses and compares with alternative strategies as well.

Chapter 4 describes CoDNS, a reliable DNS lookup service. One of the fundamental services in the Internet is name lookup, but its instability sometimes greatly undermines the reliability of other systems such as CDNs or email systems. We have found sources of instability in the client-side DNS infrastructure, and show a systematic way to improve its service availability. The CoDNS system improves availability by an order of magnitude over the existing lookup system while minimizing resource consumption.

Chapter 5 describes CoBlitz, a scalable large-file transfer CDN. The goal of CoBlitz is to scalably distribute large files to many simultaneous clients without custom software or a new protocol. We develop a novel technique to achieve high cache hit rates without overloading any individual CDN node. This technique also improves the scalability of the system while it dramatically reduces the load at the origin server. Because CoBlitz provides the service using the popular HTTP protocol, it does not require any modification on the server or the client. Finally, we summarize the lessons learned in Chapter 6, and conclude.

Chapter 2

Background

Request load balancing is one of the key functions in content distribution networks as well as in any request-and-response-based distributed caching systems. The request distribution algorithm should be designed to avoid any hot spots among the caching servers in the system and thus achieve load balancing without increasing the cache misses in the responsible nodes. Two representative static algorithms are consistent hashing [44, 45] and Highest Random Weight (HRW) [82]. More recently, dynamic request routing based on structured peer-to-peer systems [66, 72, 79] has been developed.

Another key component in large-scale distributed systems is how to establish peering relationship among the participating nodes. In many distributed systems where their operation depends on the healthy functioning of each peer, achieving appropriate peering relationships greatly affects the overall performance. In order to provide scalability and robustness in an autonomous fashion, the peering process must be highly dynamic and flexible for the system to quickly adjust to changes. Systematic monitoring helps each node independently determine available resources and services and provides insight into deciding which nodes to peer with. In this chapter, we discuss request distribution algo-

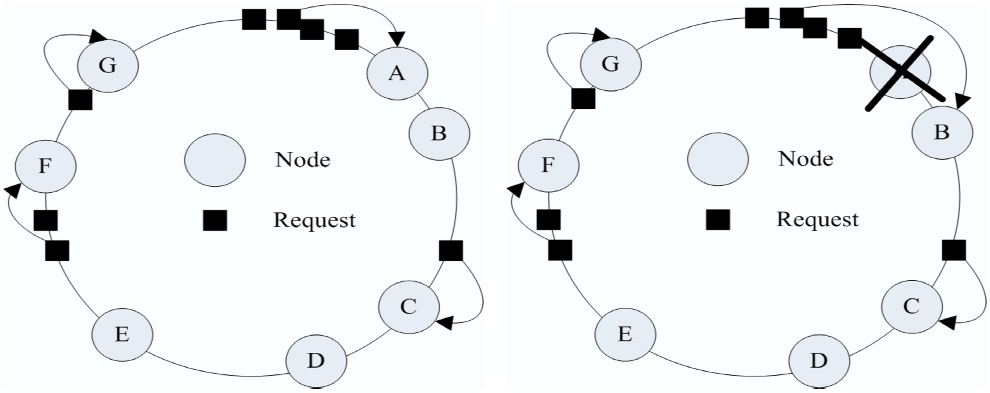


Figure 2.1: Consistent Hashing provides even request distribution under node churn, but may not be optimal when the requests are popular. When node A gets overloaded or leaves the network, then node B can be affected as well.

rithms and the general issues in node peering and resource monitoring in distributed systems. We focus on the common techniques that are used in all three systems – CoDeeN, CoDNS, and CoBlitz – here and the specifics of how each system implements them are discussed in more details in the following chapters. Finally, we breifly intoduce PlanetLab which is a distributed environment where our systems are running on.

2.1 Consistent Hashing and Highest Random Weight

The core problem in request distribution lies in how to consistently map the same requests to the same set of caching servers regardless of node churn. Traditional modulo hashing assigns each request to a server node mapped by the request's hash value modulo the number of total servers (ex: (3x + 5) mod 17). This method works well for evenly distributing the requests as long as the set of server nodes is static, but when an existing node leaves or a new node enters the system, all the requests have to be remapped to reflect the change. However, request remapping is undesirable because it reduces the cache utilization of previously-mapped requests and causes a sudden surge of cache misses.

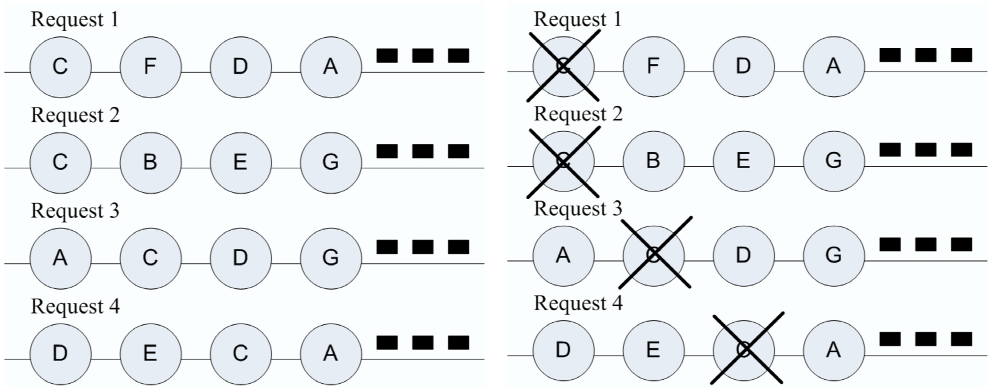


Figure 2.2: Highest Random Weight calculates the list of hashes and the highest ranked node is picked. In the example above, when node C gets unavailable (ex. due to load, etc.), request 1 and 2 are remapped to different nodes F and B, spreading out the load.

Consistent hashing [44, 45] eliminates the limitation of traditional hashing by avoiding the request remapping even when the node status changes. Its implementation uses a ring to map a request to one of participating servers. By randomly positioning the server nodes and requests in the ring, each request is mapped to the nearest node along the circle. Node churn redistributes only a fraction of requests in the ring, leaving the majority of other requests unaffected. Distributing the requests and nodes uniformly randomly makes the request mapping globally even among the servers in the long term.

Highest Random Weight (HRW) [82] is similar to consistent hashing, but instead of calculating one hash value for a request, thus mapping the request to one node, it generates a list of values to map a request to many nodes. The list of generated values are consistent with respect to the request, meaning that the same request would produce the same list of values, but the distribution of the values in each rank of the list is uniform and independent. Request redirection algorithms using HRW usually pick the highest ranked node in the list of hash values. When the top-ranked node is not live, the next live node in the list is picked as the responsible node for the request.

HRW has advantages over consistent hashing in handling popular requests but at the

cost of more CPU overhead. Assume there are two very popular requests, A and B, and they happen to be mapped to the same node under consistent hashing. If the node responsible for these requests gets overloaded or leaves the network, the load is shifted to a neighboring node, possibly overloading the neighboring node as well. However, in HRW, the next best node for A is unlikely to be the same node as the next best node for B, because the next-ranked node in the HRW list is supposed to be random with respect to the requests. This property helps distribute the load more evenly than consistent hashing in case of many popular requests. Figure 2.1 and Figure 2.2 show how both schemes work.

Structured peer-to-peer systems [66, 72, 79] can also be used to evenly distribute the requests with only partial knowledge of the participating nodes. A Distributed Hash Table (DHT) implemented on top of the peer-to-peer systems routes a request to the responsible node in O(log N) hops where N is the number of participating nodes. The basic idea is to build a local routing table per node such that the request gets forwarded to the next hop whose ID matches more bits of the request's ID than the current node. This property enables the request routing to make progress. Peer-to-peer systems are an attractive solution when the number of nodes is very large and there is no easy way of controlling the resources. One practical drawback is that it may incur significant latency overhead in multi-hop routing even if the number of hops is logarithmic in N. Also, ensuring the reliability of nodes in the route may not be trivial.

2.2 Scalable Peering and Monitoring

Large-scale distributed systems like content distribution networks require reliable cooperation among the participating server nodes. Maintaining appropriate peering relation-

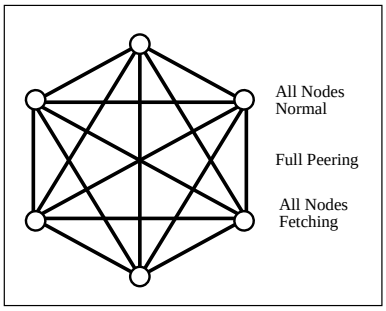


Figure 2.3: Standard peering for 6 unrestricted nodes

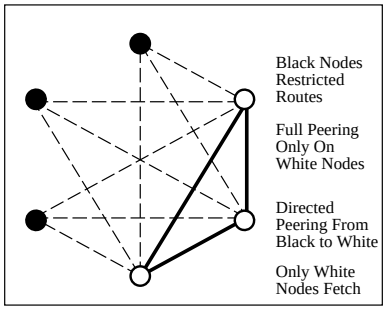


Figure 2.4: Peering with semi-routable Internet2

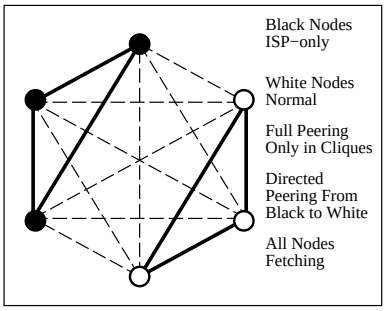


Figure 2.5: Peering with policy-restricted nodes

ships between nodes not only provides system reliability but also improves scalability and robustness. There are many ways of establishing the peering relationship depending on the particular environment, but we focus here on a decentralized environment where no central NOC (Network Operation Center) exists and thus no centralized monitoring information is available, which is similar to most peer-to-peer system environments.

The key challenge for reliable peering in this environment is how to scalably detect the system status changes, such as failures or node churn, and to propagate the information to the set of cooperating nodes in time for proper operation. Failures broadly include the node-specific ones caused by resource exhaustion or service instability and the infrastructure-related ones like network routing disruption or partitions. On a platform like PlanetLab where resources are shared by other experiments, resource exhaustion (disk space, global file table entries, physical memory) and contention (network bandwidth, CPU cycles) are not uncommon and these conditions sometimes lead to overall service degradation or failure. Therefore, to maintain reliable and smooth operation of a given service, each instance needs to monitor system health and exchange this data with its peers. Timely propagation of such information is critical in many cases because system operations often assume that a consistent view is shared by other nodes.

One approach is to use one of the view-synchronous group communication algorithms [5, 9, 75], or to apply a group membership protocol [26]. However, these protocols usually incur significant overhead in practice and thus cannot be used in latency-sensitive or near real-time environment. Instead, we pursue a radically different approach. We make every peering decision *unilaterally* and *independently* at each node, and avoid any synchronization overhead. Each node is responsible for monitoring a set of other nodes by periodically exchanging pairwise heartbeats and acknowledgments and selecting N best nodes from its own perspective. The heartbeat message includes such information as the node resource status, round-trip time (RTT) between the nodes and the network path health. The size of the peer set, N, is chosen based on the service type and the environment. For example, for the CoDNS service, which improves the DNS lookup reliability via a peer DNS service, a small number of different servers may be enough because most local DNS servers present over 95% availability, but for the CDN services like CoDeeN and CoBlitz, more peer nodes are beneficial for holding content in the aggregate cache and for distributing the request load, but the number should not be too large in order to monitor and update its peer node status within a short time period.

The motivation behind independent pairwise monitoring and peering is to favor simplicity and robustness. Even in some extreme circumstances where more than half of PlanetLab nodes froze due to a kernel bug in February 2005, CoDeeN continued to operate with its independent peering strategy. More sophisticated techniques, such as aggregating node health information using trees, can reduce the number of heartbeats, but can lead to *worse* information, since the tree may miss or use different links than those used for pairwise communication.

Another benefit of the independent peering strategy is that it improves the scalability with nodes in heterogeneous environments. These scenarios are shown in Figures 2.3,

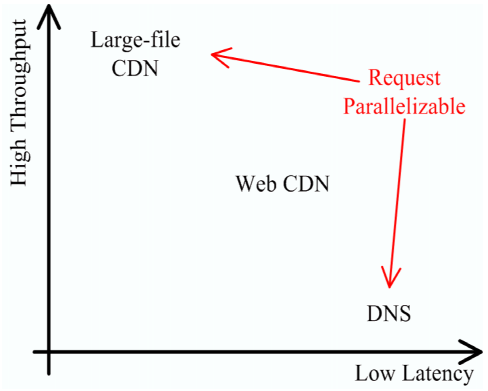


Figure 2.6: Design criteria trade-offs

2.4, and 2.5. For example, research networks like Internet2 or CANARIE (the Canadian high-speed network) do not peer with the commercial Internet, but are reachable from a number of research sites including universities and corporate labs. In our system, these nodes advertise that they do not want any regular nodes (including each other) using them as peers, since they cannot fetch content from the commercial Internet. However, these sites can unidirectionally peer with any nodes they can reach, which salvages their node utility. Also, in certain locations (like corporate environments or politically-sensitive regions), political/policy restriction makes the transfer of arbitrary content impossible, but the area may have a sizable number of nodes. These nodes will peer both with each other and with unrestricted nodes, giving them more peers than there would have been available otherwise. The same scenario applies to ISPs which host commercial CDN nodes in their network with the restriction that the CDN nodes only serve their own customers.

2.3 Trade-offs in System Design

In real-world applications, there are many other factors to be considered in addition to request locality, load balancing and peering issues. Such factors include latency, throughput and request parallelizability. For instance, Web CDNs have to maintain low response latency to support interactive service to the end users. However, in a service like large-file distribution, optimizing the throughput becomes much more important than providing low latency. For high reliability and low latency, requests can be sent in parallel and the fastest response can be taken, but this scheme depends on the properties of the particular service. For example, DNS queries can be replicated whereas an arbitrary Web request cannot.

Meeting all the criteria is not always possible because some of the factors are in conflict with each other. System designers should prioritize the factors depending on the type of the service and the environment. Figure 2.6 shows the design trade-offs regarding these criteria.

2.4 PlanetLab

PlanetLab is a research testbed consisting of 700+ geographically distributed nodes over 350+ sites in 30 countries in the world [62]. Its members include academic institutions like universities as well as various corporate and government research labs. The goal of PlanetLab is to help networking and distributed systems research projects to interact with the real environment and to innovate with the feedback from their real users. PlanetLab has been successful for validating numerous proof-of-concept research ideas in the real world, and for spawning many distributed services [31, 57, 60, 65, 86]. All of the services described in this dissertation have been evaulated and are currently running on PlanetLab.

Unlike in simulation or emulation environments, test results are not repeatable on PlanetLab because the nodes are not controllable with respect to outside changes such as network failures or partitions. Also, resource contention is not uncommon throughout the PlanetLab nodes. Resources on the nodes are shared by a number of other researchers running their own experiments at the same time. However, most essential resources (e.g., CPU cycles and physical memory) are usually available even at the busiest moments like the time just before paper deadlines [78]. Though each node does not behave like a typical desktop machine in peer-to-peer systems nor the PlanetLab network represent the Internet [78], PlanetLab is still an attractive infrastructure for testing distributed services to understand reliability and scalability of systems in operation. It has abundant heterogeniety in terms of node locations and bandwidth usage as desktop machines in peer-to-peer systems have, and the traffic from real users allows researchers to observe the dynamics of the system reliability according to different traffic patterns. Having the systems address these factors helps researchers better understand reliability and scalability which we focus on in this dissertation.

Chapter 3

CoDeeN

CoDeeN was the first deployed content distribution network on PlanetLab, with the goal of improving Web performance by using a novel request redirection algorithm [85]. It consists of 600+ [1] cooperating caching servers scattered over 300+ sites around the world, and distributes Web contents driven by the users who select a CoDeeN proxy in their browsers. Since CoDeeN has opened the service to the public in June 2003, it has handled over 5.8 billion HTTP requests, and has yet remained as one of the most stable and highly-used long-running services deployed on PlanetLab, continuously serving over 20 million daily requests from over 50,000 unique clients around the world.

The operational stability of CoDeeN in the early days of deployment, however, was not smooth enough until we observed and fixed the reliability problems in operation. CoDeeN is a complex system depending on reliable functioning of heterogeneous components, and when any component fails or becomes unavailable, the overall stability quickly degrades. One key insight in our early deployment endeavor is the observation that ensuring the system reliability in practice is more difficult to catch than traditional fail-stop

[1] As of October 2006

models assume. We have found that the status of these proxy nodes is much more dynamic and unpredictable than we had originally expected, which stimulated us to develop the essential principles for achieving the high reliability of a complex distributed system.

What CoDeeN aims to provide is a latency-sensitive CDN service with decentralized control, which is radically different from the centrally-controlled commercial CDNs or latency-insensitive peer-to-peer file sharing services. In addition, CoDeeN is competing for shared resources with other processes on each PlanetLab node, which makes it much harder to guarantee the stability of essential resources and to provide service reliability. In order to achieve high reliability in this environment, we argue that the system needs to tightly monitor the status of the resources that are needed in the near future, and avoid any unreliable components on a small time scale. This seemingly simple principle has laid the foundation of CoDeeN's operational reliability for over three years, and works surprisingly well in practice without much overhead. We also believe that this principle can be applied to other similar environments such as peer-to-peer systems where much heterogeneity is expected in terms of available resources and services. In this chapter, we discuss the essential resources and services, and how we can build a lightweight monitoring infrastructure.

3.1 CoDeeN Basics

Figure 3.1 shows the basic architecture of CoDeeN. CoDeeN consists of cooperating caching proxy servers, where each proxy operates as forward-mode proxy, reverse-mode proxy, and request redirector. When a proxy receives a request from a client and it is a cache hit, the proxy can respond with the cached object to the client (forward-mode proxy). In case of a cache miss, the proxy runs a deterministic request distribution algo-

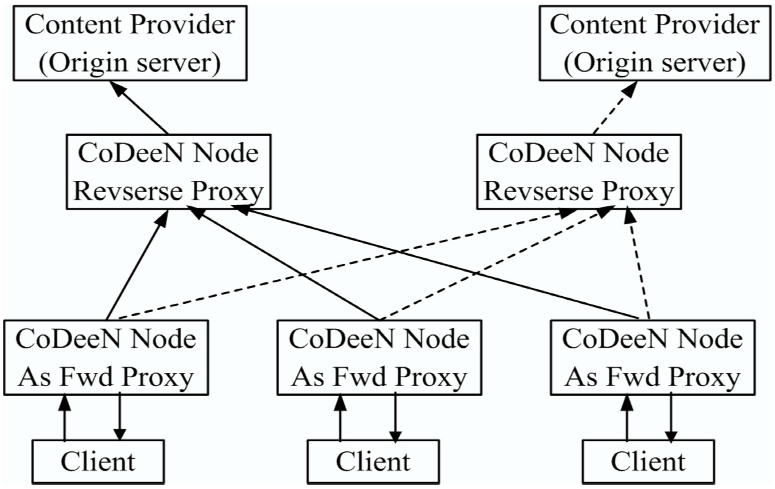


Figure 3.1: CoDeeN architecture – Clients configure their browsers to use a CoDeeN node, which acts as a forward-mode proxy. Cache misses are deterministically hashed and redirected to another CoDeeN proxy, which acts as a reverse-mode proxy, concentrating requests for a particular URL. In this way, fewer requests are forwarded to the origin site.

rithm in its peer set to pick a responsible proxy, and redirects the request to the chosen proxy (redirector). The chosen proxy now acts as a reverse-mode proxy to handle the request, and if it is a cache miss, the proxy fetches the content from the origin sever. While the fetched content is being delivered to the client, the content gets cached at both proxies in the chain and is used for future requests.

The request distribution algorithm used by CoDeeN is called Replicated Highest Ran-

```
∀ nodes, hash(i) = hashcalc(URL, node name(i))
hash = sort(hash)
hash = truncate(hash, NumCandidates)
∀ nodes, index(i) = node index number of hash(i)
minval = min(load(index(i)))
hash = select hash where load(index(i)) == minval
return index(random() modulo size(hash))
```

Figure 3.2: Replicated Highest Random Weight with Load Balancing

dom Weight (HRW) with Load Balancing [85], which provides intelligent load balancing on top of HRW. Figure 3.2 shows the pseudo code of the algorithm, which is also used for the CoBlitz scalable large-file distribution service, which will be discussed in Chapter 5. The difference in CoDeeN is that it uses a single hop routing in mapping the responsible reverse-mode proxy, whereas CoBlitz uses multi-hop routing to find the best reachable replica. In the presence of inconsistent views on the peer set of the participating nodes, the single hop routing may not always find the best replica in the peer set, though most requests do get consistently mapped as long as the difference in the peer set remains small. CoDeeN's design decision favors low latency, which is one of the key requirements for a Web CDN service, over optimal request locality, which is more important for a high-throughput service in reducing the origin server load.

3.2 Reliability from Peering and Monitoring

CoDeeN provides a latency-sensitive CDN service operating on a shared testbed, PlanetLab, where resource contention and exhaustion are not uncommon. [2] In order to maintain service reliability in cooperation with other nodes, CoDeeN adopts the pairwise monitoring and independent peering strategy discussed in Section 2.2. In a latency-sensitive environment such as CoDeeN, avoiding problematic nodes, even if they (eventually) produce a correct result, is preferable to incurring delays for keeping those nodes. Each CoDeeN instance first selects a healthy subset of the proxies, forms its peer set and then lets the redirection algorithm decide which one is the best for a given request. More specifically, each node collects up to 120 nodes within a 100ms round-trip time (RTT)

[2] PlanetLab may not be like the typical environments in which commercial distributed systems are deployed where less contention or exhaustion is expected. However, even on such platforms, failures do happen [34], and PlanetLab can be thought of as an environment with such failures more frequently happening.

boundary and periodically monitors these nodes by exchanging heartbeats. The 100ms cutoff is introduced to reduce the users' browsing latency, and the peer size, 120, is chosen because two heartbeats per second would sweep the peer nodes within one minute. Thus, the service failure duration is kept below one minute even when some peer nodes suddenly fail.

Two alternatives to active monitoring and avoidance, using retry/failover or multiple simultaneous requests, are not appropriate for this environment. Retrying failed requests requires that failure has already occurred, which implies latency before the retry. We have observed failures where the outbound connection from the reverse proxy makes no progress. In this situation, the forward proxy has no information on whether the request has been sent to the origin server. The problem in this scenario is the same reason why multiple simultaneous requests are not used – the idempotency of an HTTP request can not be determined *a priori*. Some requests, such as queries with a question mark in the URL, are generally assumed to be non-idempotent and uncacheable. However, the CGI mechanism also allows the query portion of the request to be concatenated to the URL without any special symbol (i.e., a question mark) that implies a CGI request. For example, the URL "/directory/program/query" may actually represent a CGI query like "/directory/program?query". As a result, sending multiple parallel requests and waiting for the fastest answer can cause errors.

The success of distributed monitoring and its effectiveness in avoiding problems depends on the relative difference in time between service failures and monitoring frequency. Our measurements indicate that most failures in CoDeeN are much longer than the monitoring frequency, and that short failures, while numerous, can be avoided by maintaining a recent history of peer nodes. The research challenge here is to devise effective distributed monitoring facilities that help to avoid service disruption and improve

Constant	Value
Peer node selection	Up to 120 nodes within 100ms RTT
Hearbeat rate	2 heartbeats per second
Uptime, OS CPU %, load averages	Checked every 30 seconds
Global fi le descriptor test	50 sockets creation test every 2 seconds
DNS lookup test	5 seconds as failure
OS CPU % for a bad node	95% or more are assumed to be bad
Packet loss threshold	5% loss rate as cutoff
Max. heartbeat ACK RTT	3 seconds

Table 3.1: Constants used in CoDeeN

system response latency. Our design uses heartbeat messages combined with other tests to estimate which other nodes are healthy and therefore worth using.

3.2.1 Local Monitoring

Local monitoring gathers information about the CoDeeN instance's state and its host environment, to assess resource contention as well as external service availability. Resource contention arises from competition from other processes on a node, as well as incomplete resource isolation. External services, such as DNS, can become unavailable for reasons not related to PlanetLab.

We believe that the monitoring mechanisms we employ on PlanetLab may be useful in other contexts, particularly for home users joining large peer-to-peer services. Most PlanetLab nodes tend to host a number of active experiments/projects at any given time. PlanetLab uses *vservers*, which provide a view-isolated environment with a private root filesystem and security context, but no other resource isolation. While this system falls short of true virtual machines, it is better than what can be expected on other non-dedicated systems, such as multi-tasking home systems. External factors may also affect service health. For example, a site's DNS server failure can disrupt the CoDeeN instance,

and most of these problems appear to be external to PlanetLab. The DNS problem, as well as our solution, CoDNS, will be discussed in depth in Chapter 4.

The local monitor examines the service's primary resources, such as free file descriptors, CPU cycles, and DNS resolver behavior. Non-critical information includes system load averages, node and proxy uptimes, traffic rates (classified by origin and request type), and free disk space. Some failure modes were determined by experience – when other experiments consumed all available sockets, not only could the proxy not tell that others were unable to contact it, but incoming requests appreared to be indefinitely queued inside the kernel.

Values available from the operating system/utilities include node uptime, system load averages (both via "/proc"), and system CPU usage (via "vmstat"). Uptime is read at startup and updated inside CoDeeN, while load averages are read every 30 seconds. Processor time spent inside the OS is queried every 30 seconds, and the 3-minute maximum is kept. Using the maximum over 3 minutes reduces fluctuations, and, at 120 nodes, exceeds the gap between successive heartbeats (described later) from any other node. We have the system avoid any node reporting more than 95% system CPU time, since we have found it correlates with kernel/scheduler problems. While some applications do spend much time in the OS, few spend more than 90%, and 95% generally seems failure-induced.

Other values, such as free descriptors and DNS resolver performance, are obtained via simple tests. We create and destroy 50 unconnected sockets every 2 seconds to test the availability of space in the global file table. Any failures over the past 32 attempts are reported, which causes peers to throttle traffic for roughly one minute to any node likely to fail. Similarly, a separate program periodically calls `gethostbyname()` to exercise the node's DNS resolver. To measure comparable values across nodes, and to reduce off-

site lookup traffic, only other (cacheable) PlanetLab node names are queried. Lookups requiring more than 5 seconds are deemed failed, since resolvers default to retrying at 5 seconds. We have observed DNS failures caused by misconfigured "/etc/resolv.conf" files, periodic heavyweight processes concurrently running on the same machine hosting the name servers, and heavy DNS traffic from other sources. More details will be discussed in the next chapter.

3.2.2 Peer Monitoring

To monitor the health and status of its peers, each CoDeeN instance employs two mechanisms – a lightweight UDP-based heartbeat and a "heavier" HTTP/TCP-level "fetch" helper. These mechanisms are described below.

UDP Heartbeat

As part of its tests to avoid unhealthy peers and network connectivity problems, CoDeeN uses UDP heartbeats as a simple gauge of liveness. UDP has low overhead and can be used when socket exhaustion prevents TCP-based communication. Since it is unreliable, only small amounts of non-critical information are sent using it, and failure to receive acknowledgments (ACKs) is used to infer packet loss.

Each proxy independently chooses the peer nodes within the boundary of 100ms cutoff with the maximum peer set size of 120. The 100ms cutoff was chosen to prevent noticeable lag in Web browsing for the clients. For monitoring the peer nodes, each proxy sends two heartbeat messages per second, which enables the sweep of all peers in a minute, and the peer proxies respond with information about their local state. The piggybacked load information includes the peer's average load, system time CPU, file descriptor availability, proxy and node uptimes, average hourly traffic, and DNS tim-

ing/failure statistics. With the 40-byte heartbeat and the current rate of two hearbeats per second, each node consumes only 80 bytes/sec (and another 80 bytes/sec for incoming heartbeats) on the heartbeat traffic. The aggregate heartbeat traffic is 256 Kbps (based on 400 live nodes), which is much smaller than an average of 80-100 Mbps of bandwidth spent on the actual content delivery by CoDeeN.

Heartbeat acknowledgments can get delayed or lost, giving some insight into the current network/node state. We consider acknowledgments received within 3 seconds to be acceptable, while any arriving beyond that are considered "late". The cutoff inter-node RTT within the peer set is 100ms, so not receiving an ACK in 3 seconds is abnormal. We maintain information about these late ACKs to distinguish between overloaded peers/links and failed peers/links, for which ACKs are never received. This information helps the system quickly reuse the nodes when temporary overloading is gone.

Several policies determine when missing ACKs are deemed problematic. Any node that does not respond to the most recent ACK is avoided, since it may have just recently died. Using a 5% loss rate as a limit, and understanding the short-term nature of network congestion, we avoid any node missing 2 or more ACKs in the past 32, since that implies a 6% loss rate. However, we consider viable any node that responds to the most recent 12 ACKs, since it has roughly a 54% chance of having 12 consecutive successes with a 5% packet loss rate [6], and the node is likely to be usable.

By coupling the history of ACKs with their piggybacked local status information, each instance in CoDeeN independently assesses the health of other nodes. This information is used by the redirector to determine which nodes are viable candidates for handling forwarded requests. Additionally, the UDP heartbeat facility has a mechanism by which a node can request a summary of the peer's health assessment. This mechanism is not used in normal operation, but is used for our central reporting system to observe

overall trends. For example, by querying all CoDeeN nodes, we can determine which nodes are being avoided and which are viable.

HTTP/TCP Heartbeat

While the UDP-based heartbeat is useful for excluding some nodes, it cannot definitively determine node health, since it cannot test some of the paths that may lead to service failures. For example, we have experienced site administrators port filtering TCP connections, which can lead to UDP packets being exchanged without obstruction, but all TCP connections resulting in failure after failed retransmission attempts.

To augment our simple heartbeat, we also employ a tool to fetch pages over HTTP/TCP using a proxy. This tool, conceptually similar to the "wget" program [35], is instrumented to specify what fails when it cannot retrieve a page within the allotted time. Possible causes include socket allocation failure, slow/failed DNS lookup, incomplete connection setup, and failure to retrieve data from the remote system. The DNS resolver timing measurements from this tool are fed into the instance's local monitoring facilities. Since the fetch tool tests the proxying capabilities of the peers, we must also have "known good" web servers to use as origin servers. For this reason, each CoDeeN instance also includes a dummy web server that generates a noncacheable response page for incoming requests.

The local node picks one of its presumed live peers to act as the origin server, and iterates through all of the possible peers as proxies using the fetch tool. After one iteration, it determines which nodes were unable to serve the requested page. Those nodes are tested to see if they can serve a page from their own dummy server. These tests indicate whether a peer has global connectivity or any TCP-level connectivity at all.

Over time, all CoDeeN nodes will act as an origin server and a test proxy for this testing. We keep a history of the failed fetches for each peer, and combine this with

```
    FdTstHst: 0x0
    ProxUptm: 36707
    NodeUptm: 111788
    LoadAvgs: 0.18 0.24 0.33
    ReqsHrly: 5234 3950 0 788 1004 275 2616
    DNSFails: 0.00
    DNSTimes: 2.48
    SysPtCPU: 2 2 1 3 2 4
Liveness: ..X.. ..X.. ..... .X.XX ..... ...X. .....
MissAcks: 10w00 00001 00000 0w066 00010 000v0 00020
LateAcks: 00000 00000 00000 00000 00000 00000 00000
NoFdAcks: 00000 00000 00000 00000 00000 00000 00000
VersProb: 00000 00000 00000 00000 00000 00000 00000
MaxLoads: 41022 11111 11141 20344 11514 14204 11111
SysMxCPU: 81011 11111 11151 10656 11615 15564 11111
WgetProx: 00w00 00100 00010 0w110 00000 000s0 00010
WgetTarg: 11w11 10301 01021 1w220 00111 101t0 11121
```

Figure 3.3: Sample monitoring log entry

the UDP-level heartbeats to determine if a node is viable for redirection. To allow for network delays and the possibility of the origin server becoming unavailable during one sweep, a node is considered bad if its failure count exceeds the other nodes by more than two.

3.2.3 Aggregate Information

Each CoDeeN proxy stores its local monitoring state as well as its peer summary to disk every 30 seconds, allowing offline behavior analysis as well as anomaly detection. The summary is also published and updated automatically on the CoDeeN central status page [22] every five minutes. These logs provide the raw data that we use in our analysis in Section 3.3. A sample log entry, truncated to fit in the column, is shown in Figure 3.3.

Most of the fields are the measurements that have been mentioned earlier, and the columns in the tabular output represent data about the other nodes in CoDeeN. Values in these lines are usually the counts in base-32 format, where 'w' represents 32. The exception is system CPU usage (SysMxCPU), which is the percentage value divided by 10 and rounded up. Based on collected information through UDP heartbeat and HTTP

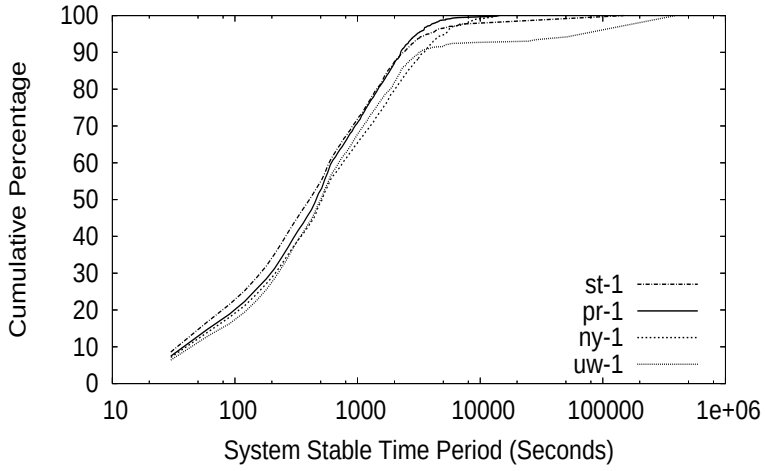


Figure 3.4: System Stability View from Individual Proxies

tests, each redirector decides the "Liveness" for each CoDeeN node, indicating whether the local node considers that peer node to be viable.

In this particular example, this node is avoiding six of its peers, mostly because they have missed several UDP ACKs. The eighth node, highlighted in boldface, is being avoided because it has a WgetTarg count of 3, indicating that it has failed the HTTP fetch test (with itself as the target) three times out of the past 32. More analysis on the statistics for node avoidance is presented in Section 3.3.

3.3 Evaluation

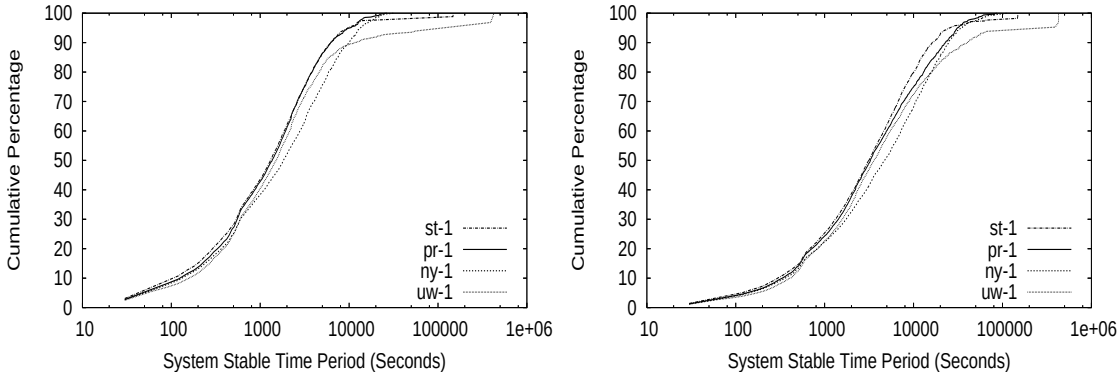
In this section, we analyze the data we collected during the first six months of CoDeeN's operation. These results not only show the status of CoDeeN over time, but also provide insights into the monitoring infrastructure.

3.3.1 Node Stability

The distributed node health monitoring system provides data about the dynamics of the system and insight into the suitability of our choices regarding monitoring. One would expect that if the system is extremely stable and has few status changes, an active monitoring facility may not be very critical and probably just increases overhead. Conversely, if most failures are short, then avoidance is pointless since the health data is too stale to be useful. Also, the rate of status changes can guide the decisions regarding peer group size upper bounds, since larger groups will require more frequent monitoring to maintain tolerable staleness.

Our measurements confirm our earlier hypothesis about the importance of taking a monitoring and avoidance approach. They show that our system exhibits fairly dynamic liveness behavior. Avoiding bad peers is essential and most failure time is in long failures so avoidance is an effective strategy. Figure 3.4 depicts the stability of the CoDeeN system with 40 proxies from four of our CoDeeN redirectors' local views. We consider the system to be stable if the status of all 40 nodes is unchanged between two monitoring intervals. We exclude the cases where the observer is partitioned and sees no other proxies alive. The x-axis is the stable period length in seconds, and the y-axis is the cumulative percentage of total time. As we can see, these 4 proxies have very similar views. For about 8% of the time, the liveness status of all proxies changes every 30 seconds (our measurement interval). In Table 3.2, we show the 50^{th} and the 90^{th} percentiles of the stable periods. For 50% of time, the liveness status of the system changes at least once every 6-7 minutes. For 90% of time, the longest stable period is about 20-30 minutes. It shows that in general, the system is quite dynamic – more than what one would expect from a few node joins/exits.

The trade-off between peer group size and stability is an open area for research, and



(a) Divided into 2 Groups (b) Divided into 4 Groups

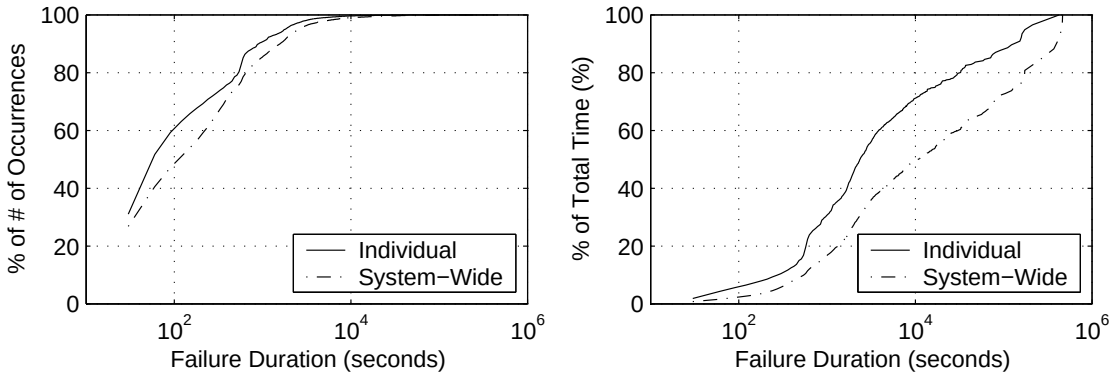
Figure 3.5: System stability for smaller groups

our data suggests, quite naturally, that stability increases as group size shrinks. The converse, that large groups become less stable, implies that large-scale peer-to-peer systems will need to sacrifice latency (via multiple hops) for stability. To measure the stability of smaller groups, we divide the 40 proxies into 2 groups of 20 and then 4 groups of 10 and measure group-wide stability. The results are shown in Figure 3.5 and also in Table 3.2. As we can see, with smaller groups, the stability improves with longer stable periods for both the 50^{th} and 90^{th} percentiles.

	40-node		2×20-node		4×10-node	
	50%	90%	50%	90%	50%	90%
pr-1	445	2224	1345	6069	3267	22752
ny-1	512	3451	1837	10020	4804	25099
uw-1	431	2085	1279	5324	3071	19579
st-1	381	2052	1256	5436	3008	14334

Table 3.2: System Stable Time Period (Seconds)

The effectiveness of monitoring-based avoidance depends on the node failure duration. To investigate this issue, we calculate node avoidance duration as seen by each node and as seen by the sum of all nodes. The distribution of these values is shown in Figure 3.6, where “Individual” represents the distribution as seen by each node, and



(a) CDF by # of Occurrences (b) CDF by Total Time

Figure 3.6: Node Failure Duration Distribution. Failures spanning across a system-wide downtime are excluded from this measurement, so that it only includes *individual* node failures. Also, due to the interval of node monitoring, it may take up to 40 seconds for a node to be probed by another nodes, thus failures that last a shorter time might be neglected.

"System-Wide" counts a node as failed if all nodes see it as failed. By examining the durations of individual failure intervals, shown in Figure 3.6a, we see that most failures are short, and last less than 100 seconds. Only about 10% of all failures last for 1000 seconds or more. Figure 3.6b shows the failures in terms of their contribution to the total amount of time spent in failures. Here, we see that these small failures are relatively insignificant – failures less than 100 seconds represent 2% of the total time, and even those less than 1000 seconds are only 30% of the total. These measurements suggest that node monitoring can successfully avoid the most problematic nodes.

3.3.2 Reasons to Avoid a Node

Similar to other research on peer-to-peer systems [8, 37, 67, 73], we initially assumed that churn, the act of nodes joining and leaving the system, would be the underlying

Site	Fetch	Miss ACKs	Node Down	Late ACKs	DNS
pr-1	6.2	18.3	29.6	13.6	32.1
ny-1	4.7	16.1	31.7	14.0	33.9
uw-1	10.4	16.8	30.0	12.8	29.7
st-1	5.0	14.7	27.2	15.4	34.3

Table 3.3: Average Percentage of Reasons to Avoid A Node

cause of availability-related failures. [3] However, as can be seen from the stability results, failure occurs at a much greater rate than churn. To investigate the root causes, we gather the logs from 4 of redirectors and investigate what causes nodes to switch from viable to avoided. Therefore, our counts also take time into account, and a long node failure receives more weight. We present each reason category with a non-negligible percentage in Table 3.3. We find that the underlying cause is roughly common across nodes – mainly dominated by DNS-related avoidance and many nodes down for long periods, followed by missed ACKs. Even simple overload, in the form of late ACKs, is a significant driver of avoidance. Finally, the HTTP fetch helper process can detect TCP-level or application-level connectivity problems.

In terms of design, these measurements show that a UDP-only heartbeat mechanism will significantly underperform our more sophisticated detection. Not only are the multiple schemes useful, but they are complementary. Variation occurs not only across nodes, but also within a node over a span of multiple days. The data for the ny-1 node, calculated on a daily basis, is shown in Figure 3.7.

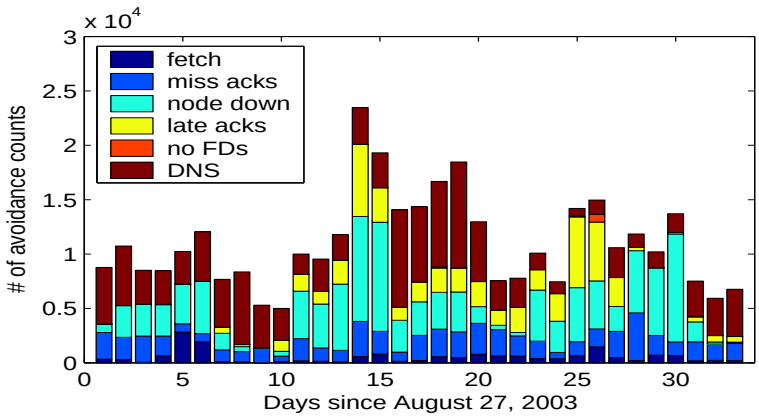


Figure 3.7: Daily counts of avoidance on ny-1 proxy

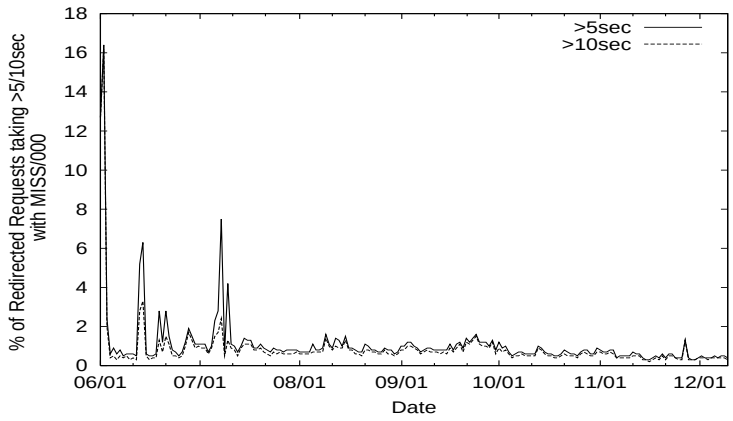


Figure 3.8: Percentage of Non-serviced Redirected Requests

3.3.3 Response Performance

The service response time behavior is largely a function of how well the system performs in avoiding bad nodes. The results of our efforts to detect/avoid bad nodes can be seen in Figure 3.8, which shows requests that did not receive any service within specific time intervals. When this occurs, the client is likely to stop the connection or visit another

[3]Even though node churn at PlanetLab is less frequent than in typical peer-to-peer systems, given the excessive exchange of messages and thus instability caused by churn as reported in the peer-to-peer systems literature above, we initially thought that churn is the major cause for most failures.

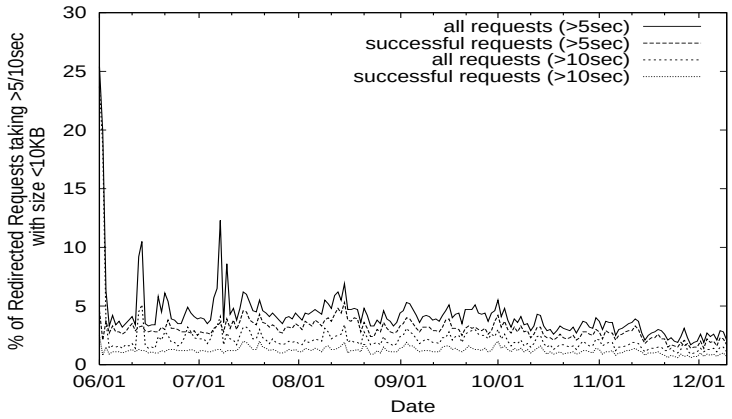


Figure 3.9: Percentage of Redirected Requests ($<$ 10KB)

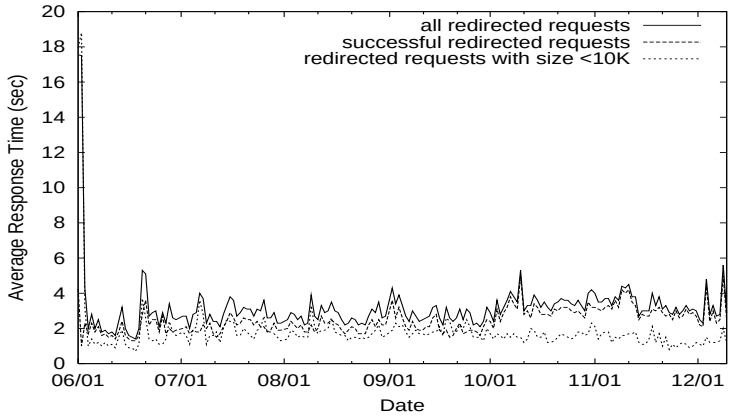


Figure 3.10: Average Response Time of Redirected Requests in 2003

page, yielding an easily-identifiable access log entry (MISS/000). These failures can be the result of the origin server being slow or a failure within CoDeeN. The trend shows that both the magnitude and frequency of the failure spikes are decreasing over time. DNS failure detection was added in late August 2003, and appears to have yielded positive results.

Since we cannot "normalize" the graphs for the different traffic over CoDeeN, other measurements are noisier, but also instructive. Figure 3.9 shows the fraction of small/failed

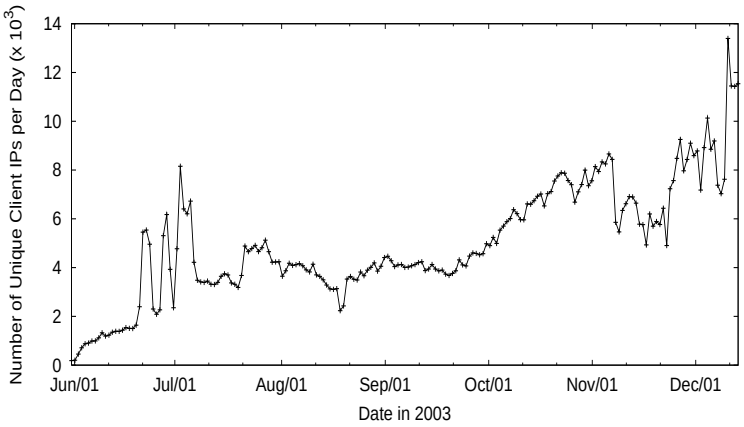


Figure 3.11: Daily Client Population (Unique IP) on CoDeeN in 2003

responses that take more than a specific amount of time. Here, we only show redirected requests, which means they are not serviced from the forward proxy cache. By focusing on small responses, we can remove the effects of slow clients downloading large files. This is to separate the case where the service itself is slow and the case where the slow response time is caused by some external factor such as slow clients. We see a similar trend where the failure rate decreases over time. The actual overall response times for successful requests, shown in Figure 3.10, has a less interesting profile. After a problematic beginning, responses have been relatively smooth. As seen from Figure 3.14, since the beginning of October 2003, we have received a rapidly increasing number of requests on CoDeeN, and consequently, the average response time for all requests slightly increases over time. However, the average response time for small files is steady and keeps decreasing. This result is not surprising, since we have focused on reducing failures rather than reducing success latency.

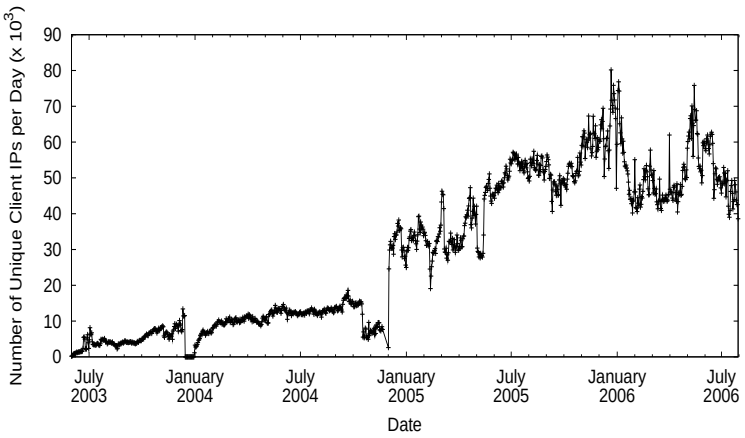


Figure 3.12: Daily Client Population (Unique IP) on CoDeeN

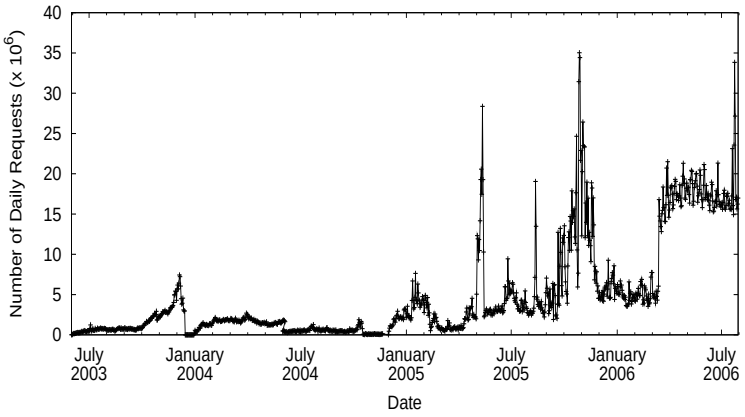


Figure 3.13: Daily Requests Serviced on CoDeeN

3.3.4 Traffic

As the system became stable, the number of daily clients used to access CoDeeN increased with the values shown in Figure 3.11. During the first six months, the number of unique client IP addresses passed 500,000, and the total number of unique IPs for the whole period is over 14 million. Now, our daily traffic regularly exceeds 50,000 unique IPs as shown in Figure 3.12. The two valleys, at December 2003 and at October 2004, are due to a PlanetLab kernel upgrade, which caused the majority of the nodes unavailable.

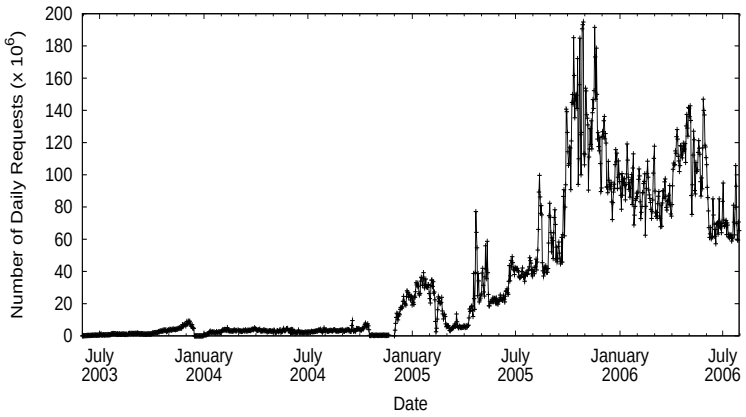


Figure 3.14: Daily Requests Received on CoDeeN

During the whole period of operation so far, CoDeeN has handled more than 5.4 billion legitimate requests (excluding abusive requests that are rejected), and the daily traffic served by CoDeeN now hovers above 20 million requests, with peaks at 35 million, as seen in Figure 3.13. The number of requests (as well as the clients) started to increase significantly when CoDeeN expanded the set of nodes to all PlanetLab nodes from North American educational sites in January 2005. The major bump in the number of requests in January 2006 is due to deployment of the robot detection mechanism [61], which increased the bandwidth for humans while removing the abusive robot traffic. The set of security mechanisms deployed on CoDeeN, which is beyond the scope of this dissertation, can be found at [56, 61].

A couple of spikes in Figure 3.11 and in Figure 3.13 are related to some external events: we have found that some anonymizer software site lists CoDeeN nodes as open proxies, and the updated list is pushed out to a large number of their customers. These events temporarily increase the number of requests and clients of CoDeeN, but as soon as they find out that CoDeeN is not useful for anonymizing their requests, they remove the CoDeeN proxy IPs from the setting, which would make those spikes.

3.4 Related Work

Similar to CoDeeN, peer-to-peer systems [66, 72, 79] also run in a distributed, unreliable environment. Nodes join or depart the system from time to time, and node failures can also happen frequently [67]. What makes the situation even worse is that the network path is sometimes non-transitive, [4] making global view sharing much difficult [32]. In order to maintain reliability in this environement, besides maintaining a membership directory, these systems typically apply a retry and fail-over scheme to deal with failing nodes while routing to the destinations. Although these retries are generally expected by peer-to-peer system users, the extra delays in retrying different next hops can cause latency problems. For latency-sensitive applications implemented on peer-to-peer routing, multiple hops or retries in each operation become even more problematic [25]. To address multiple-hop latency, recent research has started pushing more membership information into each node in a peer-to-peer system to achieve one-hop lookups [38, 68]. In this regard, similar arguments can be made that each node could monitor the status of other nodes.

Some researchers have used Byzantine fault tolerant approaches to provide higher reliability and robustness than fail-stop assumptions provide [1, 16]. While such schemes, including state machine replication in general, may seem appealing for handling failing nodes in CoDeeN, the fact that origin servers are not under our control limits their utility. Since we cannot tell that an access to an origin server is idempotent, we cannot issue multiple simultaneous requests for one object due to the possibility of side-effects. Such an approach could be used among CoDeeN's reverse proxies if the object is known to be cached.

In the cluster environment, systems with a front end [36] can deploy service-specific

[4] The fact that node A can reach node B, and node B can reach node C does not automatically mean that node A can reach node C.

load monitoring routines in the front end to monitor the status of server farms and decide to avoid failing nodes. These generally operate in a tightly-coupled environment with centralized control. There are also general cluster monitoring facilities that can watch the status of different nodes, such as the Ganglia tools [33] or CoMon [59], which have been used on PlanetLab. We can potentially take advantage of them to collect system level information. However, we are also interested in application-level metrics such as HTTP/TCP connectivity, and some of resources such as DNS behaviors that are not monitored by Ganglia.

Cooperative proxy cache schemes have been previously studied in the literature [17, 63, 81, 88], and CoDeeN shares many similar goals. However, to the best of our knowledge, the only two deployed systems have used the Harvest-like approach with proxy cache hierarchies. The main differences between CoDeeN and these systems are in the scale, the nature of who can access, and the type of service provided. Neither system uses open proxies. The NLANR Global Caching Hierarchy [55] operates ten proxy caches that only accept requests from other proxies and one end-user proxy cache that allows password-based access after registration. The JANET Web Cache Service [40] consists of 17 proxies in England, all of which are accessible only to other proxies. Joining the system requires providing your own proxy, registering, and using an access control list to specify which sites should not be forwarded to other caches. Entries on this list include electronic journals.

Coral [31] is an Akamai-like CDN based on a peer-to-peer system, and has provided public service like CoDeeN. Bad nodes are avoided by DNS-based redirection, sometimes using an explicit UDP RPC for status checking.

Chapter 4

CoDNS

The Domain Name System (DNS) [54] has become a ubiquitous part of everyday computing due to its effectiveness, human-friendliness, and scalability. It provides a distributed lookup service primarily used to convert from human-readable machine names to Internet Protocol (IP) addresses. Its service has become an integral part of Internet computing via the World Wide Web's near-complete dependence on it. Thanks in part to its redundant design, aggressive caching, and flexibility, it has become a ubiquitous part of everyday computing that most people, including researchers, take for granted,

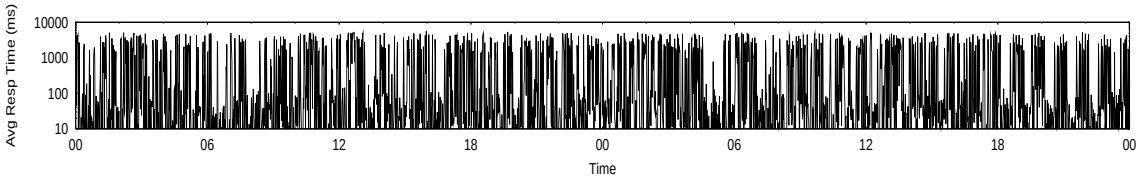
Most DNS research focuses on "server-side" problems, which arise on the systems that translate names belonging to the group that runs the systems. Such research includes understanding name hierarchy misconfiguration [28, 42] and devising more scalable distribution infrastructure [25, 43, 65]. However, due to increasing memory sizes and DNS's high cachability, "client-side" DNS hit rates are approaching 90% [42, 87], so fewer requests are dependent on server-side performance. The client-side components are responsible for contacting the appropriate servers, if necessary, to resolve any name presented by the user. This infrastructure, which has received less attention, is our focus

– understanding client-side behavior in order to improve overall DNS performance and reliability.

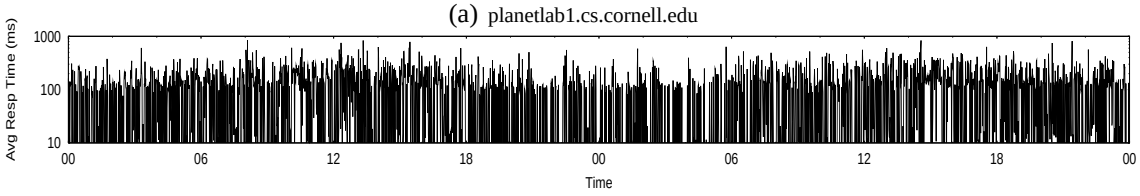
Using PlanetLab, we locally monitor the client-side DNS infrastructure of 150 sites around the world, enabling a large-scale examination of client-side DNS performance. We find that client-side failures are widespread and frequent, and that their effects degrade DNS performance and reliability. The most common problems we observe are intermittent failures to receive any response from the local nameservers, but these are generally hidden by the internal redundancy in DNS deployments. However, the cost of such redundancy is additional delay, and we find that the delays induced through such failures often dominate the time spent waiting on DNS lookups.

To address these client-side problems, we have developed CoDNS, a lightweight, cooperative DNS lookup service that can be independently and incrementally deployed to augment existing nameservers. CoDNS uses an insurance-like model of operation – groups of mutually trusting nodes agree to resolve each other's queries when their local infrastructure is failing. We find that the group size does not need to be large to provide substantial benefits – groups of size 2 provide roughly half the maximum possible benefit, and groups of size 10 achieve almost all of the possible benefit. Using locality-enhancement techniques and proximity-favoring design, CoDNS achieves low-latency, low-overhead name resolution, even in the presence of local DNS delays/failures.

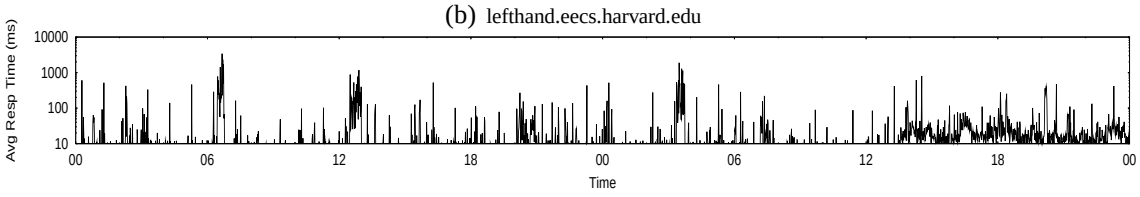
CoDNS has been serving live traffic on PlanetLab since October 2003, providing many benefits over standard DNS. CoDNS reduces average lookup latency by 27-82%, greatly reduces slow lookups, and improves DNS availability by an extra '9', from 99% to over 99.9%. Its service is more reliable and consistent than any individual node's. Additionally, CoDNS has salvaged "unusable" nodes, which had such poor local DNS infrastructure that they were unfit for normal use.



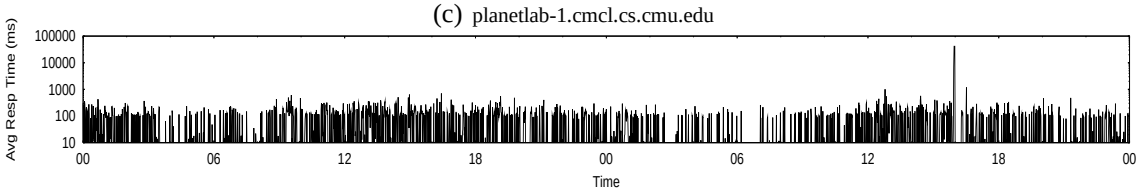
(a) planetlab1.cs.cornell.edu



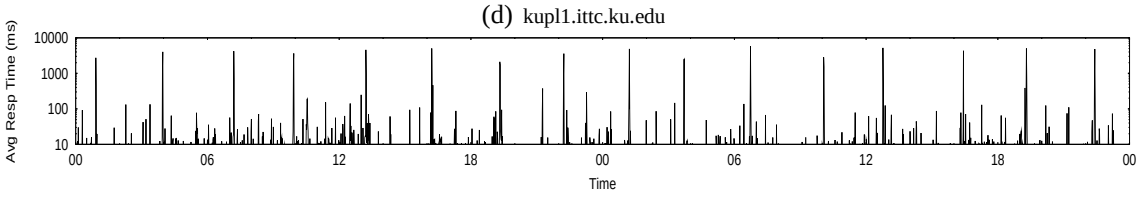
(b) lefthand.eecs.harvard.edu



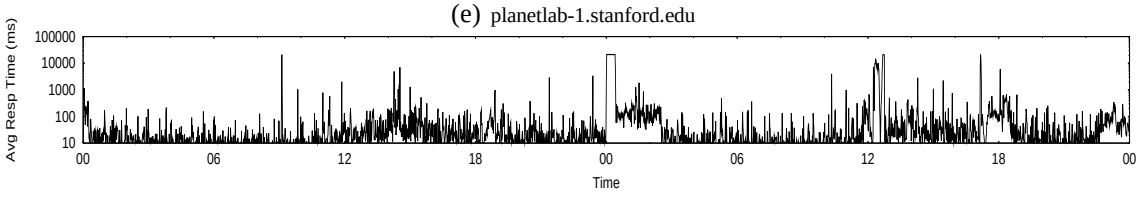
(c) planetlab-1.cmcl.cs.cmu.edu



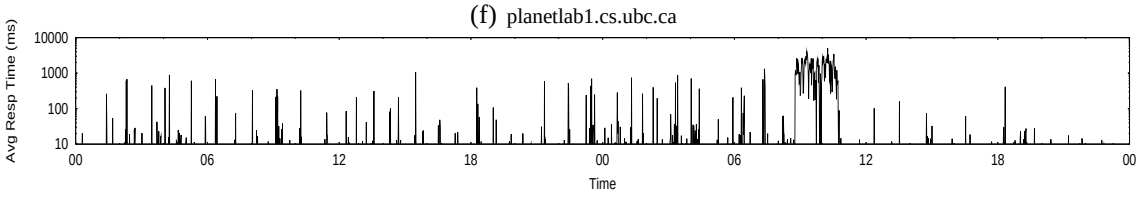
(d) kupl1.ittc.ku.edu



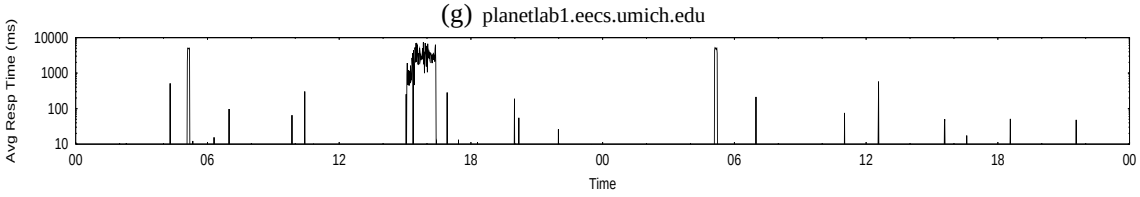
(e) planetlab-1.stanford.edu



(f) planetlab1.cs.ubc.ca

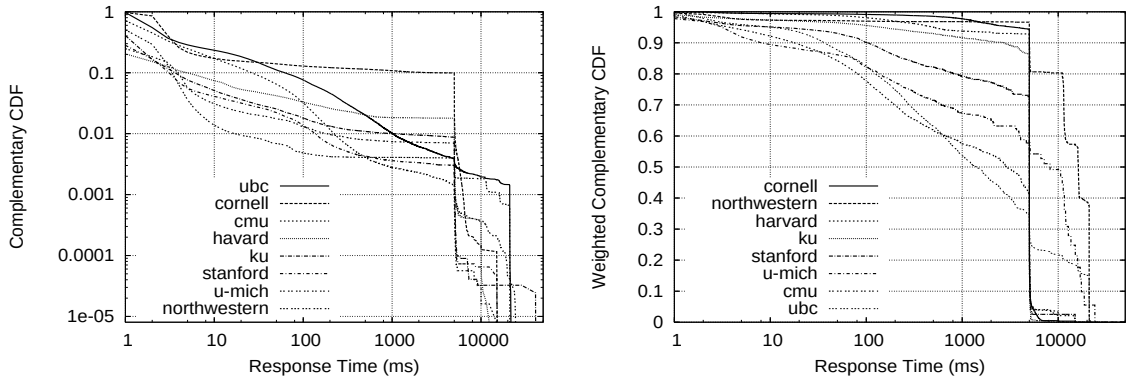


(g) planetlab1.eecs.umich.edu



(h) planetlab2.cs.northwestern.edu

Figure 4.1: Average cached DNS lookup response times on various PlanetLab nodes over two days. Note that while most Y axes span 10-1000 milliseconds, some are as large as 100,000 milliseconds.



(a) Fraction of Lookups Taking > X ms: Please note that Y-axis is also in log scale

(b) Fraction of the Sum of Lookups Taking > X ms

Figure 4.2: Complementary Cumulative Distribution of Cached DNS Lookups

4.1 Background & Analysis

While the Domain Name System was intended to be a scalable, distributed means of performing name-to-IP mappings, its flexible design has allowed it to grow far beyond its original goals. While most people would be familiar with it for Web browsing, many systems depend on fast and consistent DNS performance. Mail servers, Web proxy servers, and content distribution networks (CDNs) must all resolve hundreds or even thousands of DNS names in short periods of time, and a failure in DNS may cause a service failure, rather just delays.

The server-side infrastructure of DNS consists of hierarchically-organized name servers, with central authorities providing "root" servers and others delegated organizations handling "top-level" servers, such as ".com" and ".edu". Domain name owners are responsible for providing servers that handle queries for their names. While DNS users can manually query each level of the hierarchy in turn until the complete name has been resolved, most systems delegate this task to local nameserver machines. This approach has

performance advantages (e.g., caching replies, consolidating requests) as well as management benefits (e.g., fewer machines to update with new software or root server lists).

With local nameserver cache hit rates approaching 90% [42, 87], their performance impact can eclipse that of the server-side DNS infrastructure. However, local nameserver performance and reliability has not been well studied, and since it handles all DNS lookups for clients, its failure can disable other services. Our experiences with deploying the CoDeeN content distribution network motivated us to investigate this issue, since all CoDeeN nodes use the local nameservers at their hosting sites.

4.1.1 Frequency of Name Lookup Failures

To determine the failure properties of local DNS infrastructure, we systematically measure DNS lookup times on many PlanetLab nodes. In particular, across 40 North American sites, we perform a query once per second. We ask these nodes to resolve each other's names, all of which are cacheable, with long time-to-live (TTL) values of no less than six hours. Lookup times for these requests should be minimal, on the order of a few milliseconds, since they can be served from the local nameserver's cache. This diagnostic workload is chosen precisely because it is trivially cacheable, making local infrastructure failures more visible and quantifiable. Evaluation of DNS performance on live traffic, with and without CoDNS, is covered in Section 4.5.

Our measurements show that local DNS lookup times are generally good, but often degrade dramatically, and that this instability is widespread and frequent. To illustrate the widespread nature of the problem and its magnitude, Figure 4.1 show the lookup behavior over a two-day period across a number of PlanetLab nodes. Each point shows the per-minute average response time of name lookups. All the nodes show some sort of DNS lookup problems during the period, with lookups often taking thousands of milliseconds.

These problems are not consistent with simple configuration problems, but appear to be usage-induced or triggered by activity on the nameserver nodes. [1] For example, the Cornell node consistently shows DNS problems, with more than 20% of lookups showing high lookup times of over five seconds, the default timeout in the client's resolver library. These failed lookups are eventually retried at the campus's second nameserver, masking the first nameserver's failures. Since the first nameserver responds to 80% of queries in a timely manner, it is not completely misconfigured. Very often throughout the day, it simply stops responding, driving the per-minute average lookup times close to five seconds. The Harvard node also displays generally bad behavior. While most lookups are fine, a few failed requests every minute substantially increase the per-minute average. The Stanford node's graph shows periodic spikes roughly every three hours. This phenomenon is long-term, and we suspect the nameserver is being affected by heavy cron jobs. The Michigan node shows a 90 minute DNS problem, driving its generally low lookup times to above one second.

Although the average lookup times appear quite high at times, the individual lookups are mostly fast, with a few very slow lookups dominating the averages. Figure 4.2(a) displays the complementary cumulative distribution function (CCDF) of name lookup times over the same two days. With the exception of the Cornell node, 90% of all requests take less than 100ms on all nodes, indicating that caching is effective and that average-case latencies are quite low. Even the Cornell node works well most of the time, with over 80% of lookups being resolved within 6ms.

However, slow lookups dominate the total time spent waiting on DNS, and are large enough to be noticeable by end users. In Figure 4.2(b), we see the lookups shown by their contribution to the total lookup time, which indicates that a small percentage of failure

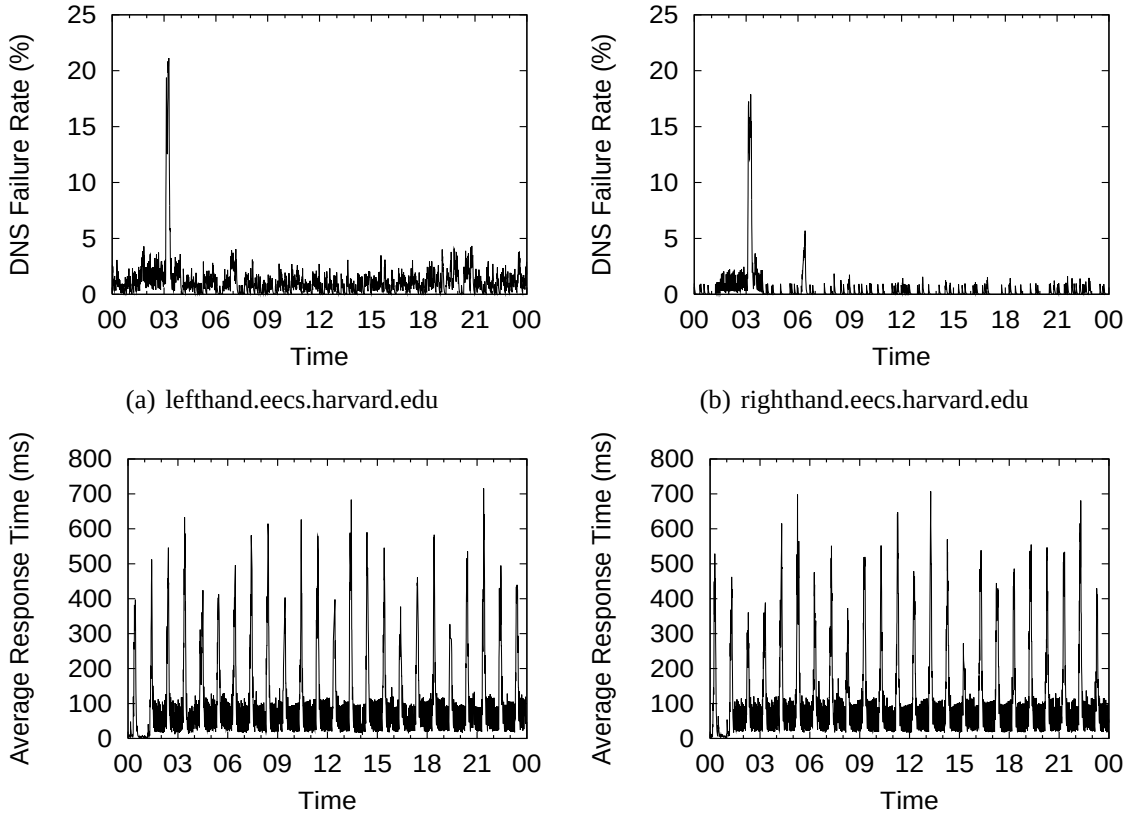
[1] More evidence is provided in the next section.

Node	Avg	Low	High	T-Low	T-High
cornell	531.7ms	82.4%	12.9%	0.5%	**99.2**%
harvard	99.4ms	92.3%	3.3%	0.7%	**97.9**%
cmu	24.0ms	81.9%	3.2%	8.3%	**71.0**%
ku	53.1ms	94.6%	1.8%	2.9%	**95.0**%
stanford	21.5ms	95.7%	1.3%	5.3%	**89.5**%
ubc	88.8ms	76.0%	7.6%	2.4%	**91.2**%
umich	43.6ms	96.7%	1.3%	2.4%	**96.1**%
northwestern	43.1ms	98.5%	0.5%	4.5%	**94.8**%

Table 4.1: Statistics over two days, Avg = Average, Low = Percentage of lookups < 10 ms, High = Percentage of lookups > 100 ms, T-Low = Percentage of total low time, T-High = Percentage of total high time

cases dominates the total time. This weighted CCDF shows, for example, that none of the nodes crosses the 0.5 value before 1000ms, indicating that more than 50% of the lookup time is spent on lookups taking more than 1000ms. If we assume that a well-behaving local nameserver can serve cached responses in 100ms, then the figures are even more dramatic. This data, shown in Table 4.1, shows that slow lookups comprise most of the lookup time.

These measurements show that client-side DNS infrastructure problems are significant and need to be addressed. If we can reduce the amount of time spent on these longer cases, particularly in the failures that require the local resolver to retry the request, we can dramatically reduce the total lookup times. Furthermore, given the sharp difference between “good” and “bad” lookups, we may also be able to ensure a more predictable (and hence less annoying) user experience. Finally, it is worth noting that these problems are not an artifact of PlanetLab – in all cases, we use the site’s local nameservers, on which hundreds or thousands of other non-PlanetLab machines depend. The PlanetLab nodes at a site see similar lookup times and failure rates, despite the fact that their other workloads may be very different. Examples from two sites are shown in Figure 4.3, and we can see that the nodes at a site see similar DNS performance. This observation further



(a) lefthand.eecs.harvard.edu (b) righthand.eecs.harvard.edu

(c) planetlab1.cs.purdue.edu (d) planetlab2.cs.purdue.edu

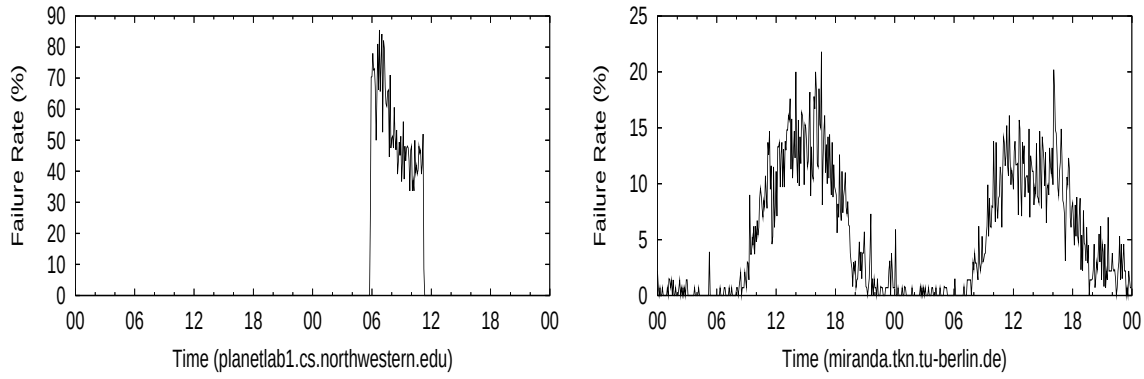
Figure 4.3: All nodes at a site see similar local DNS behavior, despite different workloads at the nodes. Shown above are one day's failure rates at Harvard, and one day's response times at Purdue.

enhances our claim that the problems are site-wide, and not PlanetLab-specific. [2]

4.2 Origin of Client-side Failures

While we do not have full access to all of the client-side infrastructure, we can try to infer the reasons for the kinds of failures we are seeing and understand their impact on lookup behavior. Absolute confirmation of the failure origins would require direct access to

[2] We have also confirmed this fact with a number of site administrators on PlanetLab, and helped them to fix the DNS problems on their sites.



(a) northwestern-1 (b) tu-berlin

Figure 4.4: Failures seemingly caused by nameserver overload – in both cases, the failure rate is always less than 100%, indicating that the server is operational, but performing poorly.

the nameservers, routers, and switches at the sites, which we do not have. Using various techniques, we can trace some problems to packet loss, nameserver overloading, resource competition, and maintenance issues. We discuss these below.

Packet Loss – The simplest cause we can guess is the packet loss in the LAN environment. Most nameservers communicate using UDP, so even a single packet loss either as a request or as a response would eventually trigger a query retransmission from the resolver. The resolver's default timeout for retransmission is five seconds, which matches some of the spikes in Figure 4.1.

Packet loss rates in LAN environments are generally assumed to be minimal, and our measurements of Princeton's LAN support this assumption. We saw no packet loss at two hops, 0.02% loss at three hops, and 0.09% at four hops, where the number of hops is measured between the same nameserver and source machines scattered around in the Princeton LAN. Though we did see bursty behavior in the loss rate, where the loss rates would stay high for a minute at a time, we do not see enough losses to account for the DNS failures. Our measurements show that 90% of PlanetLab nodes have a name-

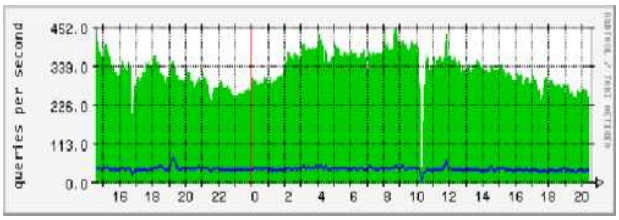


Figure 4.5: Daily Request Rate for Princeton.EDU

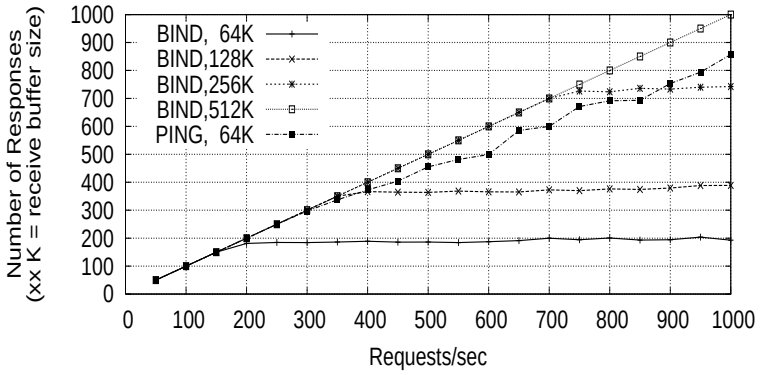


Figure 4.6: BIND 9.2.3 vs. PING with bursty traffic

server within 4 hops, and 70% are within 2 hops. However, other contexts, such as cable modems or dial-up services, have more hops [76], and may have higher loss rates.

Nameserver overloading – Since most request packets are likely to reach the nameserver, our next possible culprit is the nameserver itself. To understand their behavior, we asked all nameservers on PlanetLab to resolve a local name once every two seconds and we measured the results. For example, on planetlab-1.cs.princeton.edu, we asked for planetlab-2.cs.princeton.edu's IP address. In addition to the possibility of caching, the local nameserver is mostly likely the authoritative nameserver for the queried name, or at least the authoritative server can be found on the same local network.

In Figure 4.4, we see some evidence that nameservers can be temporarily overloaded. These graphs cover two days of traffic, and show the 5-minute average failure rate, where

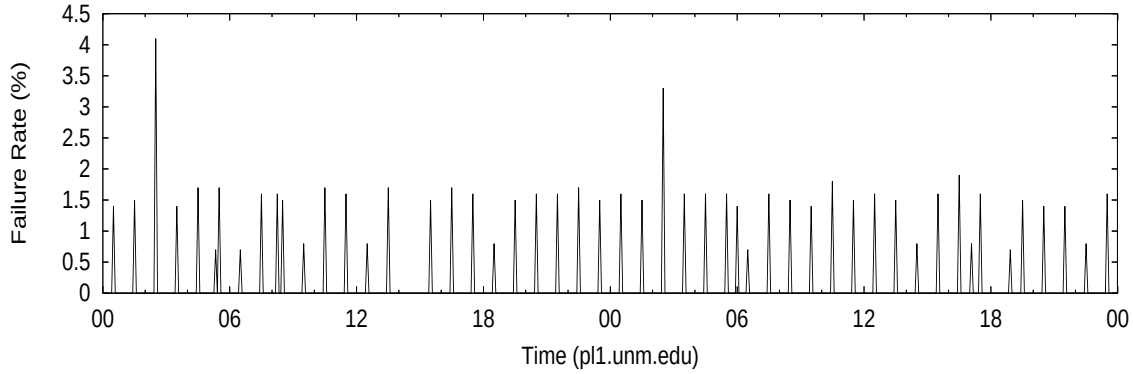


Figure 4.7: This site shows failures induced by periodic activity. In addition to the hourly failure spike, a larger failure spike is seen once per day.

a failure is either a response taking more than five seconds, or no response at all. In Figure 4.4(a), the node experiences no failures most of time but a 30% to 80% failure rate for about five hours. Figure 4.4(b) reveals a site where failures start during the beginning of the workday, gradually increase, and drop in the evening. It is reasonable to assume that human activity increases in these hours, and affects the failure rate.

We suspect that a possible factor in this overloading is the UDP receive buffer on the nameserver. These buffers are typically sized in the range of 32-64KB, and incoming packets are silently dropped when this buffer is full. If the same buffer is also used to receive the responses from other nameservers, as the BIND nameserver does, this problem gets worse. Assuming a 64KB receive buffer, a 64 byte query, and a 300 byte response, more than 250 simultaneous queries can cause packet dropping. In Figure 4.5, we see the request rate (averaged over 5 minutes) for the authoritative nameserver for princeton.edu. [3] Even with smoothing, the request rates are in the range of 250-400 reqs/sec, and we can expect that instantaneous rates are even higher. So, any activity that causes a 1-2 second delay of the server can cause requests to be dropped.

[3]By courtesy of Office of Information Technology (OIT) at Princeton University.

To test this theory of nameserver overload, we subjected BIND, the most popular nameserver, to bursty traffic. On an otherwise unloaded box (Compaq au600, Linux 2.4.9, 1 GB memory), we ran BIND 9.2.3 and an application-level UDP ping that simulates DNS request and response. Each request contains the same name query for a local domain name with a different query ID. Our UDP ping responds to it by sending a fixed response with the same size as BIND's. We send a burst of N requests from a client machine and wait 10 seconds to gather responses. Figure 4.6 shows the difference in responses between BIND 9.2.3 and our UDP ping. With the default receive buffer size of 64KB, BIND starts dropping requests at bursts of 200 reqs/sec, and the capacity linearly grows with the size of the receive buffer. Our UDP ping using the default buffer loses some requests due to temporary overflow, but the graph does not flatten because responses consume minimal CPU cycles. These experiments confirm that high-rate bursty traffic can cause server overload, aggravating the buffer overflow problem.

Resource competition – Some sites show periodic failures, similar to what is seen in Figure 4.7. These tend to have spikes every hour or every few hours, and suggests some heavy process is being launched from regularly scheduled task (e.g. a cron job). BIND is particularly susceptible to memory pressure, since its memory cache is only periodically flushed [3]. Any jobs that use large amounts of memory can evict BIND's pages, causing BIND to page fault when accessing the data. The faults can delay the server, causing the UDP buffer to fill.

In talking with system administrators, we find that even sites with good DNS service often run multiple services (some cron-initiated) [4] on the same machine. Since DNS is regarded as a low-CPU service, other services are run on the same hardware to better

[4]One site administrator told us that they are running DHCP/BOOTP, LDAP, XTACACS and RADIUS services on their nameserver machine.

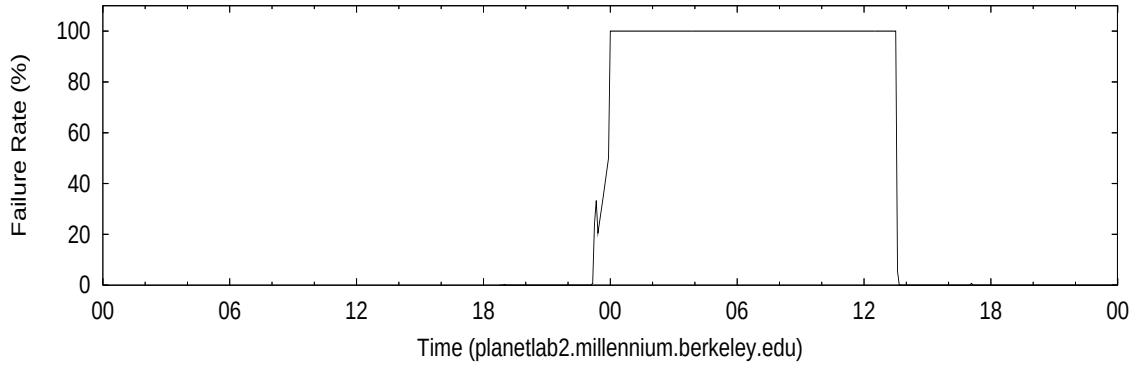


Figure 4.8: This site's nameservers were shut down before the nodes had been updated with the new nameserver information. The result was a 13-hour complete failure of all name lookups, until the information was manually updated.

utilize the resource. It seems quite common that when these other services have bursty resource behavior, the nameserver is affected.

Maintenance problems – Another common source of failure is maintenance problems which lead to service interruption, as shown in Figure 4.8. Here, the DNS lookup shows a 100% failure rate for 13 hours. Both nameservers for this site stopped working causing DNS to be completely unavailable, instead of just slow. DNS service was restored only after manual intervention. Another common case, complete failure of the primary nameserver, generates a similar pattern, with all responses being retried after five seconds and sent to the secondary nameserver.

4.3 CoDNS Design

In this section, we discuss the design of CoDNS, a name lookup system that provides faster and more reliable DNS service while minimizing extra overhead. We also discuss overheads and benefits of various desigin choices, using trace-driven workloads.

One important goal shapes our design: our system should be incrementally deployable, not only by DNS administrators, but also by individual users. The main reason for this decision is that it bypasses the bureaucratic processes involved with replacing the existing DNS infrastructure. Given the difficulty we have in even getting information about local DNS nameservers, the chances of convincing system administrators to send their live traffic to an experimental name lookup service seems low. Providing a migration path that coexists with the existing infrastructure allows people the opportunity to grow comfortable with the service over time.

Another implication of this strategy is that we should aim for minimal resource commitments. In particular, we should leverage the existing infrastructure devoted to making DNS performance generally quite good. Client-side nameservers achieve high cache hit rates by devoting memory to name caching, and if we can take advantage of the existing infrastructure, it lessens the cost of deployment. While current client-side infrastructure, including nameservers, is not perfect, it provides good performance most of the time, and it can provide a useful starting point. Low resource usage also reduces the chances for failure due to resource contention.

Our usage model is cooperative, operating similarly to insurance – nodes join a pool that shares resources in times of need. If a node's local lookup performance is acceptable, it proceeds as usual, but may have to provide service to nodes that are having problems. When its local performance degrades, it can ask other nodes to help it. The benefit of joining is the ability to get help when needed, even if there is some overhead at other times.

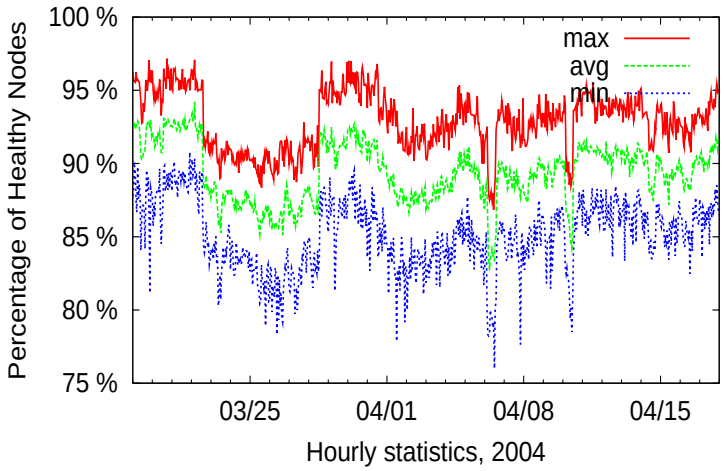


Figure 4.9: Hourly % of nodes with working nameservers

4.3.1 Cross-site Correlation of DNS Failures

The "insurance" model depends on failure being relatively uncorrelated – the system must always have a sufficient pool of working participants to help those having trouble. If failure across sites is correlated, this assumption is violated, and a cooperative lookup scheme is less feasible.

To test our assumption, we study the correlation of DNS lookup failures across PlanetLab. At every minute, we record how many nodes have "healthy" DNS performance. We define healthy as showing no failures for one minute for the local domain name lookup test. Using the per-minute data for March 2004, we show the minimum, average and maximum number of nodes available per hour. The percentage of healthy nodes (as a fraction of live nodes) is shown in Figure 4.9.

From this graph, we can see some minor correlation in failures, [5] shown as downward spikes in the percentage of available nodes, but most of the variation in availability seems largely uncorrelated. An investigation into the spikes reveals that many nodes on

[5] All Internet2 hosting nodes at PlanetLab and all Chinese 6PlanetLab nodes are set up to have the same set of nameservers. The minor correlation indicates the failure of those nameservers.

Software	PlanetLab	Packetfactory	TLD
BIND-4.9.3+/8	31.1%	36.4%	55.9%
BIND 9	48.9%	25.1%	34.0%
Other	20.0%	38.5%	10.1%

Table 4.2: Comparison of nameserver software used by PlanetLab, packetfactory survey and the TLD survey

PlanetLab are configured to use the same set of nameservers, especially those co-located at Internet2 backbone facilities (not to be confused with Internet2-connected university sites). When these nameservers experience problems, the correlation appears large due to the number of nodes affected.

More important, however, is the observation that the fraction of healthy nameservers is always high, generally above 90%. This observation provides the key insight for CoDNS – with enough healthy nameservers, we can mask locally-observed delays via cooperation.

To ensure that these failures are not tied to any specific nameserver software, we survey the software running on the local nameservers used by the PlanetLab nodes (135 unique nameservers) with "chaos" class [6] queries [53]. We find that they are mostly running a variety of BIND versions. We observe 11 different BIND 9 version strings, 13 different BIND 8 version strings and a number of humorous strings (included in "other") apparently set by the nameserver administrators. These measurements, shown in Table 4.2, are in line with two nameserver surveys conducted by Brad Knowles in 2002 [47] and by packetfactory in 2003 [74]. From this, we conclude that the failures are not likely to be specific to PlanetLab's choices of nameserver software.

[6] This is to use TXT resource record (RR) in class 3 for the domain name "HOSTNAME.BIND.". Then the BIND server is supposed to return its identification string configured by the name server administrator.

4.3.2 CoDNS

The main idea behind CoDNS is to forward name lookup queries to peer nodes when the local name service is experiencing a problem. Essentially, this strategy applies a CDN approach to DNS – spreading the load among peers improves the size and performance of the "global cache". Many of the considerations in CDN systems apply in this environment. We need to consider the proximity and availability of a node as well as the locality of the queries. A different consideration is that we need to decide when it is desirable to send remote queries. Given the fact that most name lookups are fast in the local nameserver, simply spreading the requests to peers might generate unnecessary traffic with no reduction in latency. Worse, the extra load may cause some DNS nameservers to become overloaded. We investigate considerations for deciding when to send remote queries, how many peers to involve, and what sorts of gains to expect.

To precisely determine the effects of locality, load, and proximity is difficult, since we have no control over the nameservers and have little information about their workloads, configurations, etc. The proximity of a peer server is important in that DNS response time can be affected by its peer to peer latency. Since the DNS requests and responses are not large, we are more interested in picking nearby peers with low round-trip latency instead of nodes with particularly high bandwidth. We have observed coast-to-coast round-trip ping times of 80ms in CoDeeN, with regional times in the 20ms range in America. Both of these ranges are much lower than the DNS timeout value of five seconds, so, in theory, any node in the U.S. would be an acceptable peer. In practice, choosing closer peers will reduce the difference between cache hit times and remote peer times, making CoDNS failure masking more effective. For request locality, we would like to increase the chances of remote queries being cache hits in the remote nameservers. Using any scheme that consistently partitions this workload will help improve the likelihood of cache hits.

To understand the relationship between CoDNS response times, the number of peers involved, and the policies for determining when requests should be sent remotely, we collected 44,486 unique host names from one day's HTTP traffic on CoDeeN and simulated various policies and their effects. We replayed DNS lookups of those names at 77 PlanetLab nodes with different nameservers, starting requests at the same time of day in the original logs. The replay happened one month after the data collections to avoid local nameserver caches which could skew the data. During this time, we also use application-level heartbeat measurements between all pairs of the 77 PlanetLab nodes to determine their round-trip latencies. Since all of the nodes are doing DNS lookups at about the same time, by adding the response time at peerY to the time spent for the heartbeat from peerX to peerY, we will get the response time peerX can get if it asks peerY for a remote DNS lookup for the same host name.

An interesting question is how many simultaneous lookups are needed to achieve a given average response time and to reduce the total time spent on slow lookups (defined as taking more than 1 second). As shown in the previous section, it is desirable to reduce the number of slow responses to reduce the total lookup time. Figures 4.10 and 4.11 show two graphs answering this question. The lookup scheme here is to contact the local nameserver first for a name lookup, wait for a timeout and issue x-1 simultaneous lookups using x-1 randomly-selected peer nodes. Figure 4.10 shows that even if we use only one extra lookup, we can reduce the average response time by more than half. Also, beyond about five peers, adding more simultaneous lookups produces diminishing returns. Different initial timeout values do not produce much difference in response times, because the benefit largely stems from reducing the number of slow lookups. The slow response portion graph (Figure 4.11) proves this phenomenon, showing similar reduction in the slow response percentage at any initial timeout less than 700ms.

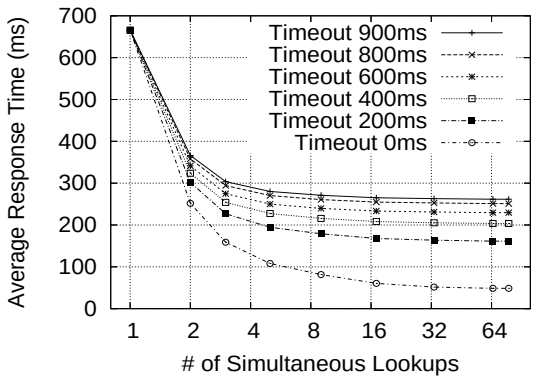


Figure 4.10: Average Response Time

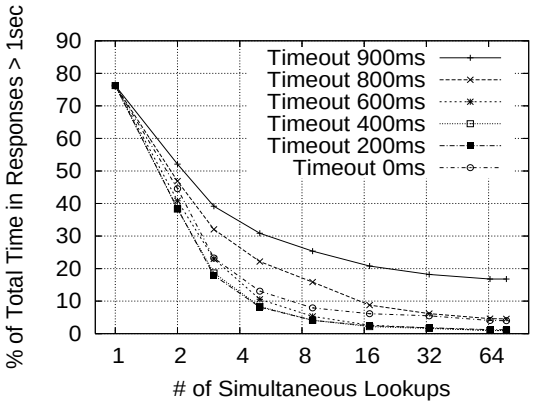


Figure 4.11: Slow Response Time Portion

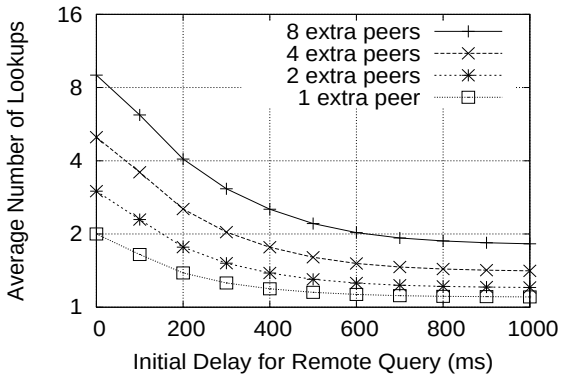


Figure 4.12: Extra DNS Lookups

We must also consider the extra overhead of the simultaneous lookups, since shorter initial timeouts and more simultaneous lookups cause more DNS traffic at all peers. Figure 4.12 shows the overhead in terms of extra lookups needed for various scenarios. Most curves start to flatten at a 500ms initial timeout, providing only diminishing returns for larger timeouts. Worth noting is that even with one peer and a 200ms initial timeout, we can still cut the average response time by more than half, with only 38% extra DNS lookups.

These results are very encouraging, demonstrating that CoDNS can be effective even

at very small scale – even a single extra site provides significant benefits, and it achieves most of its benefits with less than 10 sites. The reasons for this scale being important is twofold: only small commitments are required to try a CoDNS deployment, and DNS's limitations with respect to trust and verification (discussed in the next section) are unlikely to be an issue at these scales.

4.3.3 Trust, Verification and Implications

Some aspects of DNS and its design invariably impact our approach, and the most important is trust and verification. The central issue is whether it is possible for a requester to determine that its peer has correctly resolved the request, and that the result provided is actually a valid IP address for the given name. This issue arises if peers can be compromised or are otherwise failing.

Unfortunately, we believe that a general solution to this problem is not possible with the current DNS, though certain fault models are more amenable to checking than others. For example, if the security model assumes that at most one peer can be compromised, it may be possible to always send remote requests to at least three peers. When these nodes respond, if two results agree, then the answer must be correct. However, DNS does not mandate that any of these results have to agree, making the general case of verification impossible.

Many server-side DNS deployments use techniques to improve performance and reliability, or balance load and locality. For example, round-robin DNS can return results from a list of IP addresses in order to distribute load across a set of servers. Geography-based redirection can be used to reduce round-trip times between clients and servers by having DNS lookups resolve to closer servers. Finally, DNS-based content distribution networks will often incorporate load balancing and locality considerations when resolv-

ing their DNS names. In these cases, multiple lookups may produce different results, and lookups from different locations may receive results from different pools of IP addresses.

While it would be possible to imagine extending DNS such that each name is associated with a public key, and each IP address result is signed with this key, such a change would be significant. DNSSEC [29] attempts smaller-scale change, mainly to prevent DNS spoofing, but has been in some form of development for nearly a decade, and still has not seen wide-scale adoption.

Given the current impossibility of verifying all lookups, we rely on trusting peers in order to sidestep the problems mentioned. This approach is already used in various schemes. Name owners often use each other as their secondary servers, sometimes at large scale. For example, princeton.edu's DNS servers act as the secondary servers for 60 non-Princeton domains. BIND supports zone transfers, where all DNS information can be downloaded from another node, specifically for this kind of scenario. Similarly, large-scale distributed systems running at hosting centers already have a trust relationship in place with their hosting facility.

4.4 Implementation

We have built CoDNS and have been running it on all nodes on PlanetLab since August 2003. During that time, we have been directing the CoDeeN CDN to use CoDNS for name lookup.

CoDNS consists of a stand-alone daemon running on each node, accessible via UDP for remote queries, and via loopback TCP connection for locally-originated name lookups. The daemon is event-driven, and is implemented as a non-blocking master process and many (blocking) slave processes. The master process receives name lookup requests from

local clients and remote peers, and passes them to one of its idle slaves. A slave process resolves those names by calling `gethostbyname()` and sends the result back to the master. Then, the master returns the final result to either a local client or a remote peer depending on where it originated. Whenever a new query arrives, CoDNS checks in its queue if there is an outstanding query resolving the same host, coaleses it into the same query and answers together when resolved. Preference for idle slaves is given to locally-originated requests over remote queries to ensure good performance for local users.

The master process records each request's arrival time from local clients and sends a UDP name lookup query to a peer node when the response from the slave has not returned within a certain period. This delay is used as a boundary for deciding if the local nameserver is slow. In the event that neither the local nameserver nor the remote node has responded, CoDNS doubles the delay value before sending the next remote query to another peer. In the process, whichever result that comes first will be delivered as the response for the name lookup to the client. Peers may silently drop remote queries if they are overloaded, and remote queries that fail to resolve are also discarded. Slaves may add delay if they receive a locally-generated request that fails to resolve, with the hope that remote nodes may be able to resolve such names.

4.4.1 Remote Query Initiation and Retries

The initial delay before sending the first remote query is dynamically adjusted based on the recent performance of local nameservers and peer responses. In general, when the local nameserver performs well, we increase the delay so that fewer remote queries are sent. When most remote answers beat the local ones, we reduce the delay preferring the remote source. Specifically, if the past 32 name lookups are all resolved locally without using any remote queries, then the initial delay is set to 200ms by default. We choose 200ms

because the median response time on a well-functioning node is less than 100ms [42], so 200ms delay should respond fast during instability, while wasting a minimal amount for extra remote queries.

However, to respond quickly to local nameserver failure, if the remote query wins more than 50% of the last 16 requests, then the delay is set to 0 ms. That is, the remote query is sent immediately as the request arrives. Our test results show it is rare not to have failure when more than 8 out of 16 requests take more than 300ms to resolve, so we think it is reasonable to believe the local nameserver is having a problem in that case. Once the immediate query is sent, the delay is set to the average response time of remote query responses plus one standard deviation, to avoid swamping fast remote servers.

4.4.2 Peering and Query Distribution

Each CoDNS node gathers and manages a set of peer nodes (neighbors) within a reasonable latency boundary. Independent peering and pairwise monitoring strategy [7] also applies here, but the peering decision is simpler than the case of CoDeeN, because we are primarily interested in the reliable DNS service. "Good" neighbors are determined by the health of the network path to the neighbor and the expected remote query response time, which is calculated as the sum of round-trip time (RTT) and the rolling average of local DNS lookup response time at the neighbor site. The average local DNS response time should reflect the recent failures, so we can avoid any misbehaving remote sites.

When a CoDNS instance starts, it sends a heartbeat to each node in the preconfigured CoDNS node list every second, and the response contains RTT and the average local DNS response time, reflecting peer nameserver's proximity and availability. The top 10 nodes with different nameservers are picked as neighbors by comparing with all nodes the

[7] Please see Section 2.2.

expected DNS lookup time from the source node. Given the experiments in Section 4.3.2, 10 different nameservers are enough to provide the most benefit. Liveness of the chosen neighbors is periodically checked to see if the service is still available. One heartbeat is sent each second, so we guarantee the availability at a 10 second granularity. Dead nodes are replaced with the next best nodes in the list.

Among these neighbor nodes, one peer is chosen for each remote name lookup using the Highest Random Weight (HRW) hashing scheme [82] discussed in Section 2.1. Because HRW consistently picks the same node for the same domain name, this process enhances request locality for remote queries. Another desirable property of this approach is that some request locality is preserved as long as neighbor sets have some overlap. Better request locality can be obtained using the query re-forwarding scheme (which will be discussed in Section 5.4.2), but CoDNS favors one-hop routing because the response low latency is more desirable.

The number of neighbors is manually configurable by considering the distribution of nodes. In the future, we may make CoDNS dynamically find the peer nodes not depending on the preconfigured set of nodes. One possible solution is to make each CoDNS node advertise its neighbor set and have a few well known nodes. Then, a new CoDNS node with no information about available CoDNS peer nodes can ask the well known nodes for their peer nodes and recursively gather the nodes by asking each neighbor until it finds a reasonable pool of CoDNS nodes.

Note that our neighbor discovery mechanisms are essentially advisory in nature – once the node has enough peers, it only needs to poll other nodes in order to have a reasonable set of candidates in case one of its existing peers becomes unavailable. In the event that some sites have enough peers to make this polling a scalability issue, each node can choose to poll a nearby subset of all possible peers to reduce the background traffic.

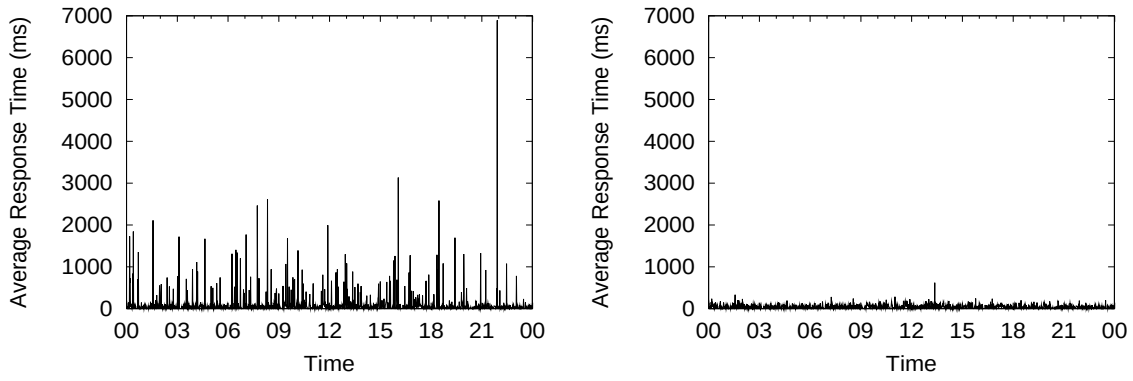
4.4.3 Policy and Tunability

In the future, we expect CoDNS node configuration policy will become an interesting research area, given the tradeoffs between overhead and latency. We have made choices for initial delay and retry behavior for our environment, and we believe that these choices are generally reasonable. However, some systems may choose to tune CoDNS to have much lower overhead, at the cost of some latency savings. In particular, systems that want to use CoDNS only to avoid situations where all local nameservers have failed could use an initial delay threshold of several seconds. In this case, if the local nameserver repeatedly fails to resolve requests in multiple seconds, the initial delay will drop to zero and all lookups will be handled remotely for the duration of the outage.

Sites may also choose to limit CoDNS overhead to a specific level, which would turn parameter choices into an optimization problem. For example, it may be reasonable to ask questions of the form "what is the best latency achievable with a maximum remote lookup rate of 10%?" Our trace-driven simulations give some insight into how to make these choices, but it may be desirable to have an online system automatically adjust parameter values continuously in order to meet these constraints. We are investigating policies for such scenarios.

4.4.4 Bootstrapping

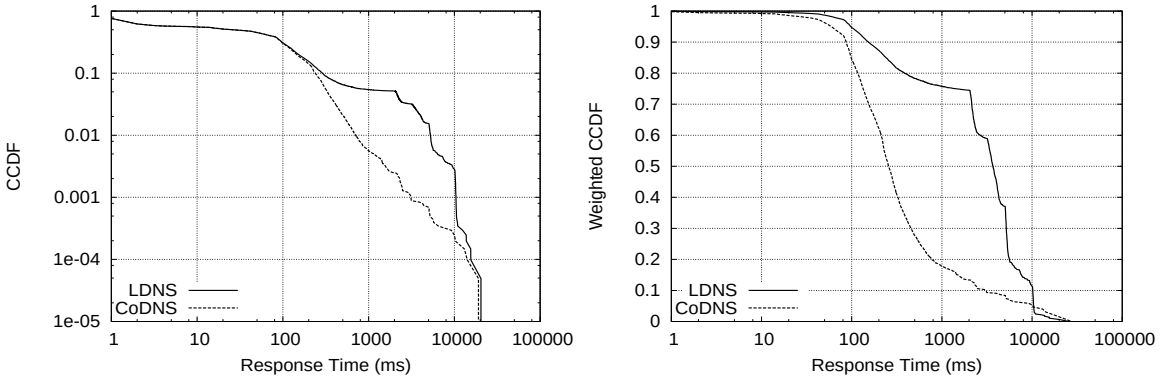
CoDNS has a bootstrapping problem, since it must resolve peer names in order to operate. In particular, when the local DNS service is slow, resolving all peer names before starting will increase CoDNS's start time. So, CoDNS begins operation immediately, and starts resolving peer names in the background, which greatly reduces its start time. The background resolver uses CoDNS itself, so as soon as a single working peer's name is



(a) Local DNS (b) CoDNS

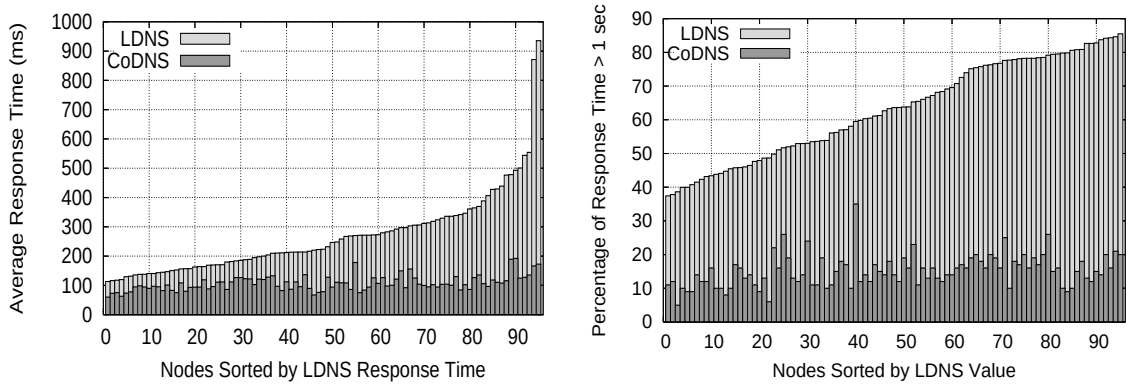
Figure 4.13: Minute-level Average Response Time for One Day on planetlab1.cs.cornell.edu

resolved, it can then quickly help resolve all other peer names. With this bootstrapping approach, CoDNS starts virtually instantaneously, and can resolve all 350 peer names in less than 10 seconds, even for slow local DNS. A special case of this problem is starting when local DNS is completely unavailable. In this case, CoDNS would be unable to resolve even any peer names, and could not send remote queries. CoDNS periodically stores all peer information on disk, and uses that information at startup. This file is shipped with CoDNS, allowing operation even on nodes that have no DNS support at all.



(a) Response Time CCDF: Y-axis is in log scale (b) Total Time CCDF

Figure 4.14: CCDF and Weighted CCDF for One Week on planetlab1.cs.cornell.edu, LDNS = local DNS



(a) Average Response Time (b) Slow Response Time portion

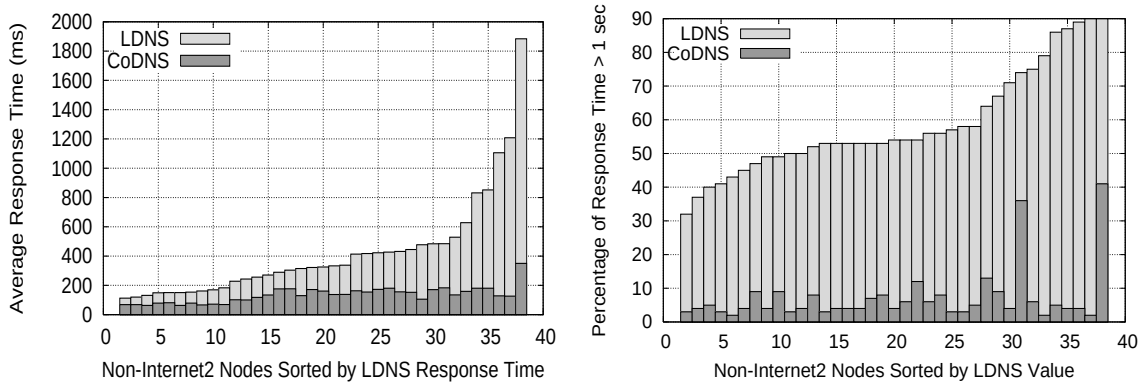
Figure 4.15: Live Traffic for One Week on the CoDeeN Nodes, LDNS = local DNS

4.5 Evaluation / Live Traffic

To gauge the effectiveness of CoDNS, we compare its behavior with local DNS on CoDeeN's live traffic using a variety of metrics. CoDeeN receives about 20 million requests daily from a world-wide client population of 50-70K users. These users have explicitly specified CoDeeN proxies in their browser, so all of their Web traffic is directed through CoDeeN. The CoDeeN proxies maintain their own DNS caches, so only uncached DNS names cause lookups. To eliminate the possible caching effect on a nameserver from other users sharing the same server, we measure both times only in CoDNS, using the slaves to indicate local DNS performance.

CoDNS effectively removes the spikes in the response time, and provides more reliable and predictable service for name lookups. Figure 4.13 compares per-minute average response times of local DNS and CoDNS for CoDeeN's live traffic for one day on one PlanetLab node. While local DNS shows response time spikes of 7 seconds, CoDNS never exceeds 0.6 seconds throughout the day. The benefit stems from redirecting slow name lookups to CoDNS peers with working nameservers.

The greater benefit of CoDNS lies in reducing the frequency of slow responses. Fig-

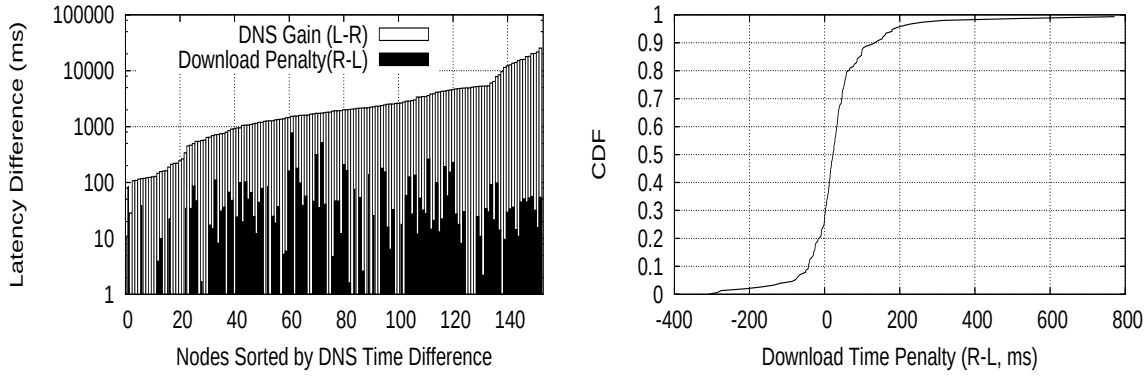


(a) Average Response Time (b) Slow Response Time Portion

Figure 4.16: Non-Internet-2 Nodes, LDNS = local DNS

ure 4.14 shows a CCDF and a weighted CCDF for name lookup response distribution for the same node for one week. The CCDF graph shows that the response distribution in both schemes is almost the same until they reach the 10th percentile, but CoDNS reduces the lookups taking more than 1000ms from 5.5% to 0.6%. This reduction greatly affects the total lookup time in the weighted CCDF. It shows CoDNS now spends 18% of total time in lookups taking more than 1000ms, while local DNS still spends 75% of the total time on them.

This improvement is widespread – Figure 4.15(a) shows the statistics of 95 CoDeeN nodes for the same period. The average number of total lookups per node is 22,208, ranging from 12,119 to 131,466 per node. The average response time in CoDNS is 60-221ms, while that of local DNS is 113-935ms. In all cases, CoDNS's response is faster, ranging from a factor of 1.37 to 5.42. Figure 4.15(b) shows the percentage of slow responses in the total response time. CoDNS again reduces the slow response's portion dramatically to less than 20% of the total lookup time in most cases, delivering more predictable response time. In contrast, local DNS spends 37% to 85% of the total time in the slow queries.



(a) DNS Lookup Time Gain vs. Downloading Time Penalty (b) Cumulative Distribution of Downloading Time Difference

Figure 4.17: CDN Effect for www.apple.com, L = Local Response Time, R = Remote Response Time, DNS gain = Local DNS time - CoDNS time, Download penalty = download time of CoDNS-provided IP - download time of DNS-provided IP, shown in log scale. Negative penalties indicate CoDNS-provided IP is faster, and are not shown in the left graph.

4.5.1 Non-Internet2 Benefits

Since many of the oDeeN sites are hosted at North American universities with Internet2 (I2) connectivity, one may suspect that low-congestion I2 peer links are responsible for our benefits. [8] To address this issue, we pick non-I2 PlanetLab nodes and replay 10,792 unique lookups of host names from one day's live traffic on a CoDeeN proxy. Figure 4.16(a) shows that CoDNS provides similar benefit on 38 non-I2 nodes as well. The average response time in CoDNS ranges from 63ms to 350ms, while local DNS is 113ms to 1884ms, an improvement of factor of 1.64 to 9.52. Figure 4.16(b) shows that CoDNS greatly reduces the slow response portion as well – CoDNS generally spends less than 10% of the total time in this range, while local DNS still spends 32% to 90%.

[8] Internet2 is a non-profi t network consisting of over 200 universities in the U.S. with 10 Gbps backbone that supports 100+ Mbps bandwidth for any pair of member sites.

4.5.2 Effects on CDNs

CoDNS replaces slow local responses with fast remote responses, which may impact DNS-based CDNs [2] that resolve names based on which DNS nameserver sends the query. CoDNS may return the address of a far replica when it uses a peer's nameserver result. We investigate this issue by testing 14 popular CDN customers including Apple, CNN, and the New York Times. We measure the DNS and download time of URLs for the logo image file on those web sites, and compare local DNS and CoDNS when their responses differ.

Since CoDNS is used only when the local DNS is slow or failing, it should come as no surprise that the total time for CDN content is still faster on CoDNS when lookup responses differ in returned IP address. The DNS time gain and the downloading time penalty presented in the difference between local and remote response time is shown in Figure 4.17(a). When local DNS is slow, CoDNS combined with a possibly sub-optimal CDN node is a much better choice, with the gain from faster name lookups dwarfing the small difference in download times when any difference exists. If we isolate the downloading time difference between the DNS-provided CDN node versus the CoDNS-provided CDN node, we get Figure 4.17(b). Surprisingly, almost a third of the CoDNS-provided nodes are closer than their DNS counterparts, and 83% of them show less than a 100ms difference. This matches the CDN's strategy to avoid notably bad servers instead of choosing the optimal server [41]. One may argue that this approach may result in "wrong" contents sometimes from different replicas possibly customized for their own regions. However, the "wrong" contents will not persist over time, because the remote responses will be used temporarily when the local DNS experiences some problems. Results for other CDN vendors are similar.

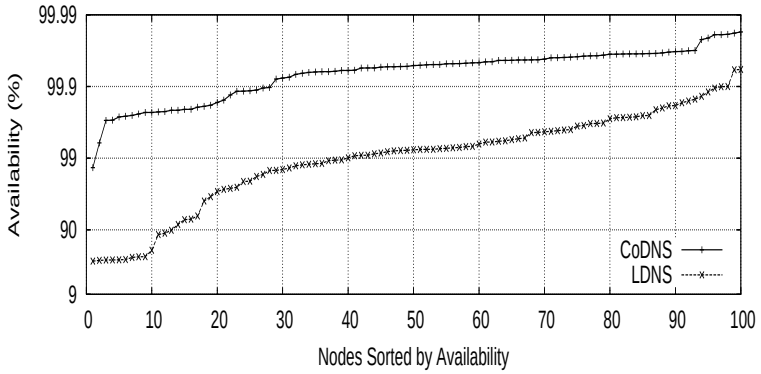


Figure 4.18: Availability of CoDNS and local DNS (LDNS)

4.5.3 Reliability and Availability

CoDNS dramatically improves DNS reliability, measured by the local nameserver availability. To quantify this effect, we measured the availability of name lookups for one month across all CoDeeN nodes, with and without CoDNS. We assume that a nameserver is available unless it fails to answer requests. If it fails, we consider the periods of time when no requests were answered as its unavailability. Each period is capped at a maximum of five seconds, and the total unavailability is measured as the sum of the unavailable periods. This data, shown in Figure 4.18, is presented using the reliability metric of "9's" of availability. Regular DNS achieves 99% availability on about 60% of the nodes, which means roughly 14 minutes per day of no service. In contrast, CoDNS is able to achieve over 99.9% availability on over 70% of nodes, reducing downtimes to less than 90 seconds per day. On some nodes, the availability approaches 99.99%, or roughly 9 seconds of unavailability per day. CoDNS provides roughly an additional '9' of availability, without any modifications to the local DNS infrastructure.

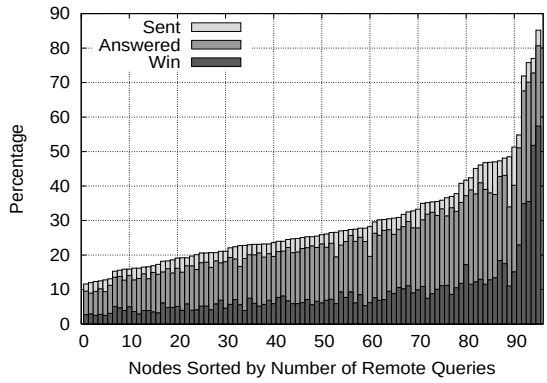


Figure 4.19: Analysis for Remote Lookups

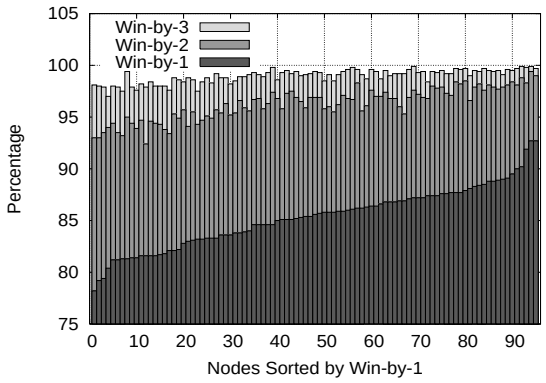


Figure 4.20: Win-by-N for Remote Lookups

4.5.4 Overhead Analysis

To analyze CoDNS's overhead, we examine the remote query traffic generated by the CoDeeN live activity. For this workload, CoDNS issued 11% to 85% of the total lookups as remote queries, as shown in Figure 4.19. The variation reflects the health of the local nameserver, and less stable nameservers require more remote queries from CoDNS. Of the six nodes that had more than 50% remote queries, all experienced complete nameserver failure at some point, during which remote queries increased to over 100% of the local requests. These periods skew the average overhead.

We believe that the additional burden on nodes with working DNS is tolerable, due to the combination of our locality-conscious redirection and already high local nameserver hit rates. Using our observed median overhead of 25% and a local hit rate of 80% - 87% [42], the local DNS will incur only 3.25 - 5.00% extra outbound queries. Since remote queries are redirected only to lightly loaded nodes, we believe the extra lookups will be tolerable on the peer node's local nameserver.

We also note that many remote queries are not answered, with Figure 4.19 showing this number varies from 6% to 31%. These can be due to WAN packet losses, unresolvable names, and remote node rate-limiting. CoDNS nodes drop remote requests if too

many are queued, which prevents a possible denial of service attack. CoDNS peers never reply if the request is unresolvable, since their own local DNS may be failing, and some other peer may be able to resolve the name.

The queries in which CoDNS "wins", by beating the local DNS, constitute 2% to 57% of the total requests. On average, 9% of the original queries were served by the remote responses, removing 47% of the slow response portion in the total lookup time shown in the Figure 4.15(b). Of the winning remote responses, more than 80% were answered by contacting the first peer, specified as "win-by-1" in Figure 4.20. Of all winning responses, 95% are resolved by the first or second peer, and only a small number require contacting three or more peers. This information can be used to further reduce CoDNS's overhead by reducing the number of peers contacted – if it has not been resolved within the first three peers, then further attempts are unlikely to resolve it, and no more peers should be contacted. We may explore this optimization in the future, but our current overheads are low enough that we have no pressing need to reduce them.

In terms of extra network traffic generated for remote queries, each query contains about 300 bytes of a request and a response. On average, each CoDNS on a CoDeeN node handles 414 to 10,287 requests per day during the week period, amounting to 243KB to 6027KB. CoDNS also consumes heartbeat messages to monitor the peers each second, which contains 32 bytes of data. In sum, each CoDNS on a CoDeeN node consumes on average 7.5 MB of extra network traffic per day, consuming only 0.2% of total CoDeeN traffic in relative terms.

4.5.5 Application Benefits

By using CoDNS, CoDeeN obtains other benefits in capacity and availability, and these may apply to other applications as well. The capacity improvements come from CoDeeN

being able to use nodes that are virtually unusable due to local DNS problems. At any given time, roughly 10 of the 100 PlanetLab nodes that run CoDeeN are experiencing significant DNS problems, ranging from high failure rates to complete failure of the primary (and even secondary) nameservers. CoDeeN nodes normally report their local status to each other, and before CoDNS, these nodes would tell other nodes to avoid them due to the DNS problems. With CoDNS, these nodes can still be used, providing an additional 10% extra capacity.

The availability improvements come from reducing startup time, which can be dramatic on some nodes. CoDeeN software upgrades are not announced downtimes, because on nodes with working local DNS, CoDeeN normally starts in 10-15 seconds. This startup process is fast enough that few people notice a service disruption. Part of this time is spent in resolving the names of all CoDeeN peers, and when the primary DNS server is failing, each lookup normally requires over five seconds. For 120 peers, this raises the startup time to over 10 minutes, which is a noticeable service outage. If CoDNS is already running on the node, startup times are virtually unaffected by local failure, since CoDNS is already sending all queries to remote servers in this environment. If CoDNS starts concurrently with CoDeeN, the startup time for CoDeeN is roughly 20 seconds.

4.6 Other Approaches

4.6.1 Private Nameservers

Since local nameservers exhibit overload, one may be tempted to run a private nameserver on each machine, and have it contact the global DNS hierarchy directly. This approach is more feasible as a backup mechanism than as a primary nameserver for several reasons. Using shared nameservers reduces maintenance issues, and the shared cache can be larger

than individual caches. Not only does cache effectiveness increase due to capacity, but the compulsory misses will also be reduced from the sharing. With increased cache misses, the global DNS failure rate becomes more of an issue, so using private nameservers may reduce performance and reliability.

As a backup mechanism, this approach is possible, but has the drawbacks common to any infrequently-used system. If the backup system is not used regularly, failure is less likely to be noticed, and the system may be unavailable when it is needed most. It also consumes resources when not in use, so other tasks on the same machine will be impacted, if only slightly.

4.6.2 Secondary Nameservers

Since most sites have two or more local nameservers, another approach would be to modify the resolver libraries to be more aggressive about using multiple nameservers. Possible options include sending requests to all nameservers simultaneously, being more aggressive about timeouts and using the secondary nameserver, or choosing whichever one has better response times.

While we believe that some of these approaches have some merit, we also note that they cannot address all of the failure modes that CoDNS can handle. In particular, we have often seen all nameservers at a site fail, [9] in which case CoDNS is still able to answer queries via the remote nameservers. Correlated failure of local nameservers renders these approaches useless, while correlated failure among groups of remote servers is less likely.

Overly aggressive requests are likely to backfire in the case of local nameservers, since we have seen that overload causes local nameserver failure. Increasing the request

[9] Most cases happen because of stale "/etc/resolv.conf", but we have seen both nameservers stay down for over 10 hours at some site. We have also noticed that some site firewall rule misconfigurations block all DNS traffic to their nameservers.

rate to a failing server is not likely to improve performance. Load balancing among local nameservers is more plausible, but still requires modifications to all clients and programs. Given the cost of changing the infrastructure, it is perhaps appealing to adopt a technique like CoDNS that covers a broader range of failures.

Finally, upgrade cost and effort are real issues we have heard from many system administrators – secondary nameservers tend to be machines that are a generation behind the primary nameservers, based on the expectation of lower load. Increasing the request rate to the secondary nameserver will require upgrading that machine, whereas CoDNS works with existing infrastructure.

4.6.3 TCP Queries

Another possible solution is to use TCP as a way of communicating with local nameservers. If the failure is caused by packet losses in the LAN or silent packet drops caused by UDP buffer overflow, TCP can improve the situation by reliable data delivery.

Although the DNS RFC [53] allows the use of TCP in addition to UDP, in practice, TCP is used only when handling AXFR queries for the zone transfer or when the requested record set is bigger than 512 bytes. The reason why TCP is not favored in name lookups is mainly because of the additional overhead. If a TCP connection is needed for every query, it would end up handling nine packets instead of two : three to establish the connection, two for the request/response, and four to tear down the connection. A persistent TCP connection might remove the per-query connection overhead, but it also needs to consume one or two extra network packets for ACKs. Also, there is another issue of reclaiming the idle connections, since they consume system resources and can degrade performance. The DNS RFC [53] specifies two minutes as a cutoff but in practice most servers disconnect the idle connection within 30 seconds.

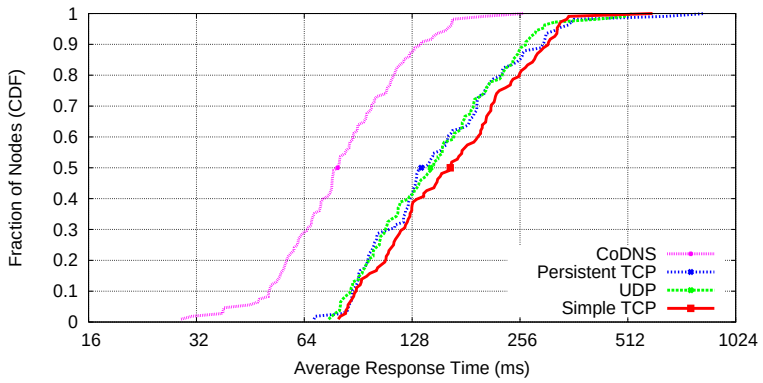
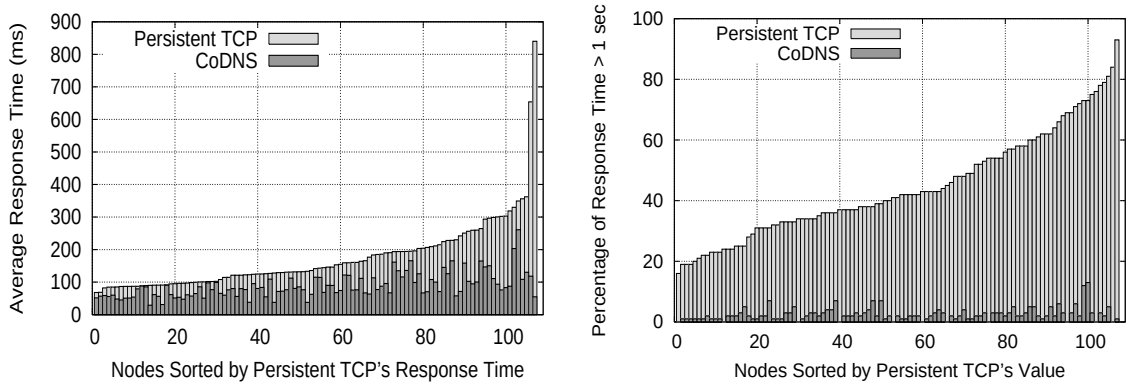


Figure 4.21: Comparison of UDP, TCP, and CoDNS latencies

To compare the performance between UDP and TCP, we replay 10,792 unique host names obtained from one day's live traffic of a CoDeeN proxy at 107 PlanetLab nodes. Carrying out a completely fair comparison is difficult, since we cannot issue the same query for all of them at the same time. Instead, to give a relatively fair comparison, we run the test for CoDNS first, and subsequently run other parts, making all but CoDNS get the benefit of cached responses from the local nameserver. Figure 4.21 shows the CDF of the average response time for all approaches. Persistent TCP and UDP have comparable performance, while simple TCP is noticeably worse. The CoDNS latencies, included for reference, are better than all three.

The replay scenario described above should be favorable to TCP, but even in this conservative configuration, CoDNS still wins. Figure 4.22(a) shows that all nodes report that CoDNS is 10% to 500% faster than TCP, confirming CoDNS is a more attractive option than TCP. The large difference is in the slow-response portion, where CoDNS wins the most and where TCP-based lookups cannot help. Figure 4.22(b) shows that a considerable amount of time is still spent on the slow queries in TCP-based lookups. CoDNS reduces this time by 16% to 92% when compared to the TCP-based measurement. Though TCP ensures that the client's request reaches the nameserver, if the nameserver is over-



(a) Average Response Time (b) Slow Response Time Portion

Figure 4.22: CoDNS vs. TCP

loaded, it may have trouble contacting the DNS hierarchy for cache misses.

4.7 Related Work

Traditional DNS-related research has mainly focused on the problems in the server-side DNS infrastructure. As a seminal study in DNS performance measurement, Danzig *et al.* found that a large number of network packets in the NSFNet were being wasted due to excessive DNS traffic, identifying nameserver software bugs, their flawed resiliency behavior, and misconfiguration by operators as major culprits [28].

Since then, bugs in the resolvers and nameservers have been reduced [50], but recent measurements show that there is still much room for improvement. In 2000, Wills *et al.* [87] and Huitema *et al.* [39] reported 29% of DNS lookups take over 2 seconds, and Cohen *et al.* [24] reported 10% of lookups exceed more than 3 seconds. Jung *et al.* also present data indicating 23% of all server-side lookups receive no results, indicating the problems of improper configurations and incorrect nameservers still persist [42]. They measure the client-side performance in terms of response time and caching hit ratio as

well. However, that work does not trace the origins of name lookup delays from the client-side, concentrating only on the wide-area DNS traffic. Given the fact that local nameserver cache hit ratios are 80% - 87% [42, 87], even a small problem in the local nameserver and its environment can skew the latency of a large number of lookups. Our study addresses this problem. Liston *et al.* indirectly provide some evidence of local nameserver problems by attributing the major sources of response time delay to edge nameservers rather than the root/gTLD servers [51].

The research community has recently renewed its focus on improving server-side infrastructure. Cox *et al.* investigate the possibility of transforming DNS into a peer-to-peer system [25] using a distributed hash table [79]. The idea is to replace the hierarchical DNS name resolving process with a flat peer-to-peer query style, in pursuit of load balancing and robustness. With this design, the configuration mistakes by administrators can be eliminated and the traffic bottleneck on the root servers is removed so that the load is distributed over the entities joining the system.

In CoDoNS, Ramasubramanian *et al.* improve the poor latency performance of this approach by using proactive replication of DNS records [65]. They exploit the Zipf-like distribution of the domain names in Web browsing [13] to reduce the replication overhead while providing O(1) proximity [64]. Our approaches differ in several important aspects – we attempt to reduce overlapping information in caches, in order to maximize the overall aggregate cache size, while they use replication to reduce latency. Our desire for a small process footprint stems from our observation that memory pressure is one of the causes of current failures in client-side infrastructure. While their system appears not to be deployed in production, they perform an evaluation using a DNS trace with a Zipf factor above 0.9 [65]. In comparison, our evaluation of CoDNS uses the live traffic generated by CoDeeN *after* its proxies have used their local DNS caches, so the request stream seen

by CoDNS has a Zipf factor of 0.50-0.55, which is a more difficult workload. In any case, since CoDNS does not depend on the specifics of the name lookup system, we expect that it can interoperate with CoDoNS if the latter provides better performance than the existing nameservers at PlanetLab sites. One issue that will have to be addressed by any proposed DNS replacement system is the use of intelligent nameservers that dynamically determine which IP address to return for a given name. These nameservers are used in CDNs and geographic load balancers, and can not be replaced with purely static lookups, such as those performed in CoDoNS. Since CoDNS does not replace existing DNS infrastructure, we can interoperate with these intelligent nameservers without any problem.

Kangasharju *et al.* pursue a similar approach to reducing the DNS lookup latency by more aggressively replicating DNS information [43]. Inspired by the fact the entire DNS record database fits into the size of a typical hard disk and with the recent emergence of terrestrial multicast and satellite broadcast systems, this scheme reduces the need to query distant nameservers by keeping the DNS information up to date by efficient world-wide replication. However, this approach intrinsically favors DNS records with long TTLs, and may not improve the lookup performance for the short TTL cases.

The difference in our approach is to temporarily use functioning nameservers of peer nodes, separate from the issue of improving the DNS infrastructure itself. We expect that benefits in improving the infrastructure “from above” will complement our “bottom up” approach.

Chapter 5

CoBlitz

Many new content distribution networks have recently been developed to focus on areas not generally associated with "traditional" Web (HTTP) CDNs. These systems often focus on distributing large files, especially in flash crowd situations where a news story or software release causes a spike in demand. These new approaches break away from the "whole-file" data transfer model, the common access pattern for Web content. Instead, clients download pieces of the file (called chunks, blocks, or objects) and exchange these pieces with each other to form the complete file. The most widely used system of this type is BitTorrent [23], while related research systems include Bullet [49], Shark [4], and FastReplica [18].

Using peer-to-peer systems makes sense when the window of interest in the content is short, or when the content provider cannot afford enough bandwidth or CDN hosting costs. However, in other scenarios, a managed CDN service may be an attractive option, especially for businesses that want to offload their bandwidth but want more predictable performance. The problem arises from the fact that HTTP CDNs have not traditionally handled this kind of traffic, and are not optimized for this workload. In an environ-

ment where objects average 10KB, and where whole-file access is dominant, suddenly introducing objects in the range of hundreds of megabytes may have undesirable consequences. For example, CDN nodes commonly cache popular objects in main memory to reduce disk access, so serving several large files at once could evict thousands of small objects, increasing their latency as they are reloaded from disk.

To address this problem, we have developed the CoBlitz large file transfer service, which runs on top of the CoDeeN content distribution network, an HTTP-based CDN. This combination provides several benefits: (a) using CoBlitz to serve large files is as simple as changing their URLs – no re-hosting, extra copies, or additional protocol support is required; (b) CoBlitz can operate with unmodified clients, servers, and tools like curl or wget, providing greater ease-of-use for users and for developers of other services; (c) obtaining maximum per-client performance does not require multiple clients to be downloading simultaneously; and (d) even after an initial burst of activity, the file stays cached in the CDN, providing latecomers with the cached copy.

From an operational standpoint, this approach of running a large-file transfer service on top of an HTTP content distribution network also has several benefits: (a) given an existing CDN, the changes to support scalable large-file transfer are small; (b) no dedicated resources need to be devoted for the large-file service, allowing it to be practical even if utilization is low or bursty; (c) the algorithmic changes to efficiently support large files also benefit smaller objects.

Over the 30 months that CoBlitz and its associated service, CoDeploy, have been running on PlanetLab, we have had the opportunity to observe its algorithms in practice, and to evolve its design, both to reflect its actual use, and to better handle real-world conditions. This utilitarian approach has given us a better understanding of the effects of scale, node peering policies, replication behavior, and congestion, giving us new insights

into how to improve performance and reliability. With these changes, CoBlitz is able to deliver in excess of 1 Gbps on PlanetLab, and to outperform a range of systems, including research systems as well as BitTorrent.

In this chapter, we discuss what we have learned in the process, and how the observations and feedback from long-term deployment have shaped our system. We discuss how our algorithms have evolved, both to improve performance and to cope with the scalability problems of our system. Some of these changes stem from observing the real behavior of the system versus the abstract underpinnings of our original algorithms, and others from observing how our system operates when pushed to its limits. We believe that our observations will be useful for three classes of researchers: (a) those who are considering deploying scalable large-file transfer services; (b) those trying to understand how to evaluate the performance of such systems; and (c) those who are trying to capture salient features of real-world behavior in order to improve the fidelity of simulators and emulators.

5.1 Background

In this section, we provide general information about HTTP CDNs, the problems caused by large files, and the history of CoBlitz and CoDeploy.

5.1.1 HTTP Content Distribution Networks

Content distribution networks relieve Web congestion by replicating content on geographically-distributed servers. To provide load balancing and to reduce the number of objects served by each node, they use partitioning schemes, such as consistent hashing [45], to assign objects to nodes. CDN nodes tend to be modified proxy servers that fetch files on demand

and cache them as needed. Partitioning reduces the number of nodes that need to fetch each object from the origin servers (or other CDN nodes), allowing the nodes to cache more objects in main memory, eliminating disk access latency and improving throughput.

In this environment, serving large files can cause several problems. Loading a large file from disk can temporarily evict several thousand small files from the in-memory cache, reducing the proxy's effectiveness. Popular large files can stay in the main memory for a longer period, making the effects more pronounced. To get a sense of the performance loss that can occur, one can examine results from the Proxy Cacheoffs [71], which show that the same proxies, when operating as "Web (server) accelerators," can handle 3-6 times the request rate than when operating in "forward mode," with much larger working sets. So, if a CDN node suddenly starts serving a data set that exceeds its physical memory, its performance will drop dramatically, and latency rises sharply. Bruce Maggs, Akamai's VP of Research, states:

> "Memory pressure is a concern for CDN developers, because for optimal latency, we want to ensure that the tens of thousands of popular objects served by each node stay in the main memory. Especially in environments where caches are deployed inside the ISP, any increase in latency caused by objects being fetched from disk would be a noticeable degradation. In these environments, whole-file caching of large files would be a concern [52]."

Akamai has a service called EdgeSuite Net Storage, where large files reside in specialized replicated storage, and are served to clients via overlay routing [2]. We believe that this service demonstrates that large files are a qualitatively different problem for CDNs.

5.1.2 Large-file Systems

As a result of these problems and other concerns, most systems to scalably serve large files departed from the use of HTTP-based CDNs. Two common design principles are evident in these systems: treat large files as a series of smaller chunks, and exchange chunks between clients, instead of always using the origin server. Operating on chunks allows finer-grained load balancing, and avoids the trade-offs associated with large-file handling in traditional CDNs. Fetching chunks from other peers not only reduces load on the origin, but also increases aggregate capacity as the number of clients increases.

We subdivide these systems based on their inter-client communication topology. We term those that rely on greedy selection or all-to-all communication as examples of the *swarm* approach, while those that use tree-like topologies are termed *stream* systems.

Swarm systems, such as BitTorrent [23] and FastReplica [18], preceded stream systems, and scaled despite relatively simple topologies. BitTorrent originally used a per-file centralized directory, called a tracker, that lists clients that are downloading or have recently downloaded the file. Clients use this directory to greedily find peers that can provide them with chunks. The newest BitTorrent can operate with tracker information shared by clients. In FastReplica, all clients are known at the start, and each client downloads a unique chunk from the origin. The clients then communicate in an all-to-all fashion to exchange chunks. These systems reduce link stress, which is persistent traffic load on the link, compared to direct downloads from the origin, but some chunks may traverse shared links repeatedly if multiple clients download them.

The stream systems, such as ESM [19], SplitStream [15], Bullet [49], Astrolabe [84], and FatNemo [10] address the issues of load balancing and link stress by optimizing the peer-selection process. Thoes systems generate a tree-like topology (sometimes a mesh or gossip-based network inside the tree), which tends to stay relatively stable during the

download process. The effort in tree-building can produce higher aggregate bandwidths, suitable for transmitting the content simultaneously to a large number of receivers. The trade-off, however, is that the higher link utilization is possible only with greater synchrony. If receivers are only loosely synchronized and chunks are transmitted repeatedly on some links, the transmission rate of any subtrees using those nodes also decreases. As a result, these systems are best suited for synchronous activity of a specified duration.

5.1.3 CoBlitz, CoDeploy, and CoDeeN

This section discusses our experience running two large-file distribution systems, CoBlitz and CoDeploy, which operate on top of the CoDeeN content distribution network. As discussed in Chapter 3, CoDeeN is a HTTP CDN that runs on every available PlanetLab node, with access restrictions in place to prevent abuse and to comply with hosting site policies. To use CoDeeN, clients configure their browsers to use a CoDeeN node as a proxy, and all of their Web traffic is then handled by CoDeeN. Note that this behavior is only part of CoDeeN as a policy decision – CoBlitz does not require changing any browser setting.

Both CoBlitz and CoDeploy use the same large-file transfer service core, which we call CoBlitz in the rest of this chapter for simplicity. The main difference between the two is the access mechanism – CoDeploy requires the client to be a PlanetLab machine, while CoBlitz is publicly accessible. CoDeploy was launched first, and allows PlanetLab researchers to use a local instance of CoDeeN to fetch experiment files. CoBlitz allows the public to access CoDeploy by providing a simpler URL-based interface. To use CoBlitz, clients prepend the original URL with `http://coblitz.codeen.org:3125/` and fetch it like any other URL. A customized DNS server maps the name coblitz.codeen.org to a nearby PlanetLab node.

In 30 months of operation, the system has undergone three sets of changes: scaling from just North American PlanetLab nodes to all of PlanetLab, changing the algorithms to reduce load at the origin server, and changing the algorithms to reduce overall congestion and increase performance. We believe the incremental development model in Section 1.4 has been absolutely critical to our success in improving CoBlitz, as we describe later in this chapter.

5.2 CoBlitz Design

Before discussing CoBlitz's optimizations, we first describe how we have made HTTP CDNs amenable to handling large files. Our approach has two components: modifying large file handling to efficiently support them on HTTP CDNs, and modifying the request routing for these CDNs to enable more swarm-like behavior under heavy load. Though we build on the CoDeeN CDN, we do not believe any of these changes are CoDeeN-specific – they could equally be applied to other CDNs. Starting from an HTTP CDN maintains compatibility with standard Web clients and servers, whereas starting with a stream-oriented CDN might require more effort to efficiently support regular Web traffic.

5.2.1 Requirements

We treat large files as a set of small files that can be spread across the CDN. To make this approach as transparent as possible to clients and servers, the dynamic fragmentation and reassembly of these small files is performed inside the CDN, on demand. Each CDN node has an agent that accepts clients' requests for large files and converts them into a series of requests for pieces of the file. Pieces are specified using HTTP/1.1 byte ranges [30], which the Apache Web server has supported since August 1996 (version 1.2), and which

original request

```
GET /file.iso
Host: www.example.com
```


resulting series of requests

```
GET /file.iso,start=0,end=9999
Host: www.example.com
X-Bigfile: 1

GET /file.iso,start=10000,end=19999
Host: www.example.com
X-Bigfile: 1
...
```

Figure 5.1: The client-facing agent converts a single request for a large file into a series of requests for smaller files. The new URLS are only a CDN-internal representation – neither the client nor the origin server sees them.

appeared in other servers in the same timeframe. After these requests are injected into the CDN, the results are reassembled by the agent and passed to the client. For simplicity, this agent occupies a different port number than regular CoDeeN requests. The process has some complications, mostly related to the design of traditional CDNs, limitations of HTTP, and the limitations of standard HTTP proxies (which are used as the CDN nodes). Some of these problems include:

Chunk naming – If chunks are named using the original URL, all of a file's chunks will share the same name, and will be routed similarly since CDNs hash URLs for routing [44, 85]. Since we want to spread chunks across the CDN, we must use a different chunk naming scheme than the whole-file naming scheme based on its URL.

Range caching – We know of no HTTP proxies that cache arbitrary ranges of Web objects, though some can serve ranges from cached objects, and even recreate a full object from all of its chunks. Since browsers are not likely to ask for arbitrary and disjoint pieces of an object, no proxies have developed the necessary support. Since we want to cache

at the chunk level instead of the file level, we must address this limitation and should support range caching.

Congestion – During periods of bursty demand and heavy request synchrony, consistent hashing may produce roving instantaneous congestion. If many clients at different locations suddenly ask for the same file, a lightly-loaded CDN node may see a burst of requests. If the clients all ask for another file as soon as the first download completes, another CDN node may become instantly congested. This bursty congestion prevents using the aggregate CDN bandwidth effectively over small time scales.

Our approach is to address these problems as a whole, which is discussed in the next section, and to avoid new problems from piecemeal fixes. One example of piecemeal fixes is following. Adding range caching to the Squid proxy has been discussed since 1998 [70], but would expand the in-memory metadata structures, increasing memory pressure, and would require changing the Internet cache protocol (ICP) used by caches to query each other. Even if we added this support to CoDeeN's proxies, it would still require extra support in the CDN, since the range information would have to be hashed along with the URL.

5.2.2 Chunk Handling Mechanics

We modify intra-CDN chunk handling and request redirection by treating each chunk as a real file with its own name, so the bulk of the CDN does not need to be modified. This name contains the start and end ranges of the file, so different chunks will have different hash values. Only the CDN ingress/egress points are affected, at the boundaries with the client and the origin server.

The agent takes the client's request, converts it into a series of requests for chunks, reassembles the responses, and sends it to the client. The client is not aware that the

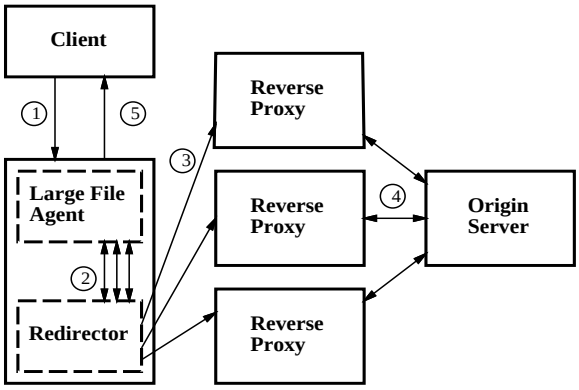


Figure 5.2: Large-file processing – 1. the client sends the agent a request, 2. the agent generates a series of URL-mangled chunk requests, 3. those requests are spread across the CDN, 4. assuming cache misses, the URLs are de-mangled on egress, and the responses are modified, 5. the agent collects the responses, reassembles if needed, and streams it to the client

request is handled in pieces, and no browser modifications are needed. This process is implemented in a small program on each CDN node, so communication between the program and the CDN infrastructure is cheap. The requests sent into the CDN, shown in Figure 5.1, contain extended filenames that specify the actual file and the desired byte range, as well as a special header so that the CDN modifies these requests on the egress of the CDN network. Otherwise, these requests look like ordinary requests with slightly longer filenames. The full set of steps is shown in Figure 5.2, where each solid rectangle is a separate machine connected via the Internet.

All byte-range interactions take place between the proxy and the origin server – on egress, the request's name is reverted, and range headers are added. The server's response is changed from an HTTP 206 code (partial content received) to 200 (full file received). The underlying proxy never sees the byte-range transformations, so no range-caching support is required. Figure 5.3 shows this process with additional temporary headers. These headers contain the file length, allowing the agent to provide the content length for

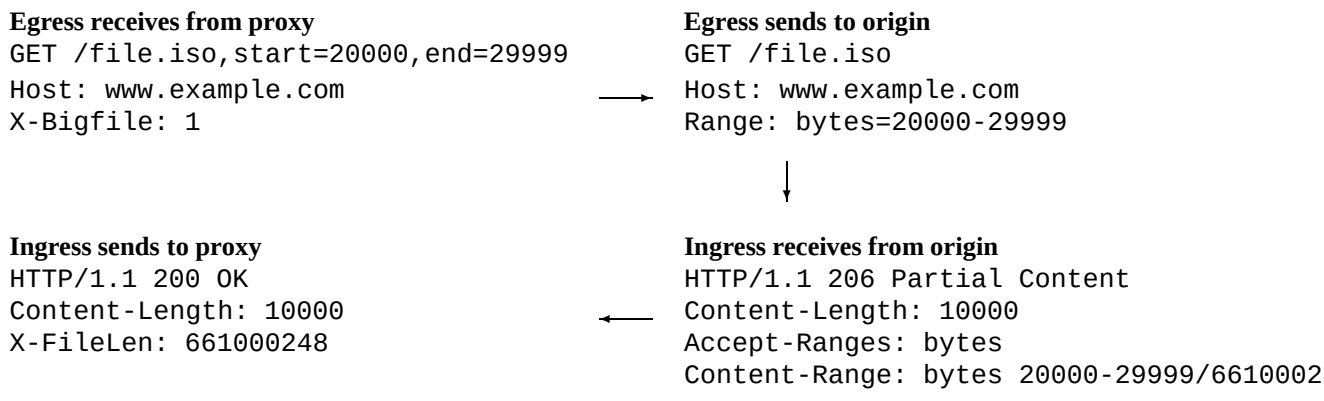


Figure 5.3: Egress and ingress transformations when the CDN communicates with the origin server. The CDN internally believes it is requesting a small file, and the egress transformation requests a byte-range of a large file. The ingress converts the server's response to a response for a complete small file, rather than a piece of a large file.

the complete download.

Having the agent use the local proxy avoids having to re-implement the CDN code (such as node liveness, or connection management) in the agent, but can cause cache pollution if the proxy caches all of the agent's requests. The ingress adds a cache-control header that disallows local caching, which is removed on egress when the proxy routes the request to the next CDN node. As a result, chunks are cached at the next-hop CDN nodes instead of the local node.

Since the CDN sees a large number of small file requests, it can use its normal routing, replication, and caching policies. These cached pieces can then be used to serve future requests. If a node experiences cache pressure, it can evict as many pieces as needed, instead of evicting one large file. Similarly, the arrival/departure of nodes will only cause missing pieces to be re-fetched, instead of the whole file. The only external difference is that the server sees byte-range requests from many proxies instead of one large file request from one proxy.

5.2.3 Agent Design

The agent is the most complicated part of CoBlitz, since it must operate smoothly, even in the face of unpredictable CDN nodes and origin servers outside of our control. The agent monitors the chunk downloads for correctness and for performance. The correctness checking ensures that the server is capable of serving HTTP byte-range requests, verifying that the response is cacheable, and comparing modification-related headers (file length, last-modified time, etc.) to detect if a file has changed at the origin during its download. In the event of problems, the agent can abort the download and return an error message to the client. The agent is the largest part of CoBlitz – it consists of 770 lines of code with statement terminators (1975 lines total), versus 60-70 lines of changes for ingress/egress modifications.

To determine when to re-issue chunk fetches, the agent maintains overall and per-chunk statistics during the download. Several factors may slow chunk fetching, including congestion between the proxy and its peers, operational problems at the peers, and congestion between the peers and the origin. After downloading the first chunk, the agent has the header containing the overall file size, and knows the total number of chunks to download. It issues parallel requests up to its limit, and uses non-blocking operations to read data from the sockets as it becomes available.

Using an approach inspired by LoCI [7], slow transfers are addressed by issuing multiple requests – whenever a chunk exceeds its download deadline, the agent opens a new connection and re-issues the chunk request. The most recent request for the same chunk is allowed to continue downloading, and any earlier requests for the chunk are terminated. In this way, each chunk can have at most two requests for it in flight from the agent, a departure from LoCI where even more connections are made as the deadline approaches. The agent modifies a non-critical field of the URL in retry requests beyond the first retried

request for each chunk. This field is stripped from the URL on egress, and exists solely to allow the agent to randomize the peer serving the chunk. In this way, the agent can exert some control over which peer serves the request, to reduce the chance of multiple failures within the CDN. Keeping the same URL on the first retry attempts to reduce cache pollution – in a load-balanced, replicated CDN, the retry is unlikely to be assigned to the same peer that is handling the original request.

The first retry timeout for each chunk is set using a combination of the standard deviation and exponentially-weighted moving average of recent chunk downloading times. Subsequent retries use exponential backoff to adjust the deadline, up to a limit of 10 backoffs per chunk. In our testing, most chunks are either delivered within 3 retries or not delivered at all, so 10 backoffs would be conservative. To bound the backoff time, we also have a hard limit of 10 seconds for the chunk timeout, which is translated into 48 Kbps with 60 KB chunks. This rate is lower than the modem speed, and probably indicates some sort of failure when the limit timeout expires. The initial timeout is set to 3 seconds for the first chunk – while most nodes finish faster, using a generous starting point avoids overloading slow origin servers. In practice, 10-20% of chunks are retried, but the original fetch usually completes before the retry. We could reduce retry aggressiveness, but the current approach is unlikely to cause much extra traffic to the origin since the first retry uses a different replica with the same URL.

By default, the agent sends completed chunks to the client as soon as they finish downloading, as long as all preceding chunks have also been sent. If the chunk at the head of the line has not completed downloading, no new data are sent to the client until the chunk completes. By using enough parallel chunk fetches, delays in downloading chunks can generally be overlapped with others in the pipeline. If clients that can use chunked transfer encoding provide a header in the request indicating they are capable of

handling chunks in any order, the agent sends chunks as they complete, with no head-of-line blocking. Chunk position information is returned in a trailer following each chunk, which the client software can use to assemble the file in the correct order.

The choice of chunk size is a trade-off between efficiency and latency – small chunks will result in faster chunk downloads, so slower clients will have less impact. However, the small chunks require more processing at all stages – the agent, the CDN infrastructure, and possibly the origin server. Larger chunks, while more efficient, can also cause more delay if head-of-line blocking arises. After some testing [58], we chose a chunk size of 60KB, which is large enough to be efficient, but small enough to be manageable. In particular, this chunk size can easily fit into Linux's default outbound kernel socket buffers, allowing the entire chunk to be written to the socket with a single system call that returns without blocking.

5.2.4 Peering Strategy

The goal of CoBlitz is to improve the overall downloading throughput rather than reducing the per-chunk response latency. This is somewhat different from the goal of a regular CDN like CoDeeN, which tries to provide an interactive Web environment to the end users. Most monitoring and peering mechanism from CoDeeN is reused in CoBlitz, but the round-trip time cutoff for peer selection is relaxed with CoBlitz to support more peers to distribute the load, and request re-forwarding is introduced to improve the cache hit rate even if it increases the chunk latency. This change also reduces the origin server load and ultimately reduces the total download time for the client. More details can be found in Section 5.3 and Section 5.4.

5.2.5 Design Benefits

We believe that this design has several important features that not only make it practical for deployment now, but will continue to make it useful in the future:

No client synchronization – Since chunks are cached in the CDN when first downloaded, no client synchronization is needed to reduce traffic load on the origin server. If clients are highly synchronized, agents can use the same chunk to serve many client requests, reducing the number of intra-CDN transfers, but synchronization is not required for efficient operation.

Trading bandwidth for disk seeks – Fetching most chunks from other CDN nodes trades network bandwidth for disk seeks. Given the rate of improvement of each, this trade-off will continue to hold for the foreseeable future. Bandwidth is continually dropping in price, and disk seek times are not scaling.

Increasing chunk utility – Having all nodes store chunks makes them available to a larger population than storing the entire file at a small number of nodes. Many more nodes can now serve large files, so the total capacity is the sum of the bandwidths they have to serve clients, and the aggregate intra-CDN capacity is available to exchange chunks.

Using cheaper bandwidth – When CDN nodes communicate with each other, this bandwidth consumption is either within a LAN cluster hosting the CDN nodes, or through the network core, away from the clients that sit at the edge of the network. Core bandwidth has been improving in price/performance more rapidly than edge bandwidth, and LAN bandwidth is virtually free, so this consumption is in the more desirable direction.

Scaling with CDN size – As CDN size increases, and aggregate physical memory increases, chunks can be replicated more widely. The net result is that desired chunks are more likely to be in nearby nodes, so link stress drops as the CDN grows.

Tunable memory consumption – Varying the number of parallel chunks downloads that are used for each client controls the memory consumption of this approach. Slower clients can be allocated fewer parallel chunks, and the aggregate number of chunks can be reduced if a node is experiencing heavy load.

In-order or out-of-order delivery – For typical Web browsers [1] or other HTTP client software [2], chunks are delivered in order so that the download appears exactly like a non-CoBlitz download from the origin, and performance-hungry clients can use software that supports chunked encoding. Service differenciation is done by allocating a different port for each or by examining the "User-Agent" HTTP header.

5.3 Coping With Scale

One first challenge for CoBlitz was handling scale – at the time of CoBlitz's original deployment, CoDeeN was running on all 100 academic PlanetLab nodes in North America. The first major scale issue was roughly quadrupling the node count, to include every PlanetLab node. In the process, we adopted three design decisions that have served us well: (a) make peering a unilateral, asynchronous decision, (b) use minimum application-level ping times when determining suitable peers, and (c) apply hysteresis to the peer set. Unilateral, asynchronous peering has already been discussed in Section 2.2, and others are described in the remainder of this section.

[1] Such as Internet Explorer, Firefox, Opera and so on.
[2] Such as curl and wget.

5.3.1 Peer Set Selection

As described in Section 3.2, CoDeeN uses application-level pings, rather than network pings, to determine round trip times (RTTs). Originally, CoDeeN kept the average of the four most recent RTT values, and selected the 60 [3] closest peers within a 100ms RTT cutoff. The 100ms cutoff was chosen to reduce noticeable lag in interactive settings, such as Web browsing. In parts of the world where nodes could not find 20 peers within 100ms, this cutoff is raised to 200ms and the 20 best peers are selected.

This approach exhibited two problems – a high rate of change in the peer sets, and low overlap among peer sets for nearby peers. The high change rate potentially impacts chunk caching in CoBlitz – if the peer that previously fetched a chunk is no longer in the peer set, the new peer that replaces it may not yet have fetched the chunk. To address this issue, hysteresis was added to the peer set selection process. Any peer not on the set could only replace a peer on the set if it was closer in two-thirds of the last 32 heartbeats. Even under the worst-case conditions, using the two-thirds threshold would keep a peer on the set for 20 minutes at a time. While hysteresis reduced peer set churn, it also reinforced the low overlap between neighboring peer sets. Further investigation indicated that CoDeeN's application-level heartbeats had an order of magnitude more variance than network pings. This variance led to instability in the average RTT calculations, so once nodes were added to the peer set, they rarely got displaced.

Switching from an *average* application-level RTT to the *minimum* observed RTT (an approach also used in other systems [12, 27, 67]) and increasing the number of samples yielded significant improvement, with application-level RTTs correlating well with ping time on all functioning nodes. Misbehaving nodes still showed large application-level minimum RTTs, despite having low ping times. The overlap of peer lists for nodes at

[3] It is now increased to 120 nodes.

the same site increased from roughly half to almost 90%. At the same time, we discovered that many intra-PlanetLab paths had very low latency, and restricting the peer size to 60 was needlessly constrained. We increased this limit to 120 nodes, and issued 2 heartbeats per second. Of the nodes regularly running CoDeeN, two-thirds tend to now have 100+ peers. More details of the redesign process and its corresponding performance improvement can be found in our previous study [11].

5.3.2 Scaling Larger

It is interesting to consider whether this approach could scale to a much larger system, such as a commercial CDN like Akamai. By the numbers, Akamai is about 40 times as large as our deployment, at 15,000 servers across 1,100 networks. However, part of what makes scaling to this size simpler is deploying clusters at each network point-of-presence (POP), which number only 2,500. Further, their servers have the ability to issue reverse ARPs and assume the IP addresses of failing nodes in the cluster, something not permitted on PlanetLab. With this ability, the algorithms need only scale to the number of POPs, since the health of a POP can be used instead of querying the status of each server. Finally, by imposing geographic hierarchy and ISP-level restrictions, the problem size is further reduced. With these assumptions, we believe that we can scale to larger sizes without significant problems.

5.4 Reducing Load & Congestion

Reducing origin server load and reducing CDN-wide congestion are related, so we present them together in this section. Origin load is an important metric for CoBlitz, because it determines CoBlitz's caching benefit and impacts the system's overall performance.

From a content provider's standpoint, CoBlitz would ideally fetch only a single copy of the content, no matter what the demand is. However, for reasons described below, this goal may not be practical.

5.4.1 Increasing Peer Set Size

Increasing the peer set size, as described in Section 5.3.1 has two effects – each node appears as a peer of many more nodes than before, and the number of nodes chosen to serve a particular URL is reduced. In the extreme, if all CDN nodes were in each others' peer sets, then the total number of nodes handling any URL would equal *NumCandidates*. [4] In practice, the peer sets give rise to overlapping regions, so the number of nodes serving a particular URL is tied to the product of the number of regions and *NumCandidates*.

When examining origin server load in CoBlitz, we found that nodes with fewer than five peers generate almost one-third of the traffic. Some poorly-connected sites have such high latency that even with an expanded RTT criterion, they find few peers. At the same time, few sites use them as peers, leading to them being an isolated cluster. For regular Web CDN traffic, these small clusters are not much of an issue because small files do not tax on lots of resources per origin server, but for large-file traffic, the extra load these clusters cause on the origin server slows the rest of the CDN significantly. Increasing the minimum number of peers per node to 60 reduces traffic to the origin. Because of unilateral peering, this change does not harm nearby nodes – other nodes still avoid these poorly-connected nodes.

Reducing the number of replicas per URL reduces origin server load, since fewer nodes fetch copies from the origin, but it also causes more bursty traffic at those replicas if downloading is synchronized. For CoBlitz, synchronized downloads occur when de-

[4] See the CoDeeN's request redirection algorithm in Figure 3.2.

velopers push software updates to all nodes, or when cron-initiated tasks simultaneously fetch the same file. In these cases, demand at any node experiences high burstiness over small time scales, which leads to congestion in the CDN.

5.4.2 Fixing Peer Set Differences

Once other problems are addressed, differences in peer sets can also cause a substantial load on the origin server. To understand how this arises, imagine a CDN consisting of n-node clusters, where each node in the cluster does not see one peer and instead picks an outsider at random. If we ask all nodes for the top candidate in the HRW list for a given URL, at least one node is likely to return the candidate that is a better choice than the node that the majority would return. [5]

In general, if each node is missing l peers at random, and we ask for top m ($m > 0$) candidates, then the probability of a node selecting k ($k \leq m$) outside nodes among the top m candidates is

$$P(X = k) = \frac{\binom{l}{k} \cdot \frac{m!}{(m-k)!} \cdot \frac{(n-m)!}{(n-m-l+k)!}}{\frac{n!}{(n-l)!}}$$

Thus, the expected number of outside nodes placed in the top m candidates for each node is $E(X) = \sum_{k=1}^{m} k \cdot P(X = k)$. When $l = 2$ and $m = 5$, then $E(X)$ becomes about 0.16, and the expected number of outsiders that get picked in the union of five top replicas is $60 \cdot 0.16 = 9.6$. [6] Making the matter worse is that these "extra" nodes fetching from the origin also provide very low utility to the rest of the nodes – since few nodes are using them to fetch the chunk, they do not reduce the traffic at the other replicas.

[5] The probability of each node picking the outside node as the top candidate is $1/n$, so the expected number of outsiders is $n \cdot 1/n = 1$. For simplicity, we assume each node sees a different outside node.

[6] The actual number in practice will be smaller than 9.6, since not all l outside nodes that each original node sees are different. The value shown here is a union bound.

To fix this problem, we observe that when a node receives a forwarded request, it can independently check to see whether it should be the node responsible for serving that request. On every forwarded request that is not satisfied from the cache, the receiving node performs its own HRW calculation. If it finds itself as one of the top candidates, it considers the forwarded request reasonable and fetches the chunk from the origin server. If the receiver finds that it is not one of the top candidates, it forwards the request again. We find that 3-7% of chunks get re-forwarded this way in CoBlitz, but it can get as high as 10-15% in some cases. When all PlanetLab nodes act as clients, this technique cuts origin server load almost in half.

Due to the deterministic order of HRW, [7] this approach is guaranteed to make forward progress and be loop-free. While the worst case is a number of hops linear in the number of peer groups, this case is also exponentially unlikely. Even so, we limit this approach to only one additional hop in the redirection, to avoid forwarding requests across the world and to limit any damage caused by bugs in the forwarding logic. Given the relatively low rate of chunks forwarded in this manner, restricting it to only one additional hop appears sufficient.

5.4.3 Reducing Burstiness

While request re-forwarding dramatically reduces cache misses, it creates temporary burstiness when synchronized requests are issued. To illustrate the burstiness resulting from improved peering, consider a fully-connected clique of 120 CDN nodes that begin fetching a large file simultaneously. If all have the same peer set, then each node in the replica set k will receive $120/k$ requests, each for a 60 KB chunk. Assuming 2 replicas, the traffic demand on each is 28.8 Mbits. Assuming a 10 Mbps link, it will be fully uti-

[7] See Section 2.1.

lized for 3 seconds just for this chunk, and then the utilization will drop until the next burst of chunks.

The simplest way of reducing the small time-scale node congestion is to increase the number of replicas for each chunk, but this would increase the number of fetches to the origin. Instead, we can improve on the purely mesh-based topology from the stream-oriented systems, which are excellent for reducing link stress. These systems all build communication trees, which eliminates the need to have the same data traverse a link multiple times. While trees are an unattractive option for standard Web CDNs because they add extra latency to every request fetched from the origin, a hybrid scheme can help the large-file case, if the extra hops can reduce congestion.

We take the re-forwarding support to forward chunks to better replicas, and use it to create deeper routing trees in the peer sets. We change the re-forwarding logic to use a different number of replicas when calculating the HRW set, leading to a broad replica set and a smaller set of nodes that fetch from the origin. We set the *NumCandidates* value to 1 when evaluating the re-forwarding logic, while dynamically selecting the value at the first proxy. The larger replica set at the first hop reduces the burstiness at any node without increasing origin load.

To dynamically select the number of replicas, we observe that we can eliminate burstiness by spreading the requests equally across the peers at all times. With a target per-client memory consumption, we can determine how many chunks are issued in parallel. So, the replication factor is governed by the following equation:

$$replicationfactor = \frac{peersize * chunksize}{memoryconsumption} \tag{5.1}$$

At 1 MB of buffer space per client, a 60KB chunk size, and 120 peers, our replication factor will be 7. We can, of course, cap the number of peers at some reasonable fraction

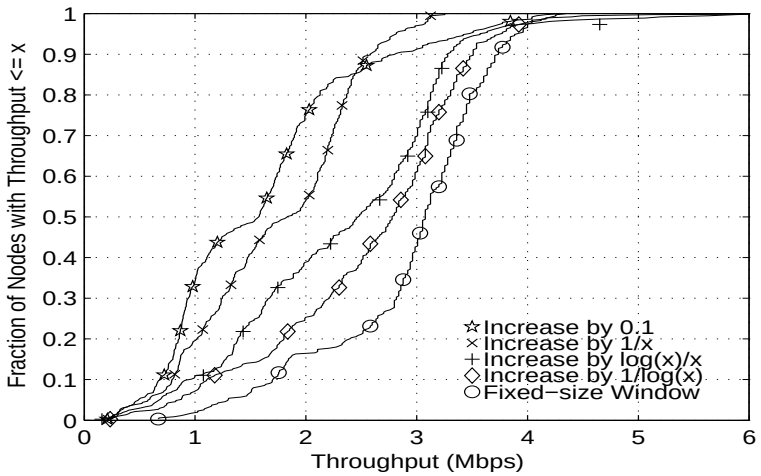


Figure 5.4: Throughput distribution for various window adjusting functions - Test scheme is described in section 5.5

of the maximum number of peers so that memory pressure does not cause runaway replication for the sake of load balancing. In practice, we limit the replication factor to 20% of the minimum target peer set, which yields a maximum factor of 12.

5.4.4 Dynamic Window Scaling

Although parallel chunk downloads can exploit multi-path bandwidth and reduce the effect of slow transfers, using a fixed number of parallel chunks also has some congestion-related drawbacks which we address. When the content is not cached, the origin server may receive more simultaneous requests than it can handle if each client is using a large number of parallel chunks. For example, the Apache Web Server is configured by default to allow 150 simultaneous connection, and some sites may not have changed this value. If a CDN node has limited bandwidth to the rest of the CDN, too many parallel fetches can cause self-congestion, possibly underutilizing bandwidth, and slowing down the time of all fetches. The problem in this scenario is that too many slow chunks will cause more retries than needed.

In either of these scenarios, using a smaller number of simultaneous fetches would be beneficial, since the per-chunk download time would improve. We view finding the "right" number of parallel chunks as a congestion issue, and address it in a manner similar to how TCP performs congestion control. Note that changing the number of parallel chunks is not an attempt to perform low-level TCP congestion control – since the fetches are themselves using TCP, we have this benefit already. Moreover, since the underlying TCP transport is already using additive-increase multiplicative-decrease, we can choose whatever scheme we desire on top of it.

Drawing on the ideas in TCP Vegas [12], we use the extra information we have in the CoBlitz agent to make the chunk "congestion window" a little more stable than a simple saw tooth. We use three criteria: (1) if the chunk finishes in less than the average chunk download time, increase the chunk window, (2) if the first fetch attempt is killed by retries, shrink the window, and (3) otherwise, leave the window size unmodified. We also decide that if more chunk fetches are in progress than the window size dictates, existing fetches are allowed to continue, but no new fetches (including retries) are allowed. Given that our condition for increasing the window is already conservative, we give ourselves some flexibility on exactly how much to add. Similarly, given that the reason for requiring a retry might be that any peer is slow, we decide against using multiplicative decrease when a chunk misses the deadline.

While determining the decrease rate is fairly easy, choosing a reasonable increase rate required some experimentation. The decrease rate was chosen to be one full chunk for each failed chunk, which would have the effect of closing the congestion window very quickly if all of the chunks outstanding were to retry. This logic is less severe than multiplicative decrease if only a small number of chunks miss their deadlines, but can shrink the window to a single chunk within one "RTT" (in this case, average chunk

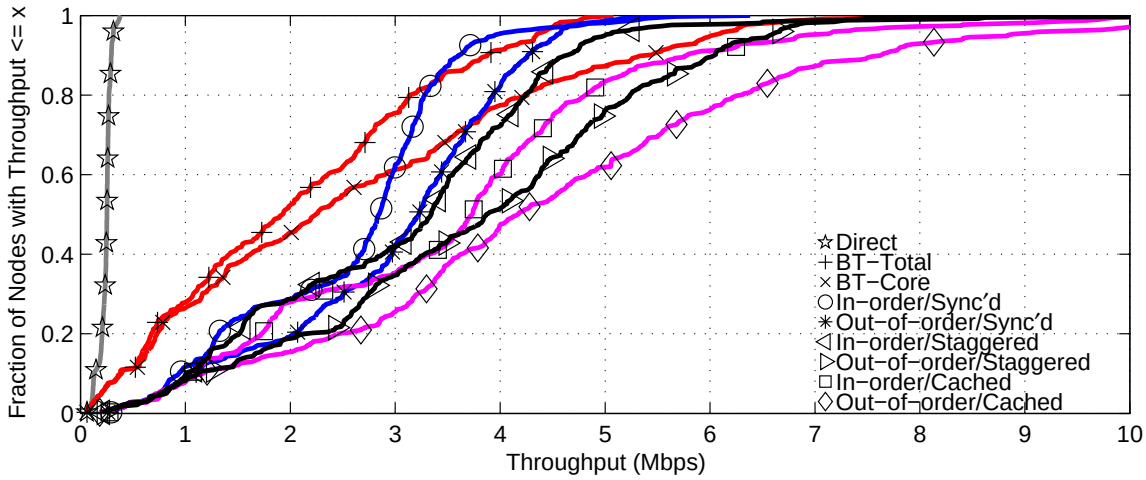


Figure 5.5: Achieved throughput distribution for all live PlanetLab nodes

download time) in the case of many failures.

Some experimentation with different increase rates is shown in Figure 5.4. The purely additive condition, 1/x on each fast chunk (where x is the current number of chunks allowed), fares poorly. Even worse is adding one-tenth of a chunk per fast chunk, which would be a slow multiplicative increase. The more promising approaches, adding log(x)/x and 1/log(x) (where we use 1 when x = 1) produce much better results. The 1/x case is not surprising, since it will always be no more than additive, since the window grows only when performing well. In TCP, the "slow start" phase would open the window exponentially faster, so we choose to use 1/log(x) to achieve a similar effect – it grows relatively quickly at first, and more slowly with larger windows. The chunk congestion window is maintained as a floating-point value, which has a lower bound of 1 chunk, and an upper bound as dictated by the buffer size available, which is normally 60 chunks. The final line in the graph, showing a fixed-size window of 60 chunks, appears to produce better performance, but comes at the cost of a higher node failure rate – 2.5 times as many nodes fail to complete with the fixed window size versus the dynamic sizing.

5.5 Evaluation

In this section, we evaluate the performance of CoBlitz, both in various scenarios, and in comparison with BitTorrent [23]. We use BitTorrent because of its wide use in large-file transfer [14], and because other research systems, such as Slurpie [77], Bullet' [48] (improved version of Bullet [49]) and Shark [4], are not running (or in some cases, available) at the time of this writing. As many of these have been evaluated on PlanetLab, we draw some performance and behavior comparisons in Section 5.6.

One unique aspect of our testing is the scale – we use every running PlanetLab node except those at Princeton, those designated as alpha testing nodes, and those behind firewalls that prevent CoDeeN traffic. The reason for excluding the Princeton nodes is because we place our origin server at Princeton, so the local PlanetLab nodes would exhibit unrealistically large throughputs and skew the means. During our testing in September and early October 2005, the number of available nodes that met the criteria above ranged from 360-380 at any given time, with a union size of 400 nodes.

Our test environment consists of a server with an AMD Sempron processor running at 1.5 GHz, with Linux 2.6.9 as its operating system and lighttpd 1.4.4 [46] as our web server. Our testing consists of downloading a 50MB file in various scenarios. The choice of this file size was to facilitate comparisons with other work [4, 48], which uses file sizes of 40-50MB in their testing. Our testing using a 630MB ISO image for the Fedora Core 4 download yielded slightly higher performance, but would complicate comparisons with other systems. Given that some PlanetLab nodes are in parts of the world with limited bandwidth, our use of 50MB files reduces contention problems for them. Each test is run three times, [8] and the reported numbers are the average value across the tests for which

[8] We have run this three-time experiments many times over a month. The trend in the results is similar though the actual values slightly vary depending on the traffi c level on PlanetLab at the time of testing.

the node was available. Due to the dynamics of PlanetLab, over any long period of time, the set of available nodes will change, and given the span of our testing, this churn is unavoidable.

We tune BitTorrent for performance – the clients and the server are configured to seed the peers indefinitely, and the maximum number of peers is increased to 60. While live BitTorrent usage will have lower performance due to fewer peers and peers departing after downloading, we want to provide similar environment for BitTorrent as CoBlitz. Note that CoBlitz's maximum chunk window size is 60, and its proxies hold the cached content long enough for the test downloads to finish.

We test a number of scenarios, as follows:

Direct – all clients fetch from the origin in a single download, which would be typical of standard browsers. For performance, we increase the socket buffer sizes from the system defaults to cover the bandwidth-delay product.

BitTorrent Total – this is a wall-clock timing of BitTorrent, which reflects the user's viewpoint. Even when all BitTorrent clients are started simultaneously, downloads begin at different times since clients spend different amounts of time contacting the tracker and finding peers.

BitTorrent Core – this is the BitTorrent performance from the start of the actual downloading at each client. In general, this value is 25-33% higher than the BitTorrent Total time, but is sometimes as much as 4 times larger.

In-order CoBlitz with Synchronization – Clients use CoBlitz to fetch a file for the first time and the chunks are delivered in order. All clients start at the same time.

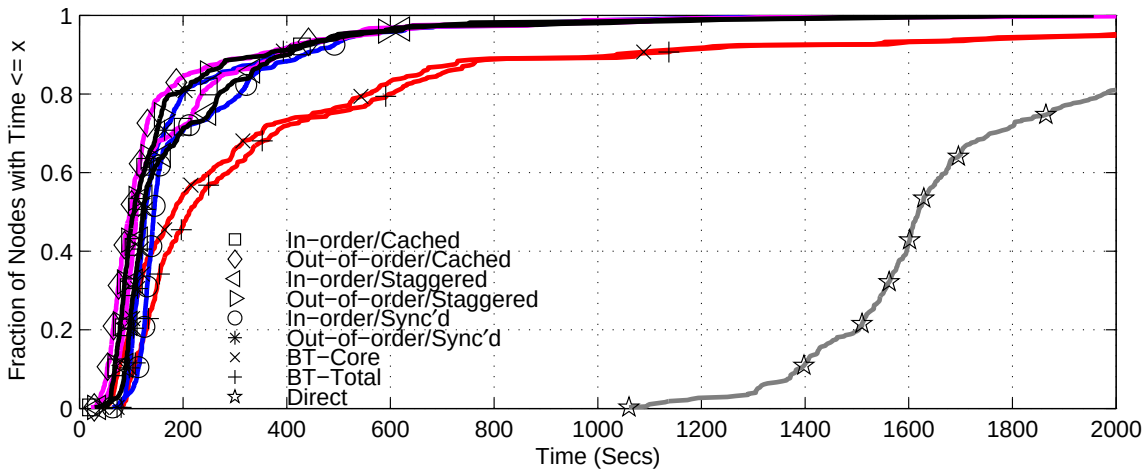


Figure 5.6: Download times across all live PlanetLab nodes

In-order CoBlitz with Staggering – We stagger the start of each client by the same amount of time that BitTorrent uses before it starts downloading. These stagger times are typically 20 to 40 seconds, with a few outliers as high as 150-230 seconds.

In-order CoBlitz with Contents Cached – Clients ask for a file that has already been fetched previously, and whose chunks are cached at the reverse proxies. All clients begin at the same time.

Out-of-order tests – Out-of-order CoBlitz with Synchronization, Out-of-order CoBlitz with Staggering, and Out-of-order CoBlitz with Contents Cached are the same as their in-order counterparts described above, but the chunks are delivered to the clients out of order.

5.5.1 Overall Performance

The throughputs and download times for all tests are shown in Figure 5.5 and Figure 5.6, with summaries presented in Table 5.1. For clarity, we trim the x axes of both graphs,

Strategy	Nodes		Throughput			Download Time		
	Good	Failed	Mean	50%	90%	Mean	50%	90%
Direct	372	17-18	0.23	0.20	0.38	1866.8	1618.2	3108.7
BitTorrent-Total	367	21-25	1.97	1.88	3.79	519.0	211.7	1078.3
BitTorrent-Core	367	21-25	2.52	2.19	5.32	485.1	181.1	1036.9
CoBlitz In-order Sync'd	380	8-12	2.50	2.78	3.52	222.4	143.6	434.3
CoBlitz In-order Staggered	383	5-9	2.99	3.26	4.54	122.4	141.7	406.4
CoBlitz In-order Cached	377	12-16	3.51	3.65	5.65	185.2	109.5	389.1
CoBlitz Out-of-order Sync'd	381	8-10	2.91	3.15	4.17	193.9	127.0	381.6
CoBlitz Out-of-order Staggered	384	5-8	3.68	3.78	5.91	105.4	124.6	365.2
CoBlitz Out-of-order Cached	379	8-13	4.36	4.08	7.46	164.3	98.1	349.5

Table 5.1: Throughputs (in Mbps) and times (in seconds) for various downloading approaches with all live PlanetLab nodes. The count of good nodes is the typical value for nodes completing the download, while the count of failed nodes shows the range of node failures.

and the CDFs shown are of all nodes completing the tests. The actual number of nodes finishing each test are shown in the table. In the throughput graph, lines to the right are more desirable, while in the download time graph, lines to the left are more desirable.

From the graphs, we can see several general trends: all schemes beat direct downloading, uncached CoBlitz generally beats BitTorrent, out-of-order CoBlitz beats in-order delivery, staggered downloading beats synchronized delivery, and cached delivery, even when synchronized, beats the others. Direct downloading at this scale is particularly problematic – we had to abruptly shut down this test because it was consuming most of Princeton's bandwidth and causing noticeable performance degradation.

The worst-case performance occurs for the uncached case where all clients request the content at exactly the same time and much load is placed on the origin server. However, this case is unlikely for regular users, since even a few seconds of difference in start times defeats this problem.

The fairest comparison between BitTorrent and CoBlitz is BT-Core versus CoBlitz

CoBlitz				BitTorrent
Sync		Stagger		
In-Order	Out	In-Order	Out	
7.0	7.9	9.0	9.0	10.0

Table 5.2: Bandwidth consumption at the origin, measured in multiples of the file size

out-of-order with Staggered, in which case CoBlitz beats BitTorrent by 11-72% in throughput and a factor of 1.5 to 4.6 in download time. Even the worst-case performance for CoBlitz, when all clients are synchronized on uncached content, generally beats BitTorrent by 27-48% in throughput and a factor of 1.47 to 2.48 in download time.

In assessing how well CoBlitz compares against BitTorrent, it is interesting to examine the 90^{th} percentile download times in Table 5.1 and compare them to the mean and median throughputs. This comparison has appeared in other papers comparing with BitTorrent [48, 77]. We see that the tail of BitTorrent's download times is much worse than comparing the mean or median values. As a result, systems that compare themselves primarily with the worst-case times may be presenting a much more optimistic benefit than seen by the majority of users.

It may be argued that worst-case times are important for systems that need to know an update has been propagated to their members, but if this is an issue, more important than delay is failure to complete. In Table 5.1, we show the number of nodes that finish each test, and these vary considerably despite the fact that the same set of machines is being used. Of the approximately 400 machines available across the union of all tests, only about 5-12 nodes fail to complete using CoBlitz, while roughly 17-18 fail in direct testing, and about 21-25 fail with BitTorrent. The 5-12 nodes where CoBlitz eventually stops trying to download are at PlanetLab sites with highly-congested links, poor bandwidth, and other problems – India, Australia, and some Switzerland nodes.

5.5.2 Load at the Origin

Another metric of interest is how much traffic reaches the origin server in these different tests, and this information is provided in Table 5.2, shown as a multiple of the file size. We see that the CoBlitz scenarios fetch a total of 7 to 9 copies in the various tests, which yields a utility of 43-55 nodes served per fetch (or a cache hit rate of 97.6 - 98.2%). BitTorrent has comparable overall load on the origin, at 10 copies, but has a lower utility value, 35, since it has fewer nodes complete. For Shark, the authors observed it downloading 24 copies from the origin to serve 185 nodes, yielding a utility of 7.7. We believe that part of the difference may stem from peering policy – CoDeeN's unilateral peering approach allows poorly-connected nodes to benefit from existing clusters, while Coral's latency-oriented clustering may adversely impact the number of fetches needed.

A closer examination of fetches per chunk, shown in Figure 5.7, shows that CoBlitz's average of 8 copies varies from 4-11 copies by chunk, and these copies appear to be spread fairly evenly geographically. The chunks that receive only 4 fetches are particularly interesting, because they suggest it may be possible to cut CoBlitz's origin load by another factor of 2. We believe these chunks happen to be served by nodes that overlap with many peer sets, which would further validate CoBlitz's unilateral peering.

5.5.3 Performance after Flash Crowds

Finally, we evaluate the performance of CoBlitz after a flash crowd, where the CDN nodes can still have the file cached. This was one of motivations for building CoBlitz on top of CoDeeN – that by using an infrastructure geared toward long-duration caching, we could serve the object quickly even after demand for it drops. This test is shown in Figure 5.8, where clients at all PlanetLab nodes try downloading the file individually, with no two

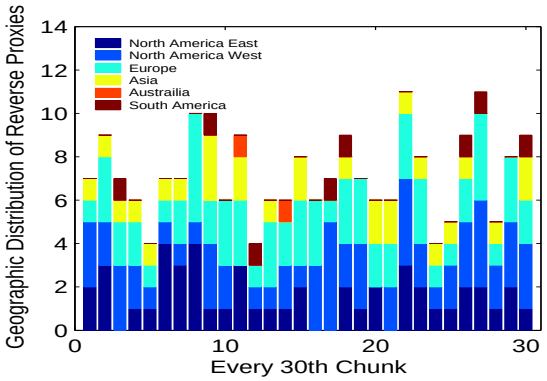


Figure 5.7: Reverse proxy location distribution

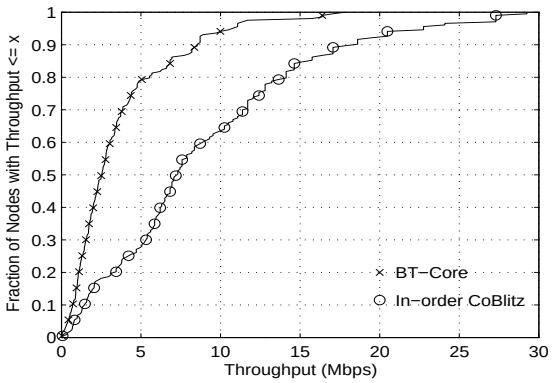


Figure 5.8: Single node download after flash crowds

operating simultaneously. We see that performance is still good after the flash crowd has dissipated – the median for this in-order test is above 7 Mbps, almost tripling the median for in-order uncached and doubling the median of in-order cached. At this bit rate, clients can watch DVD-quality video in real time. We include BitTorrent only for comparison purposes, and we see that its median has only marginally improved in this scenario.

5.5.4 Real-world Usage

One of our main motivations when developing CoBlitz was to build a system that could be used in production, and that could operate with relatively little monitoring. These decisions have led us not only to use simpler, more robust algorithms where possible, but also to restrict the content that we serve. To keep the system usage focused on large-file transfer, and to prevent general-purpose bandwidth cost-shifting, [9] we have placed restrictions on what the general public can serve using CoBlitz. Unless the original file is hosted at a university, CoBlitz will not serve HTML files, most graphics types, and most

[9]This would prevent users from abusing our service to shift their bandwidth cost to PlanetLab to serve their regular Web traffi c.

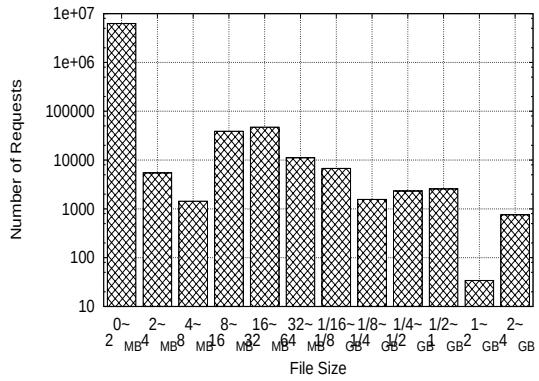


Figure 5.9: CoBlitz October 2006 usage by requests

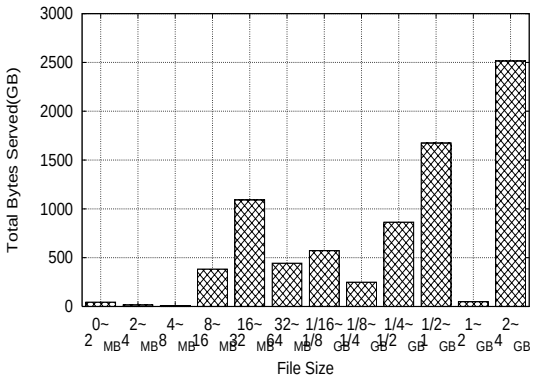


Figure 5.10: CoBlitz October 2006 usage by bytes served

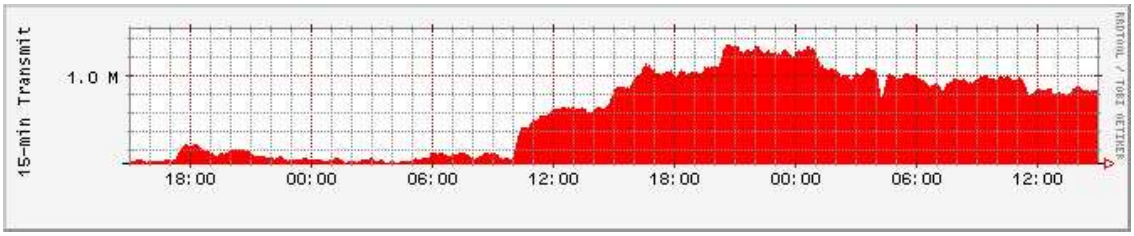


Figure 5.11: CoBlitz traffic in Kbps on release of Fedora Core 6, averaged over 15-minute intervals. 1.0 M in the graph represents 1 Gbps. The 5-minute peaks exceeded 1.4 Gbps.

audio/video formats. As a result of these policies, we have not received any complaints related to the content served by CoBlitz, which has simplified our operational overhead.

To get a sense of a typical month's CoBlitz usage, we present the breakdown for October 2006 traffic in Figures 5.9 (by number of requests) and 5.10 (by bytes served). Most of the requests for files less than 2MB come from the Stork service [80], which provides package management on PlanetLab, and the CiteSeer Digital Library [20], which provides document downloads via CoBlitz. The two spikes in bytes served are from the Fedora Core Linux distribution, available as either downloadable CD images or DVD images. Most file requests between 10MB and 100MB are audio and video podcasts from University Channel [83], which uses CoBlitz to deliver their public affairs lectures and

panel discussion. The remaining traffic comes from smaller sites, other PlanetLab users, and Fedora Core RPM downloads.

A more unusual usage pattern occurred on October 24, 2006, when the Fedora Core 6 Linux distribution was released. The measurements from this day and the previous day are shown in Figure 5.11. In less than an hour, CoBlitz went from an average of 100 Mbps of traffic to over 600 Mbps, and sustained 5-minute peaks exceeded 1.4 Gbps. The bandwidth consumed at the origin mirror server was only about 30-40 Mbps during the timeframe, maintaining the utility of the fetched content at 30-40. The seemingly lower utility number than those in our experiments (Section 5.5.2) is understandable because the mirror serves not just one file but many different files at the same time.

This flash crowd had a relatively long tail – it average 400-450 Mbps on the third day, and only dropped to less than 300 Mbps on the fourth day, a weekend. The memory footprint of CoBlitz was also low – even serving the CD and DVD images on several platforms (PPC, i386, x86_64), the average memory consumption was only 75MB per node.

5.6 Related Work

Several projects that perform large file transfers have been measured on PlanetLab, with the most closely related ones being Bullet' [48], and Shark [4], which is built on Coral [31]. Though neither system is currently accessible to the public, both have been evaluated recently. Bullet', which delivers chunks in out-of-order and uses UDP, is reported to achieve 7 Mbps when run on 41 PlanetLab hosts at different sites. In testing under similar conditions, CoBlitz achieves 7.4 Mbps (uncached) and 10.6 Mbps (cached) on average. We could potentially achieve even higher results by using a UDP-based transport protocol,

but our experience suggests that UDP traffic causes more problems in practice, [10] both from intrusion detection systems as well as stateful firewalls. Shark's performance for transferring a 40MB file across 185 PlanetLab nodes shows a median throughput of 0.96 Mbps. As discussed earlier, Shark serves an average of only 7.7 nodes per fetch, which suggests that their performance may improve if they use techniques similar to ours to reduce origin server load. The results for all of these systems are shown in Table 5.3.

The use of parallel downloads to fetch a file has been explored before, but in a more narrow context – Rodriguez *et al.* use HTTP byte-range queries to simultaneously download chunks in parallel from different mirror sites [69]. Their primary goal was to improve single client downloading performance, and the full file is pre-populated on all of their mirrors. What distinguishes CoBlitz from this earlier work is that we make no assumptions about the existence of the file on peers, and we focus on maintaining stability of the system even when a large number of nodes are trying to download simultaneously. CoBlitz works if the chunks are fully cached, partially cached, or not at all cached, fetching any missing chunks from the origin as needed. In the event that many chunks need to be fetched from the origin, CoBlitz attempts to reduce origin server overload. Finally, from a performance standpoint, CoBlitz attempts to optimize the memory cache hit rate for chunks, something not considered in Rodriguez's system.

While comparing with other work is difficult due to the difference in test environment, we can make some informed conjecture based on our experiences. FastReplica's evaluation includes tests of 4-8 clients, and their per-client throughput drops from 5.5 Mbps with 4 clients to 3.6 Mbps with 8 clients [18]. Given that their file is broken into a small number of equal-sized pieces, the slowest node in the system is the overall bottleneck. By using a large number of small, fixed-size pieces, CoBlitz can mitigate the

[10] Many firewalls do not allow incoming (or even outgoing) UDP traffic to a random port and sensitive Intrusion Detection Systems (IDS) often raise alarms on random UDP traffic.

	Shark	CoBlitz				Bullet'
		Uncached		Cached		
		In	Out	In	Out	
# Nodes	185	41	41	41	41	41
Median	1.0	6.8	7.4	7.4	9.2	
Mean		7.0	7.4	8.4	10.6	7.0

Table 5.3: Throughput results (in Mbps) for various systems at specified deployment sizes on PlanetLab. All measurements are for 50MB files, except for Shark, which uses 40MB.

effects of slow nodes, either by increasing the number of parallel fetches, or by retrying chunks that are too slow. Another system, Slurpie [77], limits the number of clients that can access the system at once by having each one randomly back off such that only a small number are contacting the server regardless of the number of nodes that want service. Their local-area testing has clients contact the server at the rate of one every three seconds, which staggers the request rate far more than BitTorrent. Slurpie's evaluation on PlanetLab provides no absolute performance numbers [77], making it difficult to draw comparisons. However, their performance appears to degrade beyond 16 nodes.

The scarcity of deployed systems for head-to-head comparisons supports part of our motivation – by reusing our CDN infrastructure, we have been able to easily deploy CoBlitz and keep it running.

Chapter 6

Conclusion

Today's Internet is increasingly dependent on the reliability and performance of large-scale distributed systems, but the current systems are often poorly designed to provide robust and scalable service. This dissertation has explored the necessary principles for building highly available and scalable systems from the perspectives of a decentralized content distribution network service, a reliable DNS name lookup service, and a scalable large-file transfer service.

By designing, building, and running CoDeeN, CoDNS, and CoBlitz on PlanetLab serving real traffic, we have gained insight for better system design. We find that active pairwise monitoring and independent peering greatly improves the system robustness and scalability, and greatly enhances the reliability of the overall service. We have also learned that intelligent composition of temporarily unreliable DNS services can produce a highly reliable lookup service with a minimal operational overhead. We believe that our findings are not specific to our systems and will be beneficial to designing other systems as well.

6.1 Reliability of Decentralized CDN

We have presented our experience with a continuously running prototype CDN on PlanetLab. We have described our reliability mechanisms that assess node health and prevent failing nodes from disrupting the operation of the overall system. The key insight is to allow pairwise monitoring and to let the nodes independently identify healthy peer nodes from their own view, and this simple design decision has made the system survive without degradation of performance as the scale of the system has quadrupled during the deployment. We believe that future services, especially peer-to-peer systems, will require similar mechanisms as more services are deployed on non-dedicated distributed systems, and as their interaction with existing protocols and systems increases.

Our distributed monitoring facilities prove to be effective at detecting and thus avoiding failing or problematic nodes. The net benefit is robustness against component disruptions and improved response latency. Although some aspects of these facilities seem application-specific, they are not confined to CDN services. Other latency-sensitive services running in a non-dedicated distributed environment can potentially benefit from them, since they also need to do extra reliability checks. Our experiences also reveal that reliability-induced problems occur almost two orders of magnitude more frequently than node joins/leaves, which makes active monitoring necessary and important for other types of systems such as peer-to-peer. We believe that our experiences with CoDeeN and the data we have obtained on availability can serve as a starting point for designers of reliable systems in the future.

6.2 Highly Available DNS Service

DNS is one of the core Internet infrastructures enabling ubiquitous access to other Internet resources. Over the two decades of operation, researchers have found many problems regarding its operational stability and performance, but they have mainly focused on the server-side aspects. However, in this dissertation, we have shown that client-side instability in DNS name lookups is widespread and relatively common, possibly suggesting one of the real causes for server-side communication delays or failures. The failure cases degrade average lookup time and increase the "tail" of response times. We show that these failures appear to be caused by temporary nameserver overload, and are largely uncorrelated across multiple sites. Through analysis of live traffic, we show that a simple peering system reduces response times and improves reliability.

Using these observations, we develop a lightweight name lookup service, CoDNS, that uses peers at remote sites to provide cooperative lookups during failures. CoDNS operates in conjunction with local DNS nameservers, allowing incremental deployment without significant resource consumption. We have shown that this system generates low overhead, cuts average response time by half or more, and increases DNS service availability by adding another '9' to that of existing service.

6.3 Scalable Large-file Transfer Service

We have shown that, with a relatively small amount of modification, a traditional, HTTP-based content distribution network can be made to efficiently support scalable large-file transfer. Even with no modifications to clients, servers, or client-side software, our approach provides good performance under demanding conditions, but can provide even

higher performance if clients implement a relatively simple HTTP feature, chunked encoding.

Additionally, we show how we have taken the experience gained from the past 30 months of CoBlitz deployment, and used it to adapt our algorithms to be more aware of real-world conditions. We demonstrate the advantages provided by this approach by evaluating CoBlitz's performance across all of PlanetLab, where it exceeds the performance of BitTorrent as well as all other research efforts known to us.

In the process of making CoBlitz handle scale and reduce congestion both within the CDN and at the origin server, we identify a number of techniques and observations that we believe can be applied to other systems of this type. Among them are: (a) unilateral peering, which simplifies communication as well as enabling the inclusion of the sites restricted by their local policy or poorly-connected nodes, (b) request re-forwarding to reduce the origin server load when nodes send requests to an overly-broad replica set, (c) dynamically adjusting replica sets to reduce burstiness in small time scales, (d) congestion-controlled parallel chunk fetching, to reduce both origin server load as well as self-interference at slower CDN nodes.

We believe that the lessons we have learned from CoBlitz should help not only the designers of future systems, but also provide a better understanding of how to design these kinds of algorithms to reflect the unpredictable behavior we have seen in real deployment.

Bibliography

[1] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, Dec. 2002.

[2] Akamai Technologies Inc. `http://www.akamai.com/`.

[3] P. Albitz and C. Liu. *DNS and BIND*, pages 292–293. O'REILLY, fourth edition, 2001.

[4] S. Annapureddy, M. J. Freedman, and D. Mazieres. Shark: Scaling file servers via cooperative caching. In *2nd USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '05)*, Boston, MA, May 2005.

[5] O. Babaoglu, R. Davoli, A. Montresor, and R. Segala. System support for partition-aware network applications. In *Proceedings of 18th International Conferences on Distributed Computing Systems ICDCS*, pages 184–191, 1998.

[6] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, M. Stemm, and R. H. Katz. TCP behavior of a busy internet server: Analysis and improvements. In *INFOCOM (1)*, pages 252–262, 1998.

[7] M. Beck, D. Arnold, A. Bassi, F. Berman, H. Casanova, J. Dongarra, T. Moore,

G. Obertelli, J. Plank, M. Swany, S. Vadhiyar, and R. Wolski. Logistical computing and internetworking: Middleware for the use of storage in communication. In *3rd Annual International Workshop on Active Middleware Services (AMS)*, San Francisco, August 2001.

[8] R. Bhagwan, S. Savage, and G. Voelker. Understanding availability. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, Feb. 2003.

[9] K. Birman. The process group approach to reliable distributed computing. *Communications of ACM*, 36(12):36–53, 1993.

[10] S. Birrer, D. Lu, F. E. Bustamante, Y. Qiao, and P. Dinda. FatNemo: Building a resilient multi-source multicast fat-tree. In *Proceedings of 9th International Workshop on Web Content Caching and Distribution (IWCW'04)*, Beijing, China, October 2004.

[11] B. Biskeborn, M. Golightly, K. Park, and V. S. Pai. (Re)Design considerations for scalable large-file content distribution. In *Proceedings of the Second Workshop on Real, Large Distributed Systems (WORLDS)*, San Francisco, CA, December 2005.

[12] L. Brakmo, S. O'Malley, and L. Peterson. TCP Vegas: New techniques for congestion detection and avoidance. In *Proceedings of SIGCOMM '94*, August 1994.

[13] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proceedings of IEEE INFOCOM*, pages 126–134, 1999.

[14] CacheLogic, 2004. `http://www.cachelogic.com/news/pr040715.php`.

[15] M. Castro, P. Druschel, A. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: High-bandwidth content distribution in a cooperative environment. In *Proceedings of SOSP'03*, Oct 2003.

[16] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, 1999.

[17] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A hierarchical Internet object cache. In *USENIX Annual Technical Conference*, 1996.

[18] L. Cherkasova and J. Lee. FastReplica: Efficient large file distribution within content delivery networks. In *Proceedings of the 4th USITS*, Seattle, WA, March 2003.

[19] Y. Chu, S. G. Rao, S. Seshan, and H. Zhang. A case for end system multicast. In *IEEE Journal on Selected Areas in Communication (JSAC), Special Issue on Networking Support for Multicast*, 2002.

[20] CiteSeer Scientific Literature Digital Library. `http://citeseer.ist.psu.edu/`.

[21] CNN News, August 2, 2006. `http://money.cnn.com/2006/08/02/smbusiness/domains/index.htm`.

[22] CoDeeN status page. `http://codeen.cs.princeton.edu/status/`.

[23] B. Cohen. Bittorrent, 2003. `http://bitconjurer.org/BitTorrent`.

[24] E. Cohen and H. Kaplan. Prefetching the Means for Document Transfer: A New Approach for Reducing Web Latency. In *Proceedings of IEEE INFOCOM*, pages 854–863, 2000.

[25] R. Cox, A. Muthitacharoen, and R. Morris. Serving DNS Using Chord. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.

[26] F. Cristian. Reaching agreement on processor group membership in synchronous distributed systems. *Distributed Computing*, 4:175–87, 1991.

[27] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A decentralized network coordinate system. In *Proceedings of SIGCOMM '04*, Portland, Oregon, August

2004.

[28] P. B. Danzig, K. Obraczka, and A. Kumar. An Analysis of Wide-Area Name Server Traffic: A Study of Internet Domain Name System. In *Proceedings of ACM SIGCOMM*, 1992.

[29] D. Eastlake. Domain Name System Security Extensions. RFC 2535, January 1999.

[30] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. RFC 2616, June 1999.

[31] M. Freedman, E. Freudenthal, and D. Mazieres. Democratizing content publication with Coral. In *Proceedings of the 1st Symposium on Networked System Design and Implementation (NSDI '04)*, 2004.

[32] M. J. Freedman, K. Lakshminarayanan, S. Rhea, and I. Stoica. Non-transitive connectivity and DHTs. In *Proceedings of 2nd Workshop on Real, Large, Distributed Systems (WORLDS)*, 2005.

[33] Ganglia. http://ganglia.sourceforge.net.

[34] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, 2003.

[35] GNU wget. `http://www.gnu.org/software/wget/wget.html/`.

[36] G. Goldszmidt and G. Hunt. NetDispatcher: A TCP connection router. Technical Report RC 20853, IBM Research White Paper, 1997.

[37] R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. The impact of DHT routing geometry on resilience and proximity. In *Proceedings of ACM SIGCOMM*, 2003.

[38] A. Gupta, B. Liskov, and R. Rodrigues. One hop lookups for peer-to-peer overlays. In *Ninth Workshop on Hot Topics in Operating Systems (HotOS-IX)*, 2003.

[39] C. Huitema and S. Weerahandi. Internet Measurements: the Rising Tide and the DNS Snag. In *Proceedings of the 13th ITC Specialist Seminar on Internet Traffic Measuremnet and Modelling*, 2000.

[40] JANET Web Cache Service. http://wwwcache.ja.net.

[41] K. Johnson, J. Carr, M. Day, and F. Kaashoek. The Measured Performance of Content Distribution Networks. In *Proceedings of the 5th International Web Caching and Content Delivery Workshop (WCW)*, 2000.

[42] J. Jung, E. Sit, H. Balakrishnan, and R. Morris. DNS Performance and the Effectiveness of Caching. In *IEEE/ACM Transactions on Networking*, volume 10, 2002.

[43] J. Kangasharju and K. W. Ross. A Replicated Architecture for the Domain Name System. In *Proceedings of IEEE INFOCOM*, pages 660–669, 2000.

[44] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi. Web caching with consistent hashing. In *Proceedings of the 8th International World-Wide Web Conference*, 1999.

[45] D. R. Karger, E. Lehman, F. T. Leighton, R. Panigrahy, M. S. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *ACM Symposium on Theory of Computing*, 1997.

[46] J. Kneschke. lighttpd. `http://www.lighttpd.net`.

[47] B. Knowles. Domain Name Server Comparison: BIND 8 vs. BIND 9 vs. djbdns vs. ???, 2002. http://www.usenix.org/events/lisa02/tech/presentations/knowles_ppt/.

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

[49] D. Kostić, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: high bandwidth data dissemination using an overlay mesh. In *Proceedings of 19th ACM SOSP*, 2003.

[50] A. Kumar, J. Postel, C. Neuman, P. Danzig, and S. Miller. Common DNS Implementation Errors and Suggested Fixes. RFC 1536, October 1993.

[51] R. Liston, S. Srinivasan, and E. Zegura. Diversity in DNS Performance Measures. In *Proceedings of the ACM SIGCOMM Internet Measurement Workshop*, 2002.

[52] B. Maggs. Personal communication (email) with Vivek S. Pai, October 20th, 2005.

[53] P. Mockapetris. Domain Names - Implementation and Specification. RFC 1035, November 1987.

[54] P. Mockapetris and K. Dunlap. Development of the Domain Name System. In *Proceedings of ACM SIGCOMM*, pages 123–133, 1988.

[55] National Laboratory for Applied Network Research (NLANR). Ircache project. http://www.ircache.net/.

[56] V. S. Pai, L. Wang, K. Park, R. Pang, and L. Peterson. The dark side of the web: An open proxy's view. In *Proceedings of the 2nd Workshop on Hot Topics in Networking (HotNets-II)*, Cambridge, MA, Nov 2003.

[57] K. Park, V. Pai, L. Peterson, and Z. Wang. CoDNS: Improving DNS Performance and Reliability via Cooperative Lookups. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation(OSDI)*, San francisco, CA, December 2004.

[58] K. Park and V. S. Pai. Deploying large file transfer on an http content distribution network. In *Proceedings of the First Workshop on Real, Large Distributed Systems (WORLDS '04)*, San Francisco, CA, December 2004.

[59] K. Park and V. S. Pai. CoMon: A mostly-scalable monitoring system for PlanetLab. In *ACM SIGOPS Operating Systems Review*, 2006.

[60] K. Park and V. S. Pai. Scale and performance in the CoBlitz large-file distribution service. In *Proceedings of the 3rd USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '06)*, 2006.

[61] K. Park, V. S. Pai, K.-W. Lee, and S. Calo. Securing web service by automatic robot detection. In *Proceedings of the USENIX Annual Technical Conference: Systems Practice & Experience Track (USENIX '06)*, 2006.

[62] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proceedings of the 1st ACM Workshop on Hot Topics in Networks (HotNets-I)*, October 2002.

[63] M. Rabinovich, J. Chase, and S. Gadde. Not all hits are created equal: Cooperative proxy caching over a wide-area network. *Computer Networks and ISDN Systems*, 30(22–23):2253–2259, 1998.

[64] V. Ramasubramanian and E. G. Sirer. Beehive: O(1) Lookup Performance for Power-Law Query Distributions in Peer-to-Peer Overlays. In *1st Symposium on Networked Systems Design and Implementation (NSDI)*, 2004.

[65] V. Ramasubramanian and E. G. Sirer. The Design and Implementation of a Next Generation Name Service for the Internet. In *Proceedings of ACM SIGCOMM*, 2004.

[66] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of ACM SIGCOMM'01*, Aug. 2001.

[67] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. In *Proceedings of the USENIX Annual Technical Conference*, June 2004.

[68] R. Rodrigues, B. Liskov, and L. Shrira. The design of a robust peer-to-peer system. In *Tenth ACM SIGOPS European Workshop*, 2002.

[69] P. Rodriguez, A. Kirpal, and E. W. Biersack. Parallel-access for mirror sites in the

Internet. In *Proceedings of IEEE Infocom*, Tel-Aviv, Israel, March 2000.

[70] A. Rousskov. Range requests - squid mailing list. `http://www.squid-cache.org/mail-archive/squid-dev/199801/0005.html`

[71] A. Rousskov, M. Weaver, and D. Wessels. The fourth cache-off. `http://www.measurement-factory.com/results/`.

[72] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, Nov. 2001.

[73] S. Saroiu, P. Gummadi, and S. Gribble. A measurement study of peer-to-peer file sharing systems, 2002.

[74] M. Schiffman. A Samping of the Security Posture of the Internet's DNS Servers. http://www.packetfactory.net/papers/DNS-posture/.

[75] A. Schiper and A. Sandoz. Uniform reliable multicast in a virtually synchronous environment. In *Proceedings of 13th International Conferences on Distributed Computing Systems ICDCS*, pages 561–8, 1993.

[76] A. Shaikh, R. Tewari, and M. Agrawal. On the Effectiveness of DNS-based Server Selection. In *Proceedings of IEEE INFOCOM*, 2001.

[77] R. Sherwood, R. Braud, and B. Bhattacharjee. Slurpie: A cooperative bulk data transfer protocol. In *Proceedings of IEEE Infocom*, Hong Kong, 2004.

[78] N. Spring, L. Peterson, A. Bavier, and V. S. Pai. Using PlanetLab for network research: Myths, realities, and best practices. *Operating Systems Review*, 40(1), 2006.

[79] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of ACM SIGCOMM 2001*, San Diego, California, Aug. 2001.

[80] Stork on PlanetLab.

`http://www.cs.arizona.edu/stork/`.

[81] R. Tewari, M. Dahlin, H. M. Vin, and J. S. Kay. Design considerations for distributed caching on the Internet. In *International Conference on Distributed Computing Systems*, pages 273–284, 1999.

[82] D. Thaler and C. Ravishankar. Using Name-based Mappings to Increase Hit Rates. In *IEEE/ACM Transactions on Networking*, volume 6, 1, pages 1–14, 1998.

[83] University Channel.

`http://uc.princeton.edu/`.

[84] R. van Renesse, K. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. In *ACM Transactions on Computer Systems*, May 2003.

[85] L. Wang, V. Pai, and L. Peterson. The Effectiveness of Request Redirecion on CDN Robustness. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation(OSDI)*, Boston, MA, December 2002.

[86] L. Wang, K. Park, R. Pang, V. Pai, and L. Peterson. Reliability and security in the CoDeeN content distribution network. In *Proceedings of the USENIX Annual Technical Conference*, 2004.

[87] C. E. Wills and H. Shang. The Contribution of DNS Lookup Costs to Web Object Retrieval. Technical Report WPI-CS-TR-00-12, Worcester Polytechnic Institute (WPI), 2000.

[88] A. Wolman, G. M. Voelker, N. Sharma, N. Cardwell, A. R. Karlin, and H. M. Levy. On the scale and performance of cooperative web proxy caching. In *Symposium on Operating Systems Principles*, 1999.