Building Robust Network Services Through Efficient Resource Management

Limin Wang

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

November 2003

Abstract

Network services have been increasingly integrated into our daily lives, but their accessibility and stability are also frequently impacted by flash crowds or Denial of Service (DoS) attacks. To be immune from flash crowds or DoS attacks, robust network services must possess two important qualities: completeness and generality. Completeness implies that all resources must be protected, including CPU time, memory and disk capacity, and link bandwidth. Generality means not handling attacks or faults as extraordinary events, but instead treating them within the same framework used during normal operations. Fundamentally, this is a matter of efficient management of networked resources.

Toward this end, we use Content Distribution Networks (CDN) as an example, and investigate how request redirection impacts CDN robustness. CDN systems deploy redundant resources (servers) geographically distributed across the Internet and distribute client requests to an appropriate server based on a variety of factors—e.g., server load, network proximity, cache locality—in an effort to reduce response time and increase the system capacity under load. We explore the design space of the redirection strategies employed by request redirectors, and define a class of new algorithms that carefully balance load, locality, and proximity. We use large-scale detailed simulations to evaluate various strategies. These simulations demonstrate the effectiveness of our new algorithms, which yield a 60-91% improvement in system capacity when compared with the best published CDN technology, yet user-perceived response latency remains low and the system scales well with the number of servers. We also build a prototype CDN, named CoDeeN, on the PlanetLab testbed. CoDeeN helps us to gain experience on managing and monitoring an operational CDN, and will be used in future research.

Through this thesis, we demonstrate that the resilience of large wide area network services can be improved through efficient management of networked resources. By adapting unified resource management schemes, we present a practical way to build network systems that not only handle a larger volume of regular traffic more easily, but also absorb flash crowds and deter DoS attacks as a natural part of their operations.

Acknowledgments

This dissertation represents a milestone in my life. It would not be possible without the support and guidance from many people. I would like to sincerely thank them.

I feel so blessed to have two excellent advisors who guided me through my thesis research, Larry Peterson and Vivek Pai. Larry is always my source of guidance and encouragement. His patience, keen insight and vision in research carried me through my Ph.D. study and helped me to stay focused. I have always been amazed by his ability to manage time and organize tasks so efficiently. I wish I could steal his gift someday. Vivek helps me enormously in many ways. He is very sharp both at identifying high-level problems and also at finding detailed solutions. His insightful feedback on my ideas often cleared my uncertainty quickly and eased my research a lot. I benefited so much from the discussions with him. His confidence and persistence also deeply affected me. I am fortunate to have constant support from both of them, otherwise, I would not have gone this far.

I would also like to thank all my other committee members: Andrea LaPaugh, Brian Kernighan and Randy Wang. Especially, Andrea, as a reader, has given me timely feedback on making my dissertation logically and technically sound, in spite of her busy schedules.

I am also grateful to Kai Li, who supported me through my general exam during my early days at Princeton, and has always been supportive. I also thank Douglas Clark and Edward Felten for their help at various times.

I am in debt to our graduate coordinator, Melissa Lawson, for patiently assisting me with so many things and making life much easier for not just me but all the graduate students. My research would also not be so smooth without support from our excellent

CS technical staff team: Jim Robert, Chris Tengi, Joe Crouthamel, Brian Jones, Chris Miller and Steve Elgersma.

I have been privileged to work with many wonderful colleagues including Andy Bavier, Zuki Gottlieb, Scott Karlin, Aki Nakao, Ruoming Pang, Xiaohu Qie, Nadia Shalaby, Tammo Spalink, Mike Wawrzoniak and Yuanyuan Zhou. I am specially thankful to Yuanyuan, who helped me to start my research at Princeton and has been a source of help ever since. I would like to thank my officemates who have made the working place a pleasant one: Kevin Skadron, George Tzanetakis, Jiannan Zheng, Lena Petrovic, Xiaohu Qie, Rob Kalnins, Zuki Gottlieb, Renato Werneck, Han Chen, Elena Nabieva and Subhash Khot. Thanks also go to many of my friends, who helped me in different ways and enriched my graduate student life: Han Chen, Jie Chen, Juan Chen, Mao Chen, Yuqun Chen, Wenjia Fang, Minwen Ji, Dongming Jiang, Zhiyan Liu, Rouming Pang, Yaoping Ruan, Hongzhang Shan, Yaoyun Shi, Zhijie Shi, Bin Wei, Xiaodong Wen, Yuan Xie, Xiang Yu, Lintao Zhang, Yuexing Zhang, Jiannan Zheng and Yuanyuan Zhou, to name just a few.

I am also grateful to people outside Princeton for their help on my research. I have enjoyed the joint work with Steven Low at Caltech on TCP modeling and look forward to future collaboration opportunities. I had a very pleasant summer intern at AT&T Research in 1999. I appreciate the guidance and help from my two mentors, Fred Douglis and Misha Rabinovich.

My deepest appreciation goes to my family who have given me all the love and support I need. My parents, Qikun and Shurong, have always tried their best to provide me with all kinds of support since I was a child. Their affections and expectations have accompanied me through so many years of my education. They deserve my utmost gratitude. My wife, Zhijian, has provided her unconditional love, patience and care throughout my Ph.D. time. She has always encouraged and supported me through thick and thin. She makes my life enjoyable, meaningful and complete. I cannot say enough thanks to her. I am also grateful to my parents-in-law who have given me a lot of support and encouragement. I am fortunate to have an elder brother, Weimin, who along with his family has helped and guided me in many ways, especially when we are far away from home.

This research is supported in part by DARPA under contract F30602–00–2–0561. The equipment used in simulations is supplied by Compaq and Hewlett-Packard/Intel. The testbed environment and proxy servers used by CoDeeN prototype CDN are supplied by PlanetLab and iMiMic, respectively. I am honored to be a Francis Upton Graduate Fellow in the School of Applied Science and Engineering at Princeton University for four years.

To my dear wife, Zhijian

To my parents, Qikun and Shurong

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As the Internet becomes more integrated into our everyday lives, the availability and stability of information services built on top of it becomes increasingly important. However, overloaded servers and congested networks present challenges to maintaining high accessibility. To alleviate these bottlenecks, it is becoming increasingly common to construct network services using redundant resources. If well managed, redundant resources are expected to provide protection to network services by eliminating single points of failure and improving the system's resilience under overload. Nevertheless, even armed with redundant resources, current network services are still frequently impacted by different kinds of abnormal traffic, and as a result, failures or attacks still cause disruptions in the service. This discrepancy between our expectation and reality motivates us to study how to efficiently harness massive resources to make network services more robust. To be more specific, we need to understand what are the challenges facing the robustness of network services, what are the general principles we need to follow to improve service resilience, and most importantly, how to apply these principles to tackle the challenges in practice.

In this thesis, we consider two sources of service failures or faults, flash crowds and Denial of Service (DoS) attacks. We believe that in order for network services to be immune from these service robustness threats, they have to possess two important qualities, *completeness* and *generality*. Completeness means we have to protect *all* the resources on the network. Generality means not handling attacks or faults as extraordinary events, but instead treating them within the same framework used during normal operations. Following these two general guidelines, we take Content Distribution Networks (CDN) as an example, and demonstrate a practical way of building robust network services through unified resource management and scheduling schemes.

1.1 Challenges to Service Robustness

Once a network service becomes available on top of the Internet, it is potentially exposed to an unlimited audience. However, it also means that the service has to handle all kinds of traffic. Some of the traffic could well exceed the servers' or networks' capacities, and some could simply be designed to disrupt the service. The robustness of network services is therefore constantly tested by different sources. These sources could be friendly, in the form of a flash crowd, or malicious, such as denial of service attacks.

In order to make network services more robust against these challenges, we need to first understand the nature of these challenges. To provide a detailed survey of all the robustness challenge sources is beyond the scope of this thesis. Here, we will focus on the following two threats—flash crowds and DoS attacks.

1.1.1 Flash Crowds

A *flash crowd* describes a situation where a large number of individuals try to access a particular service simultaneously. Interestingly, the term *flash crowd* can be dated back to a 1971 science fiction short story [54], where it referred to the situation when thousands of people went back in time to see historical events anew through teleportation. The result of a flash crowd on the Internet is typically a “hot spot” or some sort of congestion. In reality, many events can trigger flash crowds, sometimes well anticipated, such as the Star War trailer release and sports events like the Super Bowl, and sometimes hard to predict, such as breaking news stories.

Since flash crowds normally appear as a sudden drastic surge in request volumes, many services cannot handle them gracefully, even when the services have been over-provisioned. The network itself can become severely congested with many packet losses and retransmissions, and servers can be swamped by very high request streams. Users trying to get services during a flash crowd usually encounter very frustrating experiences: responses may take arbitrarily long times, or there is simply no response. The consequences of flash crowds for revenue critical on-line services can be even more severe than just unhappy users, and service degradation can cost millions of dollars. In short, flash crowds present a serious challenge to keeping network services stable and robust. In order to handle flash crowds, fundamentally, we have to employ and manage enough redundant resources to absorb the increased demand.

1.1.2 Denial of Service Attacks

If flash crowds are still considered “friendly” in the sense that they are the price paid for a successful service, the second kind of challenge we consider—Denial of Service (DoS) attacks—is “malicious”. As the name suggests, DoS attacks prevent legitimate accesses to certain services. The way DoS works is usually through exploiting bugs or software weaknesses to slow down or crash routers, servers and other network devices, or simply by overwhelming servers or routers with high workloads in an effort to exhaust limited resources such as network bandwidth and processing cycles. Although DoS attacks take many forms, the end results are similar when they are successful: well-behaved users are blocked from accessing the targeted system [33]. The consequences of these attacks are severe and disruptive to our daily lives. In a recent incident, the so-called *SQL Slammer* worm attacked many machines on the Internet via a vulnerability of Microsoft SQL Server 2000 [18]. This attack grounded flights, blocked ATMs, congested the Internet and interrupted services of many media, governments and companies. The damage was estimated to be as much as one billion dollars.

DoS attacks are hard to handle. They can target many different resources of network services, such as network bandwidth, CPU cycles, system memory, routers and DNS servers, and so on. They also evolve rapidly from simple “ping flood” or “SYN flood” to more subtle attacks that look no different from legitimate traffic [60]. As a result, detecting and stopping these attacks is becoming increasingly difficult. Moreover, damage to services can already be caused by attacks before we detect and stop them. Therefore, in addition to the detection and stopping mechanisms, our ability to mitigate attack damage is also very critical for DoS protection.

Recently, DoS attacks have also been launched from distributed locations, resulting in a more sophisticated variant, called Distributed Denial of Service (DDoS) attacks. Figure 1.1 shows the basic concept of DDoS, in which, an attacker first recruits a set of attacking agents, called *zombies* or *slave attackers*, by compromising innocent hosts across the Internet via some vulnerabilities of these nodes and installing attacking tools on

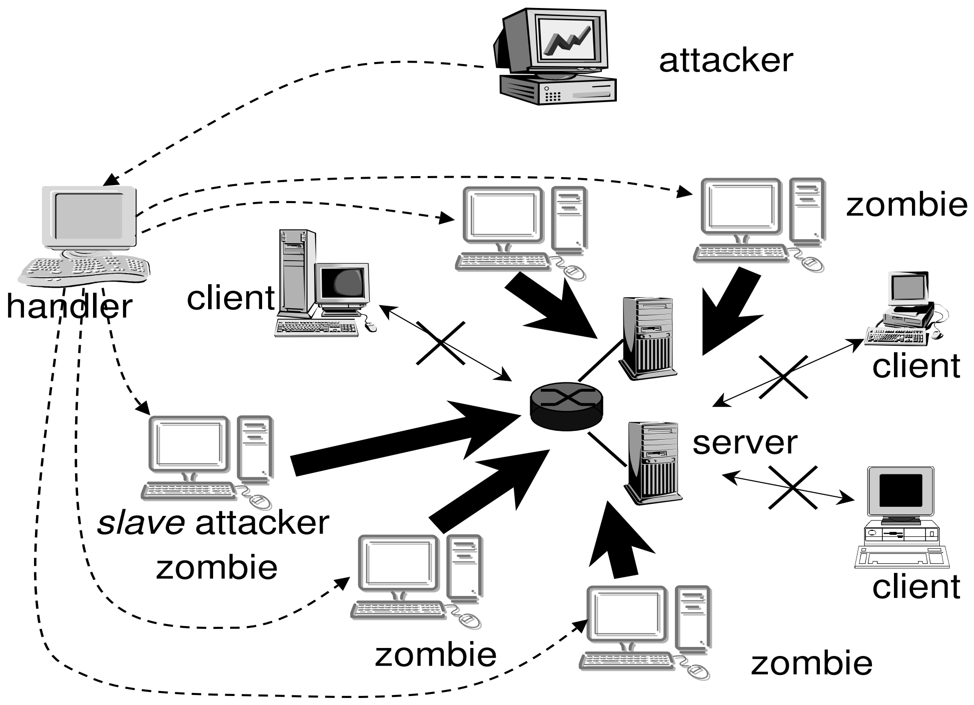

Figure 1.1: Distributed Denial of Service (DDoS)

them. To further hide the attacker's real entity, usually, one or more special zombies will act as *handlers* to control other slave attackers. Later, the real attacker will instruct the handler to launch a distributed DoS attack from those zombies that the handler controls. DDoS attacks are even more difficult to handle. On one hand, detecting and stopping DDoS attacks becomes more involved. Since zombies are simply normal hosts and there are many of them, it is hard to distinguish them from other regular users and filter their traffic. The existence of handlers and slave attackers also complicates the process of identifying the real culprit. Moreover, by employing more slaves, the attacker is able to use more resources for attacking and cause more damage. Therefore, to mitigate the damage of DDoS attacks also requires more effort.

DoS and DDoS attacks have become a big concern for the stability of the Internet [52]. Many countermeasures have been explored and applied to detect or prevent these attacks. They include applying software patches or upgrades, improving protocol and server designs, using intrusion detection systems to detect possible attacks, monitoring performance abnormalities and filtering spoofed packets. However, this thesis will not be devoted to these countermeasures. We will instead focus on how to effectively mitigate the potential damage of attacks through better utilizing the redundant resources of network services. Redundant resources are often employed to replicate services for performance reasons. However, the same set of resources can also be used to keep attacks from disrupting the services. This is particularly useful in the DDoS case, where the attackers can manipulate a large number of slave agents.

To summarize, flash crowds and DoS/DDoS attacks present serious challenges to the robustness of network services. Meanwhile, redundant resources have been increasingly employed in constructing current network services. These factors motivate us to search for ways to better harness these redundant resources to provide protection to network services.

1.2 Completeness and Generality

Dealing with different robustness challenges in large-scale network services has been an interesting topic in the fault tolerance and security research communities. The approach pursued by this thesis is to build robust network services through efficient resource management. This is based on the observation that different types of challenges, such as flash crowds and DoS attacks, consume a large amount of resources, eventually turning the service robustness protection problem into a “resource race” problem. As discussed earlier, to put network services in a better position in this resource race—in concert with specially tailored detection and protection schemes—redundant (possibly distributed) resources must also be widely used to provide protection against faults or attacks. How to efficiently utilize these existing abundant resources is of particular importance. To this end, we believe that in order for a network service to be robust against failures or attacks, it has to possess two important qualities, which we call *completeness* and *generality*.

- **Completeness** means we have to protect *all* the resources in the network. Completeness has two dimensions. One refers to different resources on a local node, such as processing cycles, memory and disk space, link bandwidth, and routing tables. The other refers to different distributed components that must work properly to provide the desired services, which may include routers, name servers and end servers.

- **Generality** means we should not treat faults or attacks as extraordinary events, but instead, we should handle them within the same framework used in normal operations. Special-case handling approaches are not regarded as general since they usually represent less-tested code paths.

It is easy to understand the first necessary quality of robust network services, since attacking or exhausting *any* resource can cause service degradation or interruption. As for the generality property, we actually treat protection against flash crowds, DoS attacks and other faults as a special case of the general problem of resource scheduling, and want to achieve a fair and load-balanced resource allocation. A fair and load-balanced system should provide fault-tolerance and protection against attacks for free, since attacks are extreme forms of unfair resource allocations and load imbalance. However, the converse is usually not true. To deal with faults and attacks, simply following a detect-and-correct model has several drawbacks. First, we have to devise accurate attack or fault detection mechanisms, distinguishing between good and bad traffic with low rates of false positives/negatives. To date, this is still an open problem. Moreover, recent research indicates that some DoS attacks take the form of legitimate traffic, making them even more difficult to distinguish from normal traffic with high request rates [60]. Second, if specially designed extra code paths are added into existing systems without being exercised during normal operations, we may not have confidence in their effectiveness. Third, it is usually the case that when attacks or faults are detected, damage has already been done to the services. The lag between the occurrence of attacks or faults and the actual protection actions could cause serious consequences. Therefore, instead of employing a set of special "bells and whistles" just for the sake of attacks and faults, we choose to take a unified approach and treat normal operations, attacks and faults within the same framework. By doing so, we not only provide a better service under normal conditions, but also gain a higher degree of confidence under abnormal conditions such as flash crowds and attacks. This is because we are confident that our mechanisms have been extensively tested and will react responsibly to robustness challenges.

Based on these guidelines, this thesis takes Content Distribution Networks as an example, to explore how a *unified*, efficient resource management mechanism can be used to make network services more robust.

In the rest of this chapter, we will first introduce our example system, Content Distribution Networks, including the research problem we face in improving CDN robustness. Then we discuss our research contributions and outline the remaining chapters of the thesis.

1.3 Content Distribution Networks

Content Distribution Networks (CDN) [1, 26, 50] are a primary example of the recent trend of constructing network services using redundant resources. CDNs deploy geographically dispersed server surrogates across the Internet and distribute client requests to an "appropriate" server based on various considerations. In this section, we describe several general aspects of CDNs, discuss the motivation of this research and identify the research problem we try to solve. More detailed information about CDNs can be found in [62, 65].

1.3.1 Overview

The World Wide Web (WWW) has become a common vehicle to publish, retrieve and exchange information on the Internet. More and more business transactions are being carried out through the WWW as well. However, WWW services suffer from various performance bottlenecks. It is generally agreed that while downloading web pages, there are three potential bottlenecks.

- **The last mile**. This refers to the link between the end user and its ISP. End users used to get connected to the Internet by a 56Kbps modem. Recent rapid growth of broadband connections, such as DSL and cable modems, improves this situation. But normally, compared to the well-provisioned Internet backbone, last mile links are still relatively slow.

- **The first mile**. This refers to the link connecting the server to the Internet. Under heavy traffic, both the first mile links and servers themselves can be overwhelmed.

- **Peering points.** To have a global connectivity, traffic from one ISP needs to pass or reach other ISPs, so there are crossing points between these ISPs where they exchange traffic. This traffic exchange is mainly directed by ISPs peering relationships and various local policies. Those ISPs that collectively implement the backbone of the Internet have little incentive to buy unused connectivity from or provide link capacity higher than the service level agreements to their peers. Peering links often run at full capacity, and links with very high utilization tend to exhibit high packet loss rates and high latency. Therefore, if a request needs to cross different ISPs, there is a chance that the request can be delayed or even dropped when the traffic is heavy.

Solving the first problem usually relies on the upgrade of end users' connections. CDNs, on the other hand, use replication to address the second and third bottlenecks.

The general idea of a CDN is to deploy a large set of geographically distributed *server surrogates* that cache web pages normally stored in *origin servers*. These surrogates are then used to service client requests. Doing so has two main advantages. First, instead of letting every client wait to connect to the origin server, now the request load can spread across multiple surrogates. This solves the first mile bottleneck through distributing the load. Second, if a surrogate happens to reside in the same ISP as some clients or be located close to the clients, the requests from these clients can be served without crossing peering points. This solves the possible peering points bottleneck. Building such a large infrastructure with thousands of server surrogates purely for a single website is certainly not cost-effective. In practice, commercial CDNs are provided by content distribution service providers (CDSPs), such as Akamai [1], which manages more than 13,000 servers in about 500 networks across the Internet with many client companies. These servers form an application *overlay network*, allowing multiple web sites to share this overlay and become customers of the CDSPs.

1.3.2 Components

A CDN typically has two main components: server surrogates and request redirectors. Figure 1.2 illustrates these two kinds of entities. We now consider them in more detail.

- **Server Surrogates**. Server surrogates are distributed elements used to cache content and service client requests. These surrogates look similar to traditional proxy caches, which will retrieve a page for a client when the requested page is not present in these caches. CDN surrogates can work exactly the same way, but the origin servers can also *proactively* propagate the content to CDN surrogates instead of having them obtain the pages on-demand [32]. Normally, only static pages are cached in these surrogates. Recently, some dynamic content has also started being replicated on CDNs, but usually with the requirement that backend data, such as databases, and computations, such as query or aggregation, can be shipped to distributed locations. How to keep the cached documents, especially the dynamic ones, fresh and consistent is also a very important topic, but will not be explored further in this thesis. Another important issue is where to strategically place these surrogates. Ideally, there should be at least one surrogate in each ISP or autonomous system (AS). The wider these surrogates are spread, the larger the potential benefits. As shown in Figure 1.2, each surrogate can possibly cache and serve any content from all of the CDN customers (content providers). However, each surrogate has only limited resources, such as link bandwidth, CPU cycles and memory cache, so the request rate each surrogate handles and the working set on each surrogate have to be controlled as well. How to allocate and manage these distributed surrogate resources will largely rely on the second kind of entity discussed below.

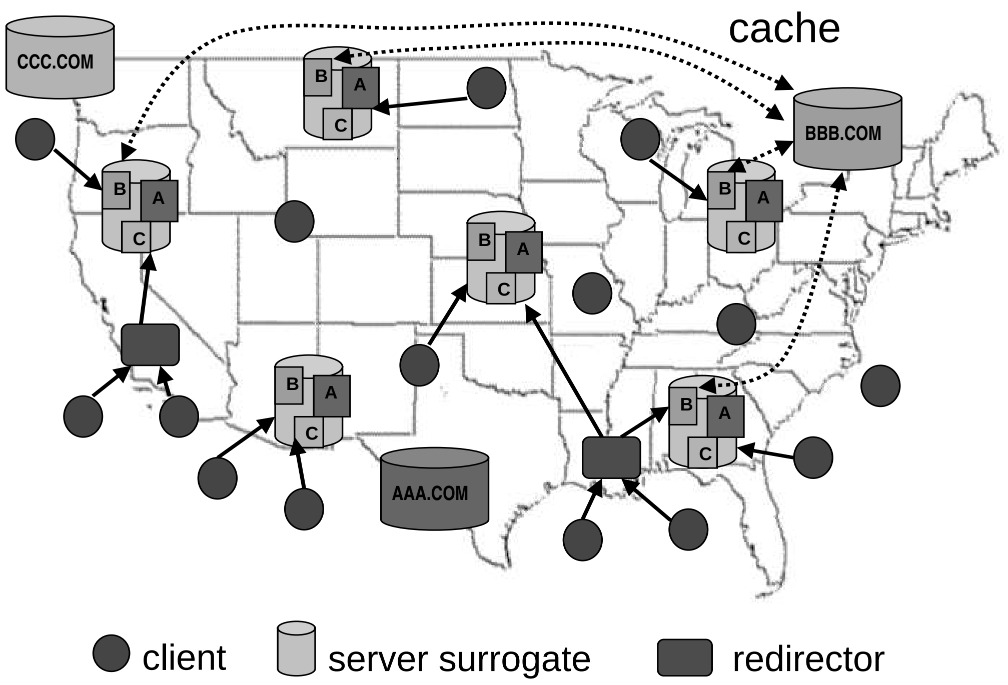

Figure 1.2: Content Distribution Networks

- **Request Redirectors**. To minimize the interface changes seen by clients, the existence of CDN surrogates is usually kept transparent. To take advantage of distributed surrogates, CDNs need to provide a set of *redirectors* that forward client requests to the most appropriate server, as shown in Figure 1.2. Logically, redirectors stand between clients and server surrogates. Practically, redirectors can be implemented in many ways. There are also different redirection strategies for selecting a server for a particular request, which will be covered more thoroughly in Chapter 2. Although what a redirector does is simply to assign a request to a surrogate, this assignment process implicitly allocates and manages resources of the CDN. As a result, the request redirection strategies employed by redirectors will impact the resource utilization and the resilience of CDN systems.

1.3.3 Research Challenge

CDNs are designed to improve two performance metrics: *response time* and *system throughput*. Response time, usually reported as a cumulative distribution of latencies, is of obvious importance to clients, and represents the primary marketing case for CDNs. System throughput, the average number of requests that can be satisfied each second, is primarily an issue when the system is heavily loaded, for example, when a flash crowd is accessing a small set of pages, or a DDoS attacker is targeting a particular site [33]. System throughput represents the overall robustness of the system since either a flash crowd or a DDoS attack can make portions of the information space inaccessible.

As described in Section 1.3.2, request redirection plays an important role in utilizing CDN resources to achieve these two goals. Given a sufficiently widespread distribution of servers, CDNs use several, sometimes conflicting, factors to decide how to distribute client requests. For example, to minimize response time, a server might be selected based on its *network proximity*. In contrast, to improve the overall system throughput, it is desirable to evenly *balance* the load across a set of servers. Both throughput and response time are improved if the distribution mechanism takes *locality* into consideration by selecting a server that is likely to already have the page being requested in its cache. Although the exact combination of factors employed by commercial systems is not clearly defined in the literature, evidence suggests that the scale is tipped in favor of reducing response time.

In practice, systems using CDNs are still susceptible to attacks or flash crowds. The service outage of CNN, Yahoo and several other high-profile sites under DDoS attacks in February 2000 [36], presents such an example. Given the massive redundant resources on CDNs, there is certainly room for improving CDN systems' responses under attacks. This motivates us to conduct research on more efficiently managing the massive resources on CDNs to increase their resilience in abnormal situations. The research challenge is to find the right tradeoffs that a redirector needs to make when deciding where to forward a client request, such that the CDN's resources are better utilized and the overall system robustness is improved. Following our principles of *completeness* and *generality*, the right tradeoffs have two implications. First, we need to manage all the resources well: CPU, memory, disk and link resources on a local surrogate host, and all the distributed servers and bandwidth as a whole. Also, redirectors themselves have to be robust, and should not rely on any centralized control. Second, the goal of request redirection strategies is to make sure that under a unified framework, CDNs are not only responsive across a wide range of load conditions, but also resilient in the face of flash crowds and DoS attacks.

1.4 Related Work

The research problem identified above is related to many previous research efforts on various fronts. However, we believe our goal of designing a wide-area CDN redirection scheme that not only handles regular traffic better but also absorbs flash crowds and deters

DoS attacks makes our research unique. The following is a brief discussion of a few related research areas.

1.4.1 Cluster Schemes

Approaches for request distribution in clusters [19, 25, 37] generally use a switch or router through which all requests for the cluster pass. As a result, they can use various forms of feedback and load information from servers in the cluster to improve system performance. In these environments, the delay between the redirector and the servers is minimal, so they can have tighter coordination [3] than in schemes like ours, which are developed for wide-area environments. As seen later, we borrow from this area of work by adapting the fine-grained server set accounting of LARD/R approach [58].

1.4.2 Distributed Servers

Not surprisingly, we share some similarities with schemes for geographically distributed caches and servers, since our approach must address some of the same issues that exist in those environments. In the case of geographically distributed caches and servers, DNS-based systems can be used to spread load among a set of servers, as in the case of round-robin DNS [8], or it can be used to take advantage of geographically dispersed server replicas [12]. More active approaches [20, 30, 35] attempt to use load/latency information to improve overall performance. These approaches collect and estimate different load and latency information using server-push based or probing schemes, and try to balance the load of distributed servers. In contrast, we are primarily focused on designing distributed algorithms that balance load, locality and latency without centralized control. However, we can borrow their load/latency collecting and deriving techniques.

1.4.3 Web Caches and CDNs

Proxy web caches are used in different content distribution schemes. The simplest approach, the static cache hierarchy [15], performs well in small environments but fails to scale to much larger populations [85]. Other schemes involve overlapping meshes [86] or networks of caches in a content distribution network [43], presumably including commercial CDNs such as Akamai. As seen later, proxies are also used in our research.

Web caches were originally used for (relatively) static web pages. Several recent efforts are aimed at supporting the caching of dynamic pages. One solution is to provide the edge caches with a template of the web page that identifies the fragments of the web page that need dynamic aggregation and assembly [28]. One can also use an object dependence graph to propagate changes from the underlying data to the cached objects that are dependent on the data [14]. Others include sending and caching programs (computations) to the proxy cache server for database-backed web sites [47] and caching equivalent or partial results [48, 71]. Although our focus is not on caching dynamic content, these efforts will benefit from our research in the long run. With more content suitable for caching, our new redirection schemes will become even more effective in distributing the request load on origin servers and improving the overall robustness of the WWW service.

One important research question with CDNs is deciding where to place server surrogates or replicas, provided client access patterns and the network topology are known beforehand. Previous research efforts have focused on how to maximize or minimize certain objective functions subject to some constraints, such as topology and degrees of replication, to optimize performance. Some of them model replica placement as a dynamic programming problem assuming that the underlying network topologies are trees [46]. Others take a more general approach, using real and synthesized topologies and traces, and formulating the problem as a facility location problem or minimum K-median problem [64]. These problems are NP-hard, but approximation algorithms [16] are known. These server placement mechanisms are useful for CDN provisioning. However, most of these approaches either require full knowledge of the request patterns and topology information *a priori*, or have to infer their future trends based on history. It is also very expensive or takes too long for the system to respond to the changes of access patterns and network congestion in a very timely manner. Moreover, most of these schemes assume that each replica can potentially satisfy all the request traffic destined to it without explicitly considering the runtime constraints on the network link bandwidth or server processing capacity, which can become a problem under load. In contrast, our redirection schemes do not need any *a priori* knowledge on the access patterns. As we can see clearer later in this thesis, they actively monitor CDN status at runtime, and proactively react to network congestion and server overload under various conditions.

1.4.4 DDoS Detection and Protection

In response to DDoS attacks, researchers have recently developed techniques to identify the source of attacks using various traceback techniques, such as ITRACE [7], probabilistic packet marking [70] and the Source Path Isolation Engine (SPIE) [72]. These approaches are effective in detecting and confining attack traffic. With their success in deterring spoofing and suspicious traffic, attackers have to use more disguised attacks, for example taking control of large number of slave hosts and instructing them to attack victims with legitimate requests. Our new redirection strategy is effective in providing protection against exactly such difficult-to-detect attacks. Some recent research proposes to use overlay networks to provide protection against DoS and DDoS attacks, such as

SOS [45] and Mayday [2]. These efforts share with us the similar goals of harnessing resources on overlays to protect network services, but with different emphases, such as client authentication and packet filtering.

1.4.5 Peer-to-Peer Networks

Peer-to-peer systems provide an alternative infrastructure for content distribution. Typical peer-to-peer systems involve a large number of participants acting as both clients and servers, and they have the responsibility of forwarding traffic on behalf of others. Given their very large scale and massive resources, peer-to-peer networks could provide a potential robust means of information dissemination or exchange. Many peer-to-peer systems, such as CAN [66], Chord [76], and Pastry [67] have been proposed and they can serve as a substrate to build other services such as directory services [4], publish/subscribe systems [69] and file storage services [24, 68]. Most of these peer-to-peer networks use a distributed hash-based scheme to combine object location and request routing and are designed for extreme scalability up to hundreds of thousands of nodes and beyond. As we can see later, we also use a hash-based approach, but we are dealing one to two orders of magnitude fewer servers than the peers in these systems, and we expect relatively stable servers. As a result, much of the effort that peer-to-peer networks spend in discovery and membership issues is not needed for our work. Also, we require fewer intermediaries between the client and server, which translates to lower latency and less aggregate network traffic.

Compared to these previous experiences, the problem we try to solve is different in that we want to employ a general and efficient resource management scheme that is effective on improving CDN robustness. By harnessing massive resources on CDNs, we not only provide a better CDN performance under normal load, but also make absorbing flash crowds and deterring DoS attacks a natural part of CDN operations.

1.5 Thesis Contributions

This thesis addresses the problem of designing a request distribution mechanism that manages all the CDN resources efficiently, making the CDN both responsive across a wide range of loads, and robust in the face of flash crowds and DDoS attacks. Specifically, we make the following three main contributions. First, we explore the design space of redirection strategies employed by the request redirectors, and define a class of new algorithms that carefully balance load, locality, and proximity. We demonstrate that these new algorithms are much more effective on improving CDN robustness when compared with published CDN technology. Second, we develop a novel hybrid network-server simulator that provides detailed modeling at both the packet level and at the operating systems level, and use it to conduct extensive large-scale simulations to evaluate various redirection strategies. Third, we build a prototype CDN, called CoDeeN [53] on PlanetLab [61]. We gain valuable experience on managing and monitoring this operational CDN and use it for ongoing research. Partial and preliminary results of this research have appeared in our OSDI'02 paper [81]. These contributions are not limited to just CDNs, and the algorithms and techniques developed in this research can potentially be applied to other systems such as distributed storage systems and peer-to-peer systems.

1.5.1 Redirection Strategies

The main research challenge we have identified is to find the appropriate tradeoffs that request redirection strategies need to make. Before proposing new schemes, we first evaluate the benefits and drawbacks of existing redirection strategies. We pick a baseline strategy, Random, which has no pathological behavior. Next, we approximate some well-known commercial CDN redirection strategies based on the published information, including their best load balancing behavior. These redirection strategies rely on a fixed number of server surrogates to serve each URL and share the request load. We then propose a class of new strategies that carefully balance load, locality, and proximity. Our new schemes dynamically change the number of surrogates for each URL based on the current system load status and server cache locality.

We use large-scale detailed simulations to evaluate the various strategies. These simulations clearly demonstrate the effectiveness of our new algorithms: they produce a 60-91% improvement in system capacity when compared with published information about commercial CDN technology, user-perceived response latency remains low, and the system scales well with the number of servers. We also vary the patterns of request traffic, for example, to create different hot spot situations. Again, using our new schemes, CDN systems show better flexibility and adaptiveness under different traffic. These new redirection schemes not only greatly improve the performance of CDNs under normal operations, but also let the system absorb flash crowds and mitigate the damages of DoS attacks more easily and naturally.

In addition, we evaluate several network proximity based redirection strategies. These strategies explicitly factor network proximity information into redirection strategies in an effort to direct more load to closer surrogates. Moving more requests to nearby servers certainly improves response latency through shortening the network paths that requests and responses have to traverse, but it could also have adverse impacts on system throughput due to load imbalance. In current CDNs, network proximity is usually handled through dividing the Internet into several geographical or topological regions.

Our network proximity based schemes can certainly take advantage of this hierarchical approach, but here we are more focused on addressing the intra-region proximity issue on CDNs, which we believe is the first attempt in this light.

We explore the design space of request redirection strategies in Chapter 2 by providing some background knowledge and categorizing different redirection strategies into several groups according to how they handle different factors, such as load and locality. In Chapter 4, we present the performance results of different redirection strategies using extensive large-scale simulations.

1.5.2 Hybrid Simulator

Evaluating various CDN redirection strategies at a large scale on the Internet is extremely difficult, especially when we care more about overload situations, such as flash crowds and DoS attacks. To artificially create these situations is not practical, and becomes very disruptive to others. Therefore, simulation is the only viable option. However, to faithfully simulate CDN systems under overload traffic, we need to be careful about details at every aspect of the system, both at the network layer and at the operating system/server application layer. At the time we started this work, there were no simulators that provided detailed modeling at both layers. We decided to take advantage of existing simulators that individually model either networks or OS/servers well, and combine the best of them. Specifically, we merged the NS-2 and LogSim simulators, combining them into a hybrid simulator that provides detailed modeling at both the packet level and the operating system level. We also optimized several modules of the hybrid simulator, allowing us to carry out simulations at a large scale. Currently, we can simulate a CDN system with over 1,000 clients and 128 servers, and the sustainable request rate can reach to more than 70,000 requests per second, which hits the limit of our current simulation environment. The simulator proves to be very useful in our evaluations and its detailed accounting helps us understand why our new redirection schemes work better.

Chapter 3 presents details on how we construct our hybrid simulator, and describes the network topology and traffic models we use in our simulations.

1.5.3 CDN Deployment

Recent developments of Internet-scale open testbed/emulation environments [61, 80, 84] provide us a valuable opportunity to build an operational CDN under realistic constraints. In particular, PlanetLab [61] is an overlay network that spans multiple universities, research institutions and industrial companies. It is designed to let researchers develop, deploy and access new, sometimes disruptive, Internet-scale services. Currently, it has been deployed on more than 65 sites, distributed across North America, Europe, Asia and Australia. We built a prototype CDN, named *CoDeeN* [53], atop PlanetLab, using a commercial proxy server. CoDeeN is now successfully running on more than 40 PlanetLab sites. It follows an opt-in service model, where users point their browsers to one of the CoDeeN proxies to take advantage of CoDeeN. CoDeeN allows us to have first hand experience with CDN management, performance tuning and monitoring. It also allows us to collect traffic patterns and learn how to auto-configure CDN servers, among other interesting research issues. We plan to use CoDeeN to further verify our simulation results about different redirection schemes.

We report our progress on deploying and managing CoDeeN in Chapter 5.

Chapter 2

Request Redirection

Request redirectors play a crucial role on CDNs, not just because they map a client request to an appropriate server while largely keeping it transparent to the client, but more importantly, because how they select servers affect how CDN resources are allocated. A well-designed redirection strategy can efficiently utilize all the resources on CDNs and make the system resilient across a wide range of loads. In contrast, a strategy with a poor resource management policy would render the CDN system susceptible to abnormal traffic or attacks.

This chapter explores the design space of CDN request redirection strategies. To be more specific, we want to understand the benefits and drawbacks of existing strategies and find the right tradeoffs we should make to improve current strategies. Before delving into the details of each individual strategy, we first discuss some building blocks of request redirectors. We describe several redirector implementation schemes commonly used by CDNs, and also discuss a few hashing schemes that serve as the substrates for some of the strategies we later investigate. With this background, we then study various strategies and group them into different categories based on how they trade off different factors such as server load, cache locality and network proximity. Most importantly, we present our new unified redirection strategies that carefully balance these factors and improve CDN robustness under both normal workload and flash crowds or attacks.

2.1 Building Blocks

As outlined in Section 1.3.2, we assume a general model of a CDN that consists of two major components: server surrogates and request redirectors. We also assume any of the server surrogates can serve any request on behalf of the origin servers. Where to place these surrogates and how to keep their contents up-to-date have been addressed by other CDN research [1, 26, 50], and are not our focuses. Here, we make no particular assumptions about servers' locations. *Request redirectors* are middleware entities that forward client requests to appropriate servers based on one of the strategies described in the next section. These redirection strategies are the main focus of this research. To help understand these strategies, this section first outlines various mechanisms that could be employed to implement redirectors, and then presents a set of hashing schemes that are at the heart of redirection.

2.1.1 Redirector Mechanisms

Several mechanisms can be used to redirect requests [5], including augmented DNS servers, HTTP-based redirects, and smart intermediaries such as routers or proxies. We describe these three redirector implementation schemes as follows.

DNS-Based Scheme

A popular redirection mechanism used by current CDNs is to augment DNS (Domain Name System) servers to return different server IP addresses to clients. DNS can be thought of as the “411” service on the Internet. Each Internet domain usually has its own authoritative DNS server(s) that collectively provide a ubiquitous name directory service on the Internet. Upon receiving a name resolution request, a DNS server returns either an A (address), NS (name server), or CNAME (alias) resource record (RR) to the client [51, 62].

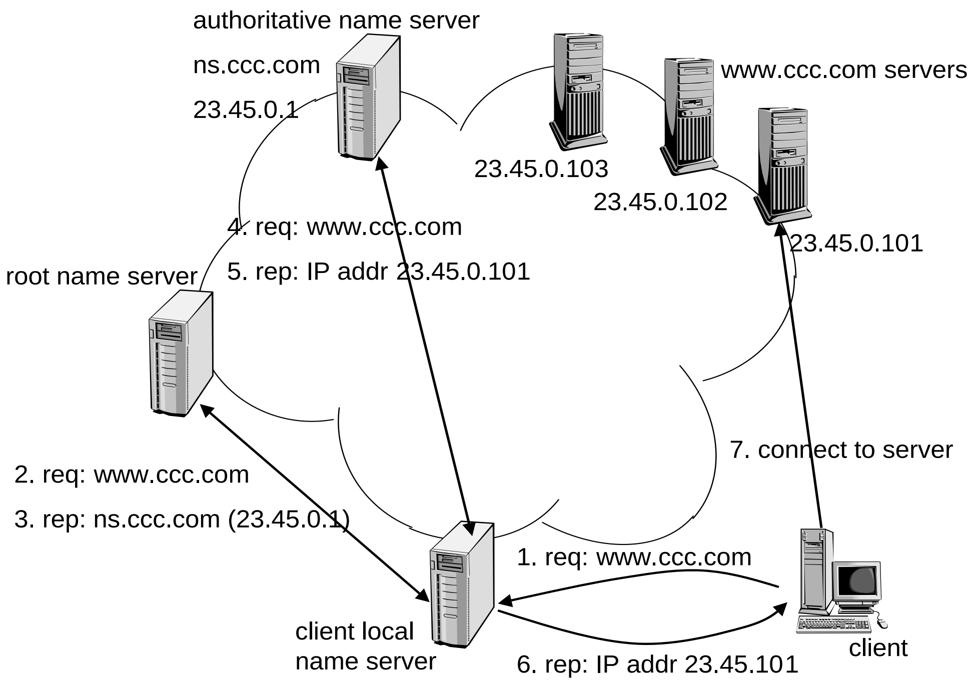

Figure 2.1: Typical DNS operation: a client resolves the address of www.ccc.com

Figure 2.1 shows an example of how a client resolves the IP address of www.ccc.com. Typically, the client application makes a recursive query to its local name server (1 in the graph). The local name server then iteratively tries to resolve the name (www.ccc.com). The local name server first contacts the root name server (2 in the graph). However, since the domain ccc.com has been delegated, the root name server returns the IP address of the authoritative name server of ccc.com domain—ns.ccc.com to the local name server (3 in the graph). Then the client side local name server contacts ns.ccc.com and gets the IP address of www.ccc.com (4 and 5) and returns this address to the client (6 in the graph). And now, the client can use the IP address to connect to the server (7 in the graph).

An augmented service-side authoritative DNS server can perform request redirection functionality. It can send a single reply with the surrogate IP address that it believes is best for the client. Alternatively, it can return multiple IP addresses of surrogates to clients allowing the client side name server to consume these multiple addresses in a round-robin fashion. For scalability reasons, current CDNs usually deploy a DNS hierarchy. The high-level DNS server first points the name query to a regional-level DNS server through NS redirection or CNAME redirection. Then the regional DNS server replies with the actual server address. To alleviate the look-up load on authoritative DNS servers, the RR records returned are usually cached on client-side name servers and these cached entries are valid for a time-to-live (TTL) period. A longer TTL can further reduce the traffic to authoritative DNS servers. On the other hand, in order to adapt quickly to network congestion or server outages, the augmented authoritative DNS servers have to set the TTL of RRs to be very short, in a hope to invalidate client-side caching when it is needed. For example, regional-level DNS servers can set the TTL to be very short such as 20 seconds, while high-level DNS servers can set it longer such as 20 minutes.

One limitation of the DNS-based schemes is that short TTLs in return could increase the load on DNS servers. Meanwhile, client-side name servers may not strictly respect this TTL, but keep using the cached RR even after its TTL expires. So the effectiveness of this shortened TTL may be limited by the lack of enforcement.

Another limitation of the DNS-based approaches lies in that it can only change the name mappings at a per-site level. Ideally, request redirection should be done at a per-object (URL) level. To achieve a finer granularity mapping and thus spread load more evenly, URL rewriting is often used in addition to augmented DNS servers. URL rewriting changes embedded objects in a page to point to different servers. This rewriting could be done either *a priori* or on-demand, and those modified URLs can still be mapped by augmented DNS servers.

HTTP Redirect

Servers can perform request redirection functionality themselves through HTTP [31] “redirect” responses. In the header of a response returned from a web server, there is a field containing a status code, using three digits, such as “200 OK”. HTTP has a class of status code called “Redirection 3xx”, which indicates that further actions need to be taken by the user agent (normally a browser) to fulfill the request. The code often used in redirection is “302 Found”, which means the requested resource resides temporarily under a different URI. So, when a server receives a client request, it can respond with “302 Found” and a new (possibly better) server that the client should contact for the page.

This approach is relatively simple to implement. However, it incurs an additional round-trip time across the Internet, and leaves the servers vulnerable to overload by the redirection task itself. Server bandwidth is also consumed by this process.

Intermediaries

The redirection function can also be distributed across intermediate nodes of the network, such as routers or proxies. These redirectors either rewrite the outbound requests, or send HTTP redirect messages back to the client. This approach allows for object (URL) level redirections, which helps to avoid congestion and balance server load at a fine granularity. If the client is not using explicit (forward mode) proxying, then the redirectors must be placed at choke points to ensure traffic in both forward and reverse directions is handled. The former is called a classical proxy, while the latter is called a transparent proxy [17]. Placing proxies closer to the edge yields well-confined easily identifiable client populations, which makes it easy to gauge the traffic each proxy handles and adjust the number of proxies to scale with the size of client population. The HTTP-redirect solution could also incur less overhead since the extra RTT between the proxy and the client is not likely to be significant. On the other hand, moving proxies closer to the server can result in more accurate feedback and load information. These proxy-based redirectors thus behave like overlay nodes and make an application-level request routing decision based on a URL and their knowledge on the load information of server surrogates and the network. Redirectors and surrogates thereby form an overlay network atop the Internet.

There are also other variations to implement request redirection on CDNs, but we cannot enumerate all of them here. To allow us to focus on redirection strategies and to reduce the complexity of considering the various combinations outlined in this section, we make the following assumptions: redirectors are located at the edge of a client site, they receive the full list of server surrogates through DNS or some other out-of-band communication, they rewrite outbound requests to pick the appropriate server, and they passively learn approximate server load information by observing client communications. We do not rely on any centralization, and all redirectors operate independently. Our simulations show that these assumptions—in particular, the imperfect information about server load—do not have a significant impact on the results.

2.1.2 Hashing Schemes

Our geographically dispersed redirectors cannot easily adapt the request routing schemes suited for more tightly-coupled LAN environments [37, 58], since the latter can easily obtain instantaneous state about the entire system. Instead, we construct strategies that use hashing to deterministically map URLs into a small range of values. The main benefit of this approach is that it eliminates inter-redirector communication since the same output is produced regardless of which redirector receives the URL. The second benefit is that the range of resulting hash values can be controlled, trading precision for the amount of memory used by bookkeeping.

The choice of which hashing style to use is one component of the design space, and is somewhat flexible. The various hashing schemes have some impact on computational time and request reassignment behavior on node failure/overload. However, as we discuss later, the computational requirements of the various schemes can be reduced by caching.

Modulo Hashing — In this "classic" approach, the URL is hashed to a number modulo the number of servers. While this approach is computationally efficient, it is unsuitable because the modulus changes when the server set changes, causing most documents to change server assignments. While we do not expect frequent changes in the set of servers, the fact that the addition of new servers into the set will cause massive reassignment is undesirable.

Consistent Hashing [43, 44] — In this approach, the URL is hashed to a number in a large, circular space, as are the names of the servers. The URL is assigned to the server that lies closest on the circle to its hash value.

Figure 2.2 illustrates how consistent hashing works. As shown, each server IP address is hashed to a value such as svr0, svr1..., and every URL also gets hashed to a value, *url1*, *url2* ... All these hash values are later converted to the range of [0,1]. Then a *unit circle*, a circle with unit circumference, is used to lay out these normalized hash values in a circular space. The server that a URL is assigned to is now obtained through finding the server whose hash point is first encountered by moving clockwise from the URL's point on the unit circle. In Figure 2.2, *url1* is assigned to svr2, *url2* and *url4* are assigned to svr3 and *url3* is assigned to svr0.

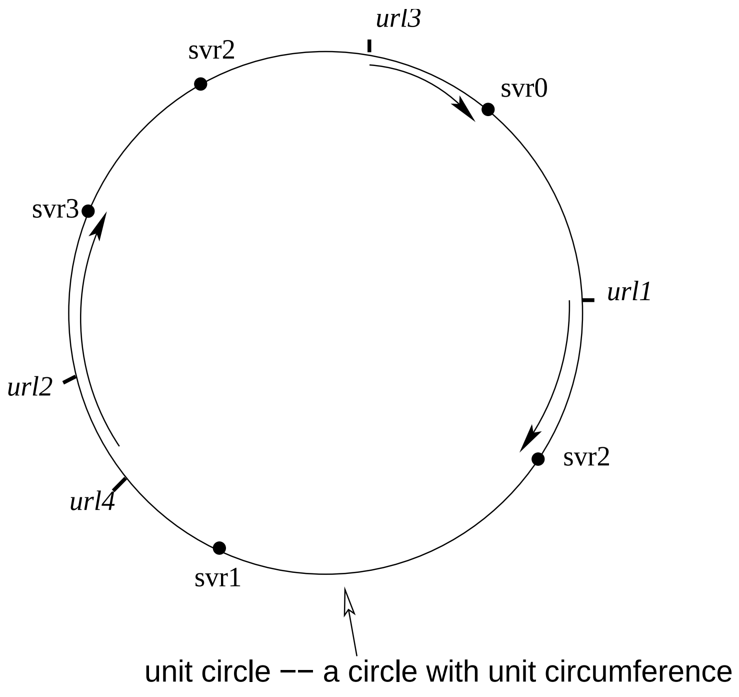

Figure 2.2: Consistent Hashing

There is a detailed technical analysis of Consistent Hashing in [44], which theoretically bounds its properties on load, balancing and computation cost. Intuitively, the hash function will randomly and uniformly map both URLs and servers to points onto the unit circle. As a result, each server is expected to get a fair share of all the URLs. When a server node fails in this scheme, its load shifts to its "neighbor" servers on the unit circle. When a new node is added, it only "steals" URLs that are closer to its own hash point from other servers, while other URLs whose points are far away are left unaffected. Thus, the addition or removal of a server only causes local changes in request assignments. Most of the URLs stick to their assigned servers, resulting in good server cache locality and a high hit rate. Even when a redirector only knows a subset of all the servers, this scheme will not cause too much discrepancy among different redirectors on deciding which server to be responsible for an URL. That is because only a few servers are close to that URL on the unit circle. In practice, to achieve a uniform distribution of URLs to servers, it is necessary to make several copies of each server to different "random" points on the unit circle. The base hash function that maps URLs and servers to the unit circle is also crucial for a uniform distribution, and it turns out that a random universal hash function [13] is a good candidate. Since this hashing scheme requires finding the closest match, a search tree can be used to reduce the search to logarithmic time. Alternatively, breaking the circle into equal length intervals and putting server points into "bins" according to intervals can further improve this match operation to constant time. Conceivably, to map a URL to multiple servers, one possible solution is to find the *k* closest servers on the unit circle. Consistent Hashing has been applied in commercial CDNs and some peer-to-peer systems [76] as well.

Highest Random Weight [78] — This approach, is the basis for Cache Array Routing Protocol (CARP) [19], and consists of generating a list of hash values by hashing each

URL with each server's name and sorting the results. These sorted hash values are called server *weights* for this URL. Each URL then has a deterministic order to access the set of servers. This weight list is traversed to find a server for the URL, and typically the top-most server from the list (the one with the highest weight) is selected. Similarly, for load balancing purposes, a URL can be mapped to more than one server, such as k servers, where the first k servers on the sorted list can be used.

A detailed analysis of Highest Random Weight (HRW) hashing is presented in [78]. In HRW hashing, server addition or removal only affects a small number of URLs, since only those URLs whose highest ranked server changes have to be reassigned. This approach requires more computation than Consistent Hashing, since generating the ordered list takes $O(NlogN)$ time, where N is the number of servers. However, it has the benefit that each URL has a different server order, so a server failure results in the remaining servers evenly sharing the load. To reduce computation cost, the sorted server lists can be cached using $O(N \times num\_url\_hashes)$ space, and even this can be reduced by keeping only the top few entries for each URL hash value.

2.2 Redirection Strategies

This section explores the design space for the request redirection strategies. As a quick reference, we summarize the properties of the different redirection algorithms in Table 2.1, where the strategies are categorized based on how they address locality, load and proximity.

The first category, Random, contains a single strategy, and is used primarily as a baseline. We then discuss four static algorithms, in which each URL is mapped onto a fixed set of server replicas—the Static category includes two schemes based on the best-

Category	Strategy	Hashing Scheme	Dynamic Server Set	Load Aware
Random	Random			No
Static	R-CHash	CHash	No	No
	R-HRW	HRW	No	No
Static +Load	LR-CHash	CHash	No	Yes
	LR-HRW	HRW	No	Yes
Dynamic	CDR	HRW	Yes	Yes
	FDR	HRW	Yes	Yes
	FDR-Global	HRW	Yes	Yes
Network Proximity	NPR-CHash	CHash	No	No
	NPLR-CHash	CHash	No	Yes
	NP-FDR	HRW	Yes	Yes

Table 2.1: Properties of Request Redirection Strategies

known published algorithms, and the Static+Load category contains two variants that are aware of each replica's load. The four algorithms in these two static categories pay more attention to locality than Random. Next, we introduce two new algorithms—denoted **CDR** and **FDR**—that factor both load and locality into their decision with each URL mapped onto a dynamic set of server replicas. We call this the Dynamic category. Finally, we factor network proximity into the equation, and present another new algorithm—denoted **NP-FDR**—that considers all aspects of network proximity, server locality, and load.

2.2.1 Random

In the random policy, each request is randomly sent to one of the server surrogates. We use this scheme as a baseline to determine a reasonable level of performance, since we expect the approach to scale with the number of servers and not to exhibit any pathological behavior due to patterns in the assignment. It has the drawback that adding more servers does not reduce the working set of each server. Since serving requests from main memory is faster than disk access, this approach is at a disadvantage versus schemes that exploit URL locality.

2.2.2 Static Server Set

We now consider a set of strategies that assign a fixed number of server replicas to each URL. This has the effect of improving locality over the Random strategy, but still sharing the load for a URL among several servers.

Replicated Consistent Hashing

In the Replicated Consistent Hashing (**R-CHash**) strategy, each URL is assigned to a set of replicated servers. The number of replicas is fixed, but configurable. Following the Consistent Hashing scheme, both the URL and the addresses of all servers are hashed to a value in the circular space (unit circle), and instead of only choosing the single closest server to the URL, the replicas are evenly spaced starting from this original point. On each request, the redirector randomly assigns the request to one of the replicas for the URL.

This strategy is intended to model the mechanism published about commercial CDNs, and is virtually identical to the scheme described in [43] and [44] with the network treated as a single geographic region. The scheme described in these two papers also includes a mechanism to use coarse-grained load balancing via virtual server names. When server overload is detected, the corresponding content is replicated across all servers in the region, and the degree of replication shrinks over time. However, the schemes are not described in enough detail to replicate.

Replicated Highest Random Weight

The Replicated Highest Random Weight (**R-HRW**) strategy is the counterpart to R-CHash, but with a different hashing scheme used to determine the replicas. To the best of our knowledge, this approach is not used in any existing content distribution network. In this approach, the set of replicas for each URL is determined by using the top *k* servers from the ordered list generated by Highest Random Weight hashing. And on each request, the redirector randomly picks one member of the appropriate set and sends the request to that server. Compared to R-CHash, this scheme is less likely to generate the same set of replicas for two different URLs. As a result, less-popular URLs that may have some overlapping servers with popular URLs are also likely to have some other less-loaded nodes in their replica sets.

2.2.3 Load-Aware Static Server Set

The Static Server Set schemes randomly distribute requests across a set of replicas, which shares the load but without any active monitoring. We extend these schemes by introducing load-aware variants of these approaches. To perform fine-grained load balancing, these load-aware schemes maintain local estimates of server load at the redirectors, and use this information to pick the least-loaded member of the server set. A possible load measure is the number of outstanding connections currently open to each server surrogate. The load-balanced variant of R-CHash is called **LR-CHash**, while the counterpart for R-HRW is called **LR-HRW**.

2.2.4 Dynamic Server Set

We now consider a new category of algorithms that dynamically adjust the number of replicas used for each URL in an attempt to maintain both good server locality and load balancing. By reducing unnecessary replication, the working set of each server is reduced, resulting in better file system caching behavior.

Coarse Dynamic Replication

Coarse Dynamic Replication (**CDR**) adjusts the number of replicas used by redirectors in response to server load and demand for each URL. Like R-HRW, CDR uses HRW hashing to generate an ordered list of servers. Rather than using a fixed number of replicas, however, the request target is chosen using coarse-grained server load information to select the first "available" server on the list.

Figure 2.3 shows how a request redirector picks the destination server for each request. This decision process is done at each redirector independently, using the load status of the possible servers. Instead of relying on heavy communications between servers and request redirectors to get server load status, we use local load information observed by each redirector as an approximation. We currently use the number of active connections to infer the load level, but we can also combine this information with response latency, bandwidth consumption, etc.

As the load increases, this scheme changes from using only the first server on the sorted list to spreading requests across several servers. Some documents normally handled by "busy" servers will also start being handled by less busy servers. Since this process is based on aggregate server load rather than the popularity of individual documents, servers hosting some popular documents may find more servers sharing their load

```
find_server($url$, $S$) {
    foreach server $s_i$ in server set $S$,
        $weight_i$ = hash($url$, address($s_i$));
    sort $weight$;
    foreach server $s_j$ in decreasing order of $weight_j$ {
        if $satisfy\_load\_criteria(s_j)$ then {
            $targetServer \leftarrow s_j$;
            stop search;
        }
    }
    if $targetServer$ is not valid then
        $targetServer \leftarrow$ server with highest weight;
    route request $url$ to $targetServer$;
}
```

Figure 2.3: Coarse Dynamic Replication

than servers hosting collectively unpopular documents. In the process, some unpopular documents will be replicated in the system simply because they happen to be primarily hosted on busy servers. At the same time, if some documents become extremely popular, it is conceivable that all of the servers in the system could be responsible for serving them.

Fine Dynamic Replication

A second dynamic algorithm—Fine Dynamic Replication (**FDR**)—addresses the problem of unnecessary replication in Coarse Dynamic Replication (CDR) by keeping information on URL popularity and using it to more precisely adjust the number of replicas. By controlling the replication process, the per-server working sets should be reduced, leading to better server locality, and thus, better response time and throughput. This scheme uses HRW hashing as in CDR, but additionally, each URL is also hashed to a smaller identifier, in the range of thousands to millions of values. With each of these identifiers is kept the number of servers to use in the sorted list, allowing URLs to be replicated across a number of servers based on their popularity.

Unpopular URLs could be unnecessarily replicated in CDR, simply because they share their primary servers with some popular URLs. The introduction of finer-grained bookkeeping is an attempt to counter the possibility of a “ripple effect” in CDR, which could gradually reduce the system to round robin under heavy load. In this scenario, a very popular URL causes its primary server to become overloaded, causing extra load on other machines. Those machines, in turn, also become overloaded, causing documents destined for them to be served by their secondary servers. Under heavy load, it is conceivable that this displacement process ripples through the system, reducing or eliminating the intended locality benefits of this approach.

To reduce extra replication, FDR keeps an auxiliary structure at each redirector that maps each URL to a “walk length,” indicating how many servers in the HRW list should be used for this URL. Using a minimum walk length of one provides minimal replication for most URLs, while using a higher minimum will always distribute URLs over multiple servers. When the redirector receives a request, it uses the current walk length for the URL and picks the least-loaded server from the current set. If even this server is busy, the walk length is increased and the least-loaded server is used.

This approach tries to keep popular URLs from overloading servers and displacing less-popular objects in the process. The size of the auxiliary structure is capped by hashing the URL into a range in the thousands to millions. Hash collisions may cause some URLs to have their replication policies affected by popular URLs. As long as the number of hash values exceeds the number of servers, the granularity will be significantly better than the Coarse Dynamic Replication approach. The redirector logic for this approach is shown in Figure 2.4. To handle URLs that become less popular over time,

```
find_server($url$, $S$) {
  walk_entry $\leftarrow$ walkLenHash($url$);
  $w\_len$ $\leftarrow$ walk_entry.length;
  foreach server $s_i$ in server set $S$,
     $weight_i$ = hash($url$, address($s_i$));
  sort $weight$;
  $s_{candidate}$ $\leftarrow$ least-loaded server of top $w\_len$ servers;
  if $satisfy\_load\_criteria(s_{candidate})$ then {
    $targetServer$ $\leftarrow$ $s_{candidate}$;
    if ($w\_len > 1$ &&
        timenow() $-$ walk_entry.lastUpd $> chgThresh$)
      walk_entry.length $--$;
  } else {
     foreach rest server $s_j$ in decreasing weight order {
        if $satisfy\_load\_criteria(s_j)$ then {
          $targetServer$ $\leftarrow$ $s_j$;
          stop search;
        }
     }
     walk_entry.length $\leftarrow$ actual search steps;
  }
  if walk_entry.length changed then
     walk_entry.lastUpd $\leftarrow$ timenow();
  if $targetServer$ is not valid then
     $targetServer$ $\leftarrow$ server with highest weight;
  route request $url$ to $targetServer$;
}
```

Figure 2.4: Fine Dynamic Replication

with each walk length, we also keep the time of its last modification. We decrease the walk length if it has not changed in some period of time.

As a final note, both dynamic replication approaches require some information about server load, specifically how many outstanding requests can be sent to a server by a redirector before the redirector believes it is busy. We currently allow the redirectors to have 300 outstanding requests per server, at which point the redirector locally decides the server is busy. It would also be possible to calibrate these values using both local and global information—using its own request traffic, the redirector can adjust its view of what constitutes heavy load, and it can perform opportunistic communication with other redirectors to see what sort of collective loads are being generated. The count of outstanding requests already has some feedback, in the sense that if a server becomes slow due to its resources (CPU, disk, bandwidth, etc.) being stressed, it will respond more slowly, increasing the number of outstanding connections. To account for the inaccuracy of local approximation of server load at each redirector, in our evaluations, we also include a reference strategy, **FDR-Global**, where all redirectors have perfect knowledge of the load at all servers.

```
satisfy_load_criteria(server) {
   if (server's load > low_thresh
       && exists server l with load < low_thresh) ,
      return true;
   if (server's load > 2 × high_thresh)
      return true;
   return false;
}
```

Figure 2.5: Example Load Evaluation

Figure 2.5 gives an example of load evaluation functions, where the *low_thresh* and *high_thresh* are configurable parameters. In the case of using the number of outstanding requests as the load measure, *low_thresh* and *high_thresh* can be set to 50 and 150 respectively.

Conceivably, Consistent Hashing could also be used to implement CDR and FDR. We tested CHash-based CDR and FDR schemes, but they suffer from the "ripple effect" and sometimes yield even worse performance than load-aware static replication schemes. Part of the reason is that in Consistent Hashing, since servers are mapped onto a circular space, the relative order of servers for each URL will be *effectively* the same. This means the load migration will take a uniform pattern; and the less-popular URLs that may have overlapping servers with popular URLs are unlikely to have some other less-loaded nodes in their replica sets. Therefore, we will mainly present CDR and FDR based on HRW. And we will show our test results of CHash-based CDR and FDR in Section 4.4.3.

2.2.5 Network Proximity

Many commercial CDNs start server selection with network proximity matching. For instance, [43] indicates that CDN's hierarchical authoritative DNS servers can map a client's (actually its local DNS server's) IP address to a geographic region within a particular network and then combine it with network and server load information to select a server separately within each region. Other research [41] shows that in practice, CDNs succeed not by always choosing the "optimal" server, but by avoiding notably bad servers.

For the sake of studying system capacity, we make a conservative simplification by treating the entire network topology as a single geographic region. We could also simply take the hierarchical region approach as in [43], however, to see the effect of *integrating* proximity into server selection, we introduce three strategies that explicitly factor intra-region network proximity into the decision. Our redirector measures servers' geographical/topological location information through *ping*, *traceroute* or similar mechanisms and uses this information to calculate an "effective load" when choosing servers.

$$\begin{cases} min\_distance &= MIN\{distance_i, i = 1...n\} \\ std\_distance_i &= distance_i / min\_distance \\ effect\_load_i &= load_i \times std\_distance_i \end{cases} \tag{2.1}$$

To calculate the *effective load*, redirectors multiply the raw load metric with a normalized *standard distance* between the redirector and the server. Redirectors gather distances to servers using round trip time (RTT), routing hops, or similar information. These raw distances are normalized by dividing by the minimum locally observed distance, yielding the standard distance. In our simulations, we use RTT for calculating raw distances. Equation group 2.1 depicts the calculation of effective load.

FDR with Network Proximity (NP-FDR) is the counterpart of FDR, but it uses effective load rather than raw load. Similarly, **NPLR-CHash** is the proximity-aware version of LR-CHash. The third strategy, **NPR-CHash**, adds network proximity to the load-oblivious R-CHash approach by assigning requests such that each surrogate in the fixed-size server set of a URL will get a share of total requests for that URL inversely proportional to the surrogate's distance from the redirector. As a result, closer servers in the set get a larger share of the load.

The use of effective load biases server selection in favor of closer servers when raw load values are comparable. For example, in standard FDR, raw load values reflect the fact that distant servers generate replies more slowly, so some implicit biasing exists. However, by explicitly factoring in proximity, NP-FDR attempts to reduce global resource consumption by favoring shorter network journeys.

Although we currently calculate effective load this way, other options exist. For example, effective load can take other dynamic load/proximity metrics into account, such as network congestion status through real time measurement, thereby reflecting instantaneous load conditions.

Chapter 3

Evaluation Methodology

The main goal of this work is to examine how the strategies discussed in Chapter 2 respond under different loads, and especially how robust they are in the face of flash crowds and other abnormal workloads that might be used for a DDoS attack. Attacks may take the form of legitimate traffic [60], making them difficult to distinguish from flash crowds.

Evaluating the various algorithms described in Section 2.2 on the Internet is not practical, both due to the scale of the experiment required and the impact a flash crowd or attack is likely to have on regular users. Simulations are clearly the only option, since they provide us with easier controls in a confined environment. Unfortunately, there has not been (up to the point we started) a simulator that considers both network traffic and server load. Existing simulators abstract different aspects of the system by making certain assumptions or simplifications, which could affect the accuracy of the results. They either focus on the network, assuming a constant processing cost at the server, or they accurately model server processing (including the cache replacement strategy), but use a static estimate for the network transfer time without considering possible congestion. In the situations that interest us, both the network and the server are important.

To remedy this situation, we develop a new simulator that combines network-level simulation with OS/server simulation. Specifically, we combine the NS-2 simulator with LogSim, allowing us to simulate network bottlenecks, round-trip delays, and OS/server performance. NS-2 [55] is a packet-level simulator that has been widely used to test TCP implementations. However, it does not simulate much server-side behavior. LogSim is a server cluster simulator used in previous research on LARD [58], and it provides detailed and accurate simulation of server CPU processing, memory usage, and disk access. This chapter describes how we combine these two simulators, and discusses how we configure the resulting simulator to study the algorithms presented in Section 2.2.

3.1 Simulator

A model of LogSim is shown in Figure 3.1. Each server node consists of a CPU and locally attached disk(s), with separate queues for each. At the same time, each server node maintains its own memory cache of a configurable size and replacement policy. Incoming requests are first put into the holding queue, and then moved to the active queue. The active queue models the parallelism of the server, such as the maximum number of processes or threads allowed on each server in multiple-process or multiple-thread server systems.

3.1.1 Overview of the Hybrid Simulator

Both NS-2 and LogSim are discrete event-driven simulators, and we show the way of combining them in Figure 3.2. We keep NS-2's event engine as the main event manager

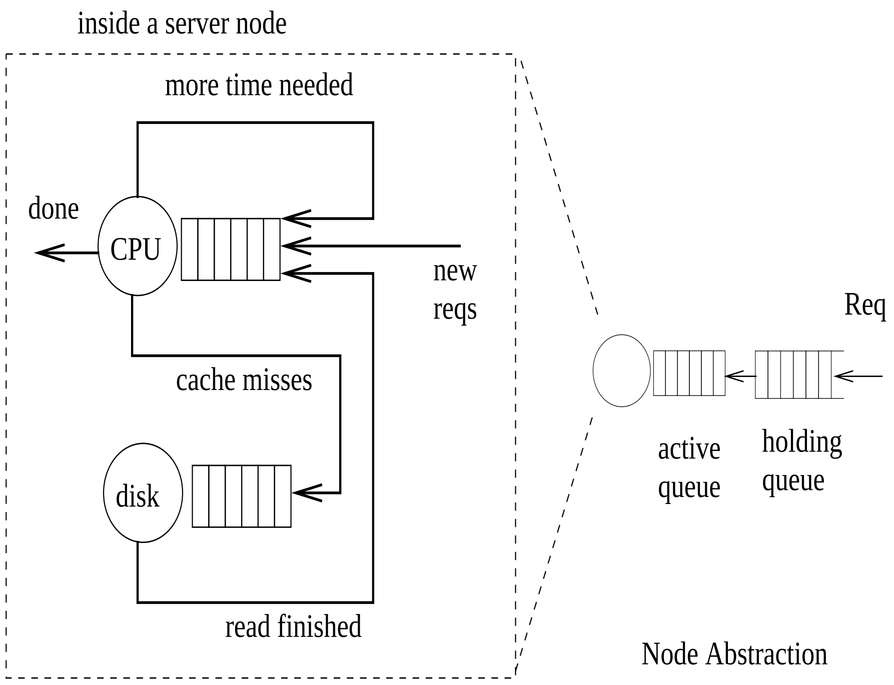

Figure 3.1: LogSim Simulator

for the hybrid simulator, wrap each LogSim event as an NS-2 event, and insert it into the NS-2 event queue. All the callback functions are kept unchanged in LogSim. We maintain a "server" module in both NS-2 and LogSim, and a pair of server modules, one from each simulator, are used to model the behaviors of a simulated server surrogate. The module in NS-2 takes care of communication, and the module in LogSim is responsible for request processing. When crossing the boundary between the two simulators, tokens (continuations) are used to carry side-specific information. These tokens are pointers to private data structures maintained respectively in NS-2 and LogSim. The processing state of corresponding server modules is kept in these data structures. For simplicity, clients are only modeled in NS-2, which are capable of setting up TCP connections with servers, sending requests and receiving responses. Redirectors are also simulated in NS-2, which can be configured with different redirection strategies. Since we assume a proxy or router based redirector implementation scheme, each redirector will be able to see the traffic originating from and destined to the clients in its local region. As a result, redirectors rely on this traffic information to approximate load status on each surrogate.

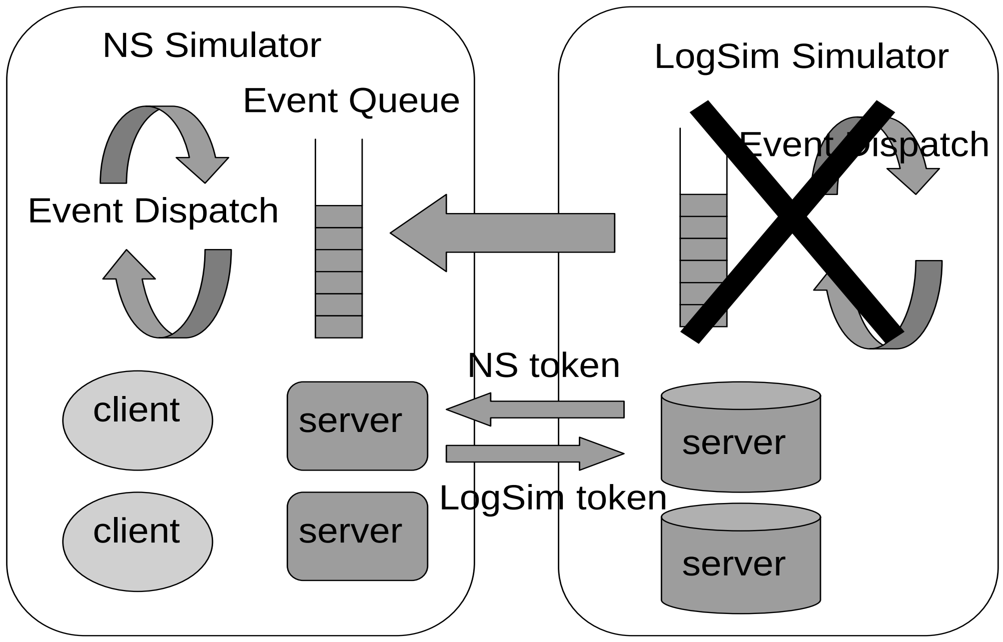

Figure 3.2: Merging NS-2 and LogSim Simulator

A typical request-response session looks as follows (we follow the HTTP/1.0 model). When a client wants to access a URL, the redirector near this client first chooses an appropriate server surrogate for this URL, and then the client directly opens a new connection to that server for the URL, mimicking instant packet rewriting at the redirector. The requested URL then gets transferred to the server through two-way TCP in NS-2. After the request fully arrives at the server module in NS-2, it is passed along with an NS-2 token to its counterpart server module in LogSim. Next, the LogSim server module processes the request. In LogSim, all the content exists on disks. The LogSim server module first tries to find the content of the URL in its memory cache. If it is a cache miss, the server has to read the content from its disk. Reading content from its disk to memory cache may trigger the cache replacement process if there is not enough memory space. After locating the content, the LogSim server module passes back the NS-2 token in addition to its own token to its NS-2 counterpart, asking the NS-2 server module to transfer the response to the client through the TCP connection. If the content has to be broken into chunks, this token passing (processing hand-over) may be repeated several times. Finally, when all the response data have been shipped to the client, per its LogSim counterpart's request, the NS-2 server module initiates the teardown of the TCP connection.

On the NS-2 side, all packets are stored and forwarded, as in a real network. We use two-way TCP at the transport layer, which includes detailed modeling of three-way handshake, disconnection, and TCP Reno's congestion control mechanism [55]. We currently use static routing within NS-2, although we may run simulations with dynamic routing in the future.

On the LogSim side, the costs for the basic request processing were derived by performing measurements on a 300MHz Pentium II machine running FreeBSD 2.2.5 and the Flash web server [57]. Although this machine is clearly slower than the cutting-edge high-end web servers, it makes it possible for us to simulate at a large scale. Connection establishment and teardown costs are set at 145μs, while transmit processing incurs 40μs per 512 bytes. Using these numbers, an 8-Kbyte document can be served from the main memory cache at a rate of approximately 1075 requests/sec. When disk access is needed, reading a file from the disk has a latency of 28ms. The disk transfer time is 410μs per 4 Kbytes. For files larger than 44 Kbytes, an additional 14ms is charged for every 44 Kbytes of file length in excess of 44 Kbytes. The replacement policy used on the servers is Greedy-Dual-Size (GDS)[11], as it appears to be the best-known policy for

Web workloads. 32MB memory is available for caching documents on each server and every server node has one disk. This server is intentionally slower than the current state-of-the-art (it is able to service approximately 600 requests per second), but this allows the simulation to scale to a larger number of nodes.

3.1.2 Optimizations

When we scaled up our simulations to many nodes, for example, 1000 clients and 128 servers, and a high request rate, currently up to more than 70,000 requests per second, the experiments became extremely slow, taking more than 70 hours wall-clock time. This greatly affected our ability to conduct multiple experiments and collect many data points. We instrumented the hybrid simulator to identify the problems. The LogSim part was quite efficient, and there was little room for further optimizations. On the NS-2 side, however, we found a couple of bottlenecks. To speed up the simulation time and increase the simulation scale, we re-implemented several NS-2 modules and performed other optimizations. Compared to the existing largest commercial CDNs with over 10,000 servers, our final simulated system (up to 128 servers) may still be small, and can be viewed as a model of servers within a geographical or administrative region. However, these simulations have already reached the limit of our current simulation environment. Below, we discuss our three main optimizations on the NS-2 side. We are still refining our current improvements and examining other possible ways to speed up the simulations.

NS-2 is an object-oriented simulator, and follows the idea of split-language programming [56]. It is implemented in C++ with an OTcl [83] (an object-oriented variant of Tcl) front end. NS-2 simulation configurations, such as topology setup and start/stop routines, are generally written in OTcl. This is because the OTcl script is executed through interpretation, which makes it easy to update settings without re-compiling the whole NS-2 package whenever some minor configuration changes are needed. On the other hand, the core processing logic is written in C++ for performance reasons. The NS-2 simulator supports a class hierarchy in C++ and a similar class hierarchy within the OTcl interpreter. From a user's perspective, there is a one-to-one correspondence between classes from these two hierarchies, respectively. In other words, entities, such as network links, TCP sources and sinks and so on, have a main object in C++ with a "shadow" object in OTcl. By doing this, these entities can be easily manipulated in OTcl. In fact, this shadowing or binding happens with an even finer granularity at the individual variable level. In this case, bi-directional bindings are established between member variables in C++ and corresponding member variables in OTcl. Changes of variables in one language are automatically propagated to their counterparts in the other language. However, cross-language updates are very expensive and can considerably slow down the simulations. For example, when creating a new TCP source, all its state variables, such as window sizes and timeout values, have to be initialized to their default values in OTcl and these initializations are propagated to C++ as well, resulting in a fairly large overhead. Keeping many variables in both C++ and OTcl also increases memory pressure of simulations. All of these present particular challenges to our experiments, where a large number of TCP connections are set up or torn down every second of simulated time.

NS-2 is not optimized for this type of experiments that require a very large memory footprint and CPU processing power. To alleviate these problems, we modify the two-way TCP modules of NS-2 and keep everything in C++. By doing this, each time a new two-way TCP agent is created, all the initializations will be done in C++ only, which saves processing in OTcl. Eliminating shadow variables in OTcl also reduces memory use, and any change of these TCP state variables in the future will also not cause any additional changes in OTcl (Of course, now we cannot directly manipulate these variables in OTcl). However, in the previous NS-2 implementation, attaching a TCP agent to or detaching it from a node is achieved in OTcl through classifier objects. To accommodate our pure C++ TCP agents with no corresponding OTcl shadow objects, we also need a pure C++ classifier. Having these two pieces (pure C++ TCP agents and classifiers) in place, opening a new TCP connection becomes less heavy-weight, and the memory requirement for a single two-way TCP agent is reduced from about 3.1KB to about 1.2KB. We are therefore able to support a very large number of simultaneous TCP connections, currently up to 160,000 concurrent connections at a given time.

Another limitation of the original NS-2 implementation is related to its port management and two-way TCP details. In NS-2, each transport agent, such as TCP or UDP, has to be bound to a port on a node. The port number is part of the demultiplex key used by classifiers on NS-2 nodes. This port-based scheme, to some extent, resembles the socket interface [75], but not entirely. When an agent is detached from a node, or more precisely from the port classifier of the node, the port number of this agent will then be put in a free port pool by clearing its bit in the port map of that node. The next time a new agent has to be attached to the same node's port classifier, NS-2 linearly searches the port map to find the first (smallest) available (free) port number. This agent attaching and detaching process is necessary since the port space is limited, and port numbers have to be recycled. Meanwhile, although we have optimized the two-way TCP implementation by eliminating shadowing OTcl TCP objects, each TCP agent still requires a considerable amount of memory, on the order of a thousand bytes. In large-scale simulations, to save memory, it is also very important to recycle ports and TCP agents as soon as possible. However, fast recycling stands at odds with TCP "dangling packets", especially the retransmitted FIN packets.

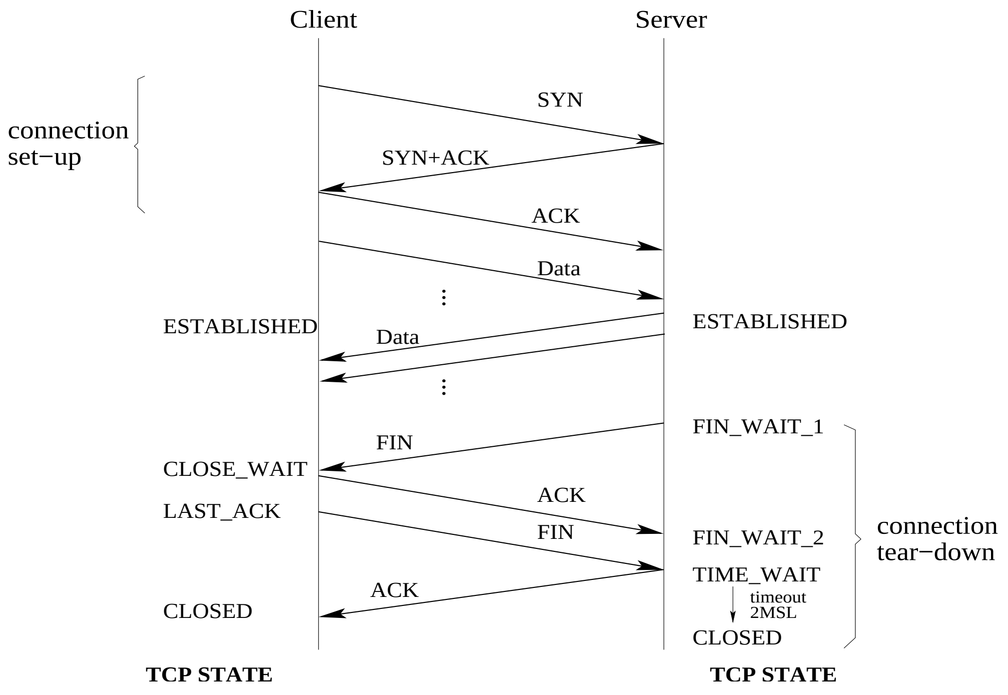

Figure 3.3: Time Line for TCP connection Set-up and Tear-down in CDN experiments

Figure 3.3 depicts a typical time line of TCP with its state, where the client initiates the connection and the server tears it down, as in the case of our CDN experiments. At the beginning, after the three-way handshake, the client sets up a connection with the server and both parties enter the ESTABLISHED state. Then the client sends the request and the server replies with the response in data packets. When the server finishes sending all the response data, it sends a FIN packet to the client to initiate the disconnection, and itself enters the FIN_WAIT_1 state. When the client gets the FIN packet, it acknowledges it and enters the CLOSE_WAIT state. This ACK packet causes the server-side TCP to enter the FIN_WAIT_2 state. Then the client-side TCP sends a FIN packet to the server-side TCP indicating it is ready to disconnect, and the client enters the LAST_ACK state waiting for the (last) ACK packet for its just-sent FIN packet. If everything goes well, after the client gets the last ACK from the server in its LAST_ACK state, it enters the CLOSED state. Now it is safe to recycle the port and TCP agent on the client side. However, if the ACK from the server does not come in time, the client-side TCP has to resend FIN. The server-side TCP, on receiving the FIN packet from the client, sends an ACK to this FIN, but has to remain in the TIME_WAIT state before moving to the CLOSED state. This is because it does not know if the final ACK has been successfully delivered to the client. The client may try retransmitting FIN packet as discussed above. During network congestion, the retransmitted FIN from the client may get delayed, and there could be even more retransmitted FINs from the client if ACKs (or FINs) keep getting lost. So if we allow the current port on the server to get recycled without waiting long enough in the TIME_WAIT state, a new connection that re-uses the same port number can get “dangling” FIN packets belonging to legacy connections. Even worse, since it is a (retransmitted) FIN packet, this new connection can be terminated unexpectedly. This is a well studied and documented situation in the TCP specification [79] and the wait period in the TIME_WAIT state is typically set as two times of the Maximum Segment Lifetime (2MSL), for example 120 seconds.

However, the TIME_WAIT state is missing in NS-2’s two-way TCP implementation. After just sending one ACK to the client’s FIN, the sever side TCP immediately enters the CLOSED state and this TCP agent and its port then get recycled. As a result, in our earlier experiments, we experienced quite a few unexpected disconnections on recent recycled ports due to the dangling FIN packets.

To solve this problem, one possible solution is to add the TIME_WAIT state into NS-2’s two way TCP implementation. However, waiting such a long time (120 seconds of simulated time) will force a lot of memory tied up to those TCP agents doing nothing but waiting. An alternative is to simply ignore the dangling FIN packets. However, doing this will cause the client side TCP to stay in the CLOSE_WAIT state for an indefinite time, since its FIN packet never gets ACKed.

To alleviate this dangling FIN packet problem and also strike a balance between recycling TCP agents fast and sticking to TCP specifications, we make the following adaptations. We leave the basic logic of two-way TCP of NS-2 unchanged, and after the server TCP sends its last ACK, it still enters the CLOSED state immediately. However, we replace this server TCP agent with a dummyACK agent at the same port when we detach the TCP agent. This dummyACK agent simply acknowledges any packet it receives by swapping the source and destination addresses and ports in the packet. In other words, this dummyACK agent can be attached to multiple nodes or to multiple ports on the same node simultaneously. It is only used to make sure client-side TCPs can get ACKs to their FINs. This approach consumes only a constant but small amount of memory; however, it achieves similar effects as if a full server-side TCP agent is waiting for the possibly retransmitted (dangling) FIN packets, and it reduces our simulations' memory footprint through releasing full TCP agents earlier. This spoofed TIME_WAIT state is intentionally kept shorter than 2MSL, allowing port numbers to be recycled faster. Now, when the client side TCP does not receive an ACK in the LAST_ACK state and resends FIN packet, for most of the time, it can still get ACKs from this dummyACK agent and move to its CLOSED state even if the original server side TCP gets recycled. The specification of TCP disconnection is therefore largely preserved. Since dummyACK waits shorter than 2MSL, it is possible that there are still dangling FIN packets under very heavy network congestion. As a last resort, we take advantage of the unique flow ID assigned to each TCP connection, which is originally used for bookkeeping purposes. A TCP agent will then ignore any packet that is not bearing the same flow ID as determined at the connection setup time. This eventually avoids the unexpected disconnection problem.

And we also force to recycle client side TCPs when they wait for their last ACKs for too long.

In summary, we try our best to keep our simulations as realistic as possible, but at the same time, still shoot for a less resource consumption. Since this optimization facilitates fast recycling of TCP agents, it also contributes to sustaining the large number of concurrent TCP connections discussed earlier.

A third major bottleneck of NS-2 lies in its event management. Since we use NS-2's event dispatcher as the main event engine for the hybrid simulator, its performance determines how fast we can move the events along. The default NS-2 event dispatcher uses a calendar queue [9, 39], which has an asymptotic O(1) time for event insertion/deletion operations. Calendar queue schedulers perform well when they have many buckets and each bucket (of time) has a relatively small number of events. However, in our CDN experiments, within each second, even millisecond of simulated time, there are many events. This is because we have a large number of simultaneously active connections, thus a large number of packets from these connections. Each packet becomes an event in NS-2. (In fact, in a typical run of our simulations for 600 seconds of simulated time, the total number of events generated already far exceeds four billion, and we have to use 64-bit event IDs.) So, too many events will be mapped into one bucket, and many of the event operations will result in a linear search in some bucket. As a result, the calendar queue scheduler with default settings does not run very fast. Although we could adapt the number of buckets in a calendar queue for possible speed up, it has to be done on a case-by-case base. In fact, empirically in many cases, without tuning any parameter, the calendar queue scheduler does not provide better event dispatching performance for our simulations than some other schedulers based on different approaches, for example, a heap [21] scheduler can be 20% faster.

We choose to use a heap-based event dispatcher instead. However, the heap implementation in NS-2 suffers on event deletion operations. This is because given the pointer to an event for deletion, the heap-based event manager in NS-2 needs to first find the index of this event through a linear search of its entire array that stores the heap, which is an O(N) operation. Only after finding the index, the heap can delete the element and perform a “heapify” operation to restore heap properties. As pointed out before, at any given simulated time point, there could be a large number of simultaneous events, so the heap array itself can be quite large and linear searches of this array will never be fast. These event deletion operations thus can become very slow. On the other hand, from time to time, TCP agents have to cancel timeout timers, which are translated to deleting the corresponding timer events. So when there are a large number of TCP connections, there will also be a lot of event deletion operations. Therefore, the total simulation time can be prolonged due to slow event deletions. To remedy this situation, we trade some space for speed by adding a heap index field into each event. The index field is updated whenever the event gets moved around in the heap array. This extra field saves the initial linear search step in event deletions in a heap and improves the event deletion time by more than two orders of magnitude at a price of 8 extra bytes for each event. And the total event scheduling time is shortened by about 20–30% compared to the previous heap-based implementation.

All the above optimizations in NS-2 are due to the large scale of our experiments, which have high requirements on CPU processing power and memory size. However, the final simulations are still very heavy weight, with over a thousand nodes and a very high aggregate request rate. We run the simulator on two platforms, one is a 4-processor/667MHz AlphaServer ES-40 with 8GB RAM and the other is a 4-way HP/Intel rx4610 IA-64 Itanium with 16GB RAM. Each simulation requires 2-6GB of RAM, and generally takes 20-50 hours of wall-clock time.

3.2 Network Topology

It is not easy to find a topology that is both realistic and makes the simulation manageable. We choose to use a slightly modified version the NSFNET backbone network T3 topology, as shown in Figure 3.4.

In this topology, the round-cornered boxes represent backbone routers with the approximate geographical location label on it. The circles, tagged as R1, R2..., are regional routers;[1] small circles with "C" stand for client hosts; and shaded circles with "S" are the server surrogates. In the particular configuration shown in the figure, we put 64 servers behind regional routers R0, R1, R7, R8, R9, R10, R15, R19, where each router sits in front of 8 servers. We distribute 1,000 client hosts evenly behind the other regional routers, yielding a topology of nearly 1,100 nodes. The redirector algorithms run on the regional routers that sit in front of the clients.

Based on common practice and without loosing diversity, the latencies of servers to regional routers are set randomly between 1ms to 3ms, those of clients to regional routers are between 5ms and 20ms, those of regional routers to backbone routers are between 1 to 10ms, and latencies between backbone routers are set roughly according to their geographical distances, ranging from 8ms to 28ms.

To simulate high request volume, we deliberately provision the network with high link bandwidth by setting the backbone links at 2,488Mbps, and links between regional routers and backbone routers at 622Mbps. Links between servers and regional routers

[1]These can also be thought of as edge/site routers, or the boundary to an autonomous system

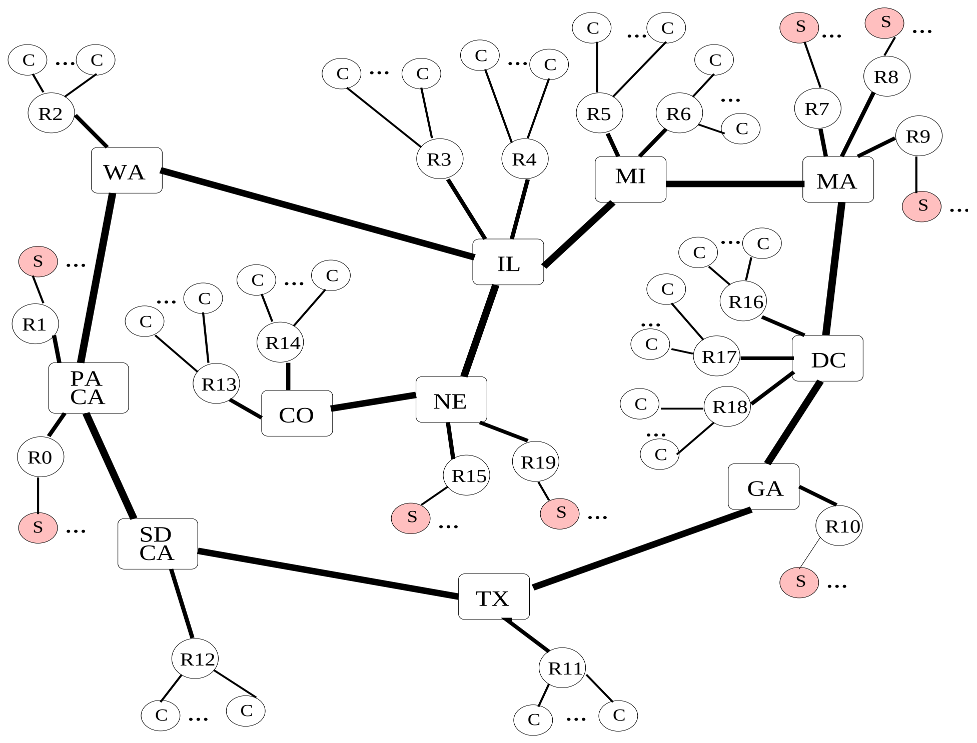

Figure 3.4: Network Topology

are 100Mbps and those between clients and their regional servers are randomly between 10Mbps and 45Mbps. All the queues at routers are drop tail, with the backbone routers having room to buffer 1024 packets, and all other routers able to buffer 512 packets.

It is worth noting that network topology generators are also used to produce synthetic topologies in many network simulations. The types of topology generators can span a wide range. There are random topology generators based on Waxman [82] model. There are also structural generators like GT-ITM [10] that generate topologies based on hierarchy, such as Transit-Stub [10] and Tiers [27]. And most recently, inspired by the power-law properties of degree distributions discovered in Internet topology graphs [29], there appear a series of degree-based generators such as BRITE [49] and INET [40]. An in-depth analysis of these generators is beyond the scope of this thesis, and a recent comparison study can be found in [77]. In contrast, our topology is based on a real backbone. It is both easy to manage and also provides proximity variations among nodes. It bears the flavor of Transit-Stub model as well. Besides these generated topologies, there's also some recent effort on inferring an ISP's topology [73]. Both of these generated and inferred topologies can potentially benefit our future experiments.

3.3 Workload and Stability

We determine system capacity using a trace-driven simulation and gradually increase the aggregate request rate until the system fails. We use a two month trace of server logs obtained at Rice University, which contains 2.3 million requests for 37,703 files with a total size of 1,418MB [58], and has properties similar to other published traces.

Number of Requests				URL Size in Bytes			
Total	Max	Median	Min	Total	Max	Median	Min
2,290,909	47,170	6	1	1,418M	26.6M	2,529	1

Table 3.1: Rice Trace Properties

Table 3.1 gives a summary of the trace in terms of the total number of requests, the maximum, median and minimum number of requests that a URL gets, and total, maximum, median and minimum URL sizes in bytes. Figure 3.5 and 3.6 further visualize different aspects of the trace. Figure 3.5 shows the cumulative URL size distribution, and the curve is heavy-tailed. Figure 3.6 illustrates individual URL's popularity and it roughly follows Zipf's Law [87]. Zipf's Law states that if files are ordered from most popular to least popular, the number of references to a file (P) tends to be inversely proportional to its rank (r), i.e. $P = kr^{-1}$. All these properties of the Rice trace we use conform to previous findings about web traces in the literature [6, 87].

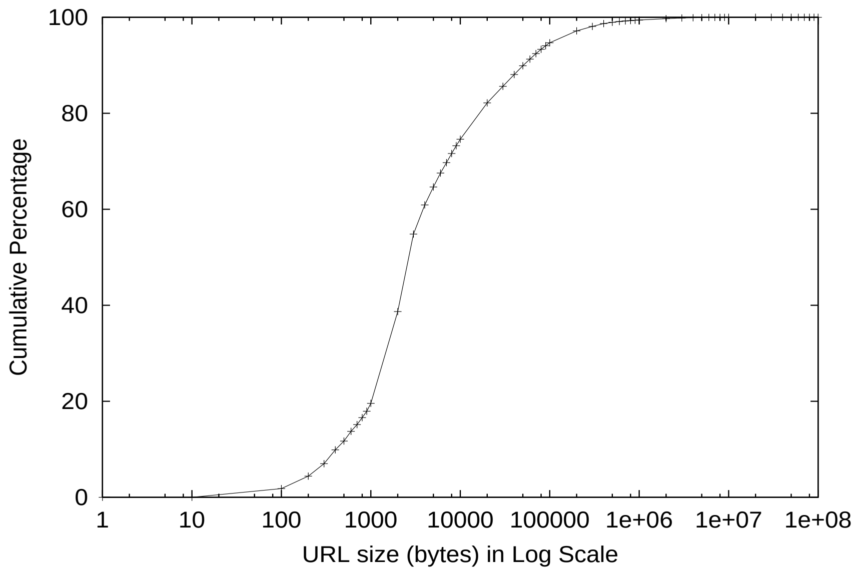

Figure 3.5: URL Size Distribution of Rice Trace

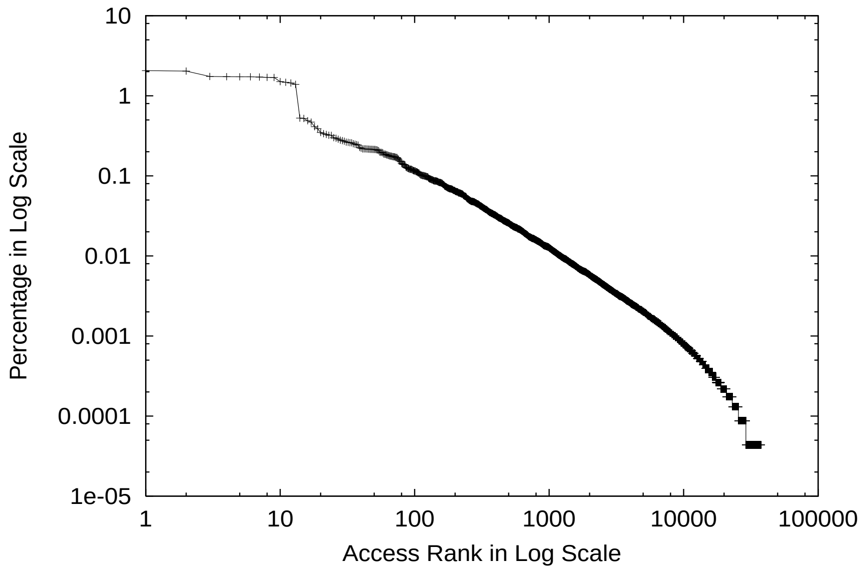

Figure 3.6: Popularity of URLs in Rice Trace

The simulation starts with the clients sharing the trace and sending requests at a low aggregate rate in an open-queue model (clients keep sending requests at a rate-defined time interval without waiting for previous ones to finish). Each client gets the name of the document sequentially from the shared trace when it needs to send a request, and the timing information in the trace is ignored. The request rate is increased by 1% every simulated six seconds, regardless of whether previous requests have completed. This approach gradually warms the server memory caches and drives the servers to their limits over time as described below. We configure LogSim to handle at most 512 simultaneous requests and queue the rest. The simulation is terminated when the offered load overwhelms the servers.

We define a server as being overwhelmed when it can no longer satisfy the rate of incoming requests and is unlikely to recover in the future. This approach is designed to determine when service is actually being denied to clients, and to ignore any short-term behavior which may be only undesirable, rather than fatal. Through experimentation, we find that when a server's request queue grows beyond 4 to 5 times the number of simultaneous connections it can handle, throughput drops and the server is unlikely to recover. Thus, we define the threshold for a server *failure* to be when the request queue length exceeds five times the simultaneous connection parameter. Since we increase the offered load 1% every 6 seconds, we record the request load exactly 30 seconds before the first server fails, and declare this to be the system's maximum capacity.

Although we regard any single server failure as a system failure in our simulation, the strategies we evaluate all exhibit similar behavior—significant numbers of servers fail at the same time, implying that our approach to deciding system capacity is not biased toward any particular scheme.

Flash crowds, or DDoS attacks in bursty legitimate traffic form, are simulated by randomly selecting some clients as *intensive* requesters and randomly picking a certain number of hot-spot documents. These intensive requesters randomly request the hot documents at the same rate as normal clients, making them look no different than other legitimate users. Therefore, “intensive” does not mean these requesters send requests much faster, rather indicates that they focus on some hot-spots. We believe that this random distribution of intensive requesters and hot documents is a quite general assumption since we do not require any special detection or manual intervention to signal the start of a flash crowd or DDoS attack.

Chapter 4

Performance Results

This section evaluates how the different strategies in Table 2.1 perform, both under normal conditions and under flash crowds or DDoS attacks. Network proximity and other factors that affect the performance of these strategies are also addressed.

4.1 Normal Workload

Before evaluating these strategies under flash crowds or other attack, we first measure their behavior under normal workloads. In these simulations, all clients generate traffic similar to normal users and gradually increase their request rates as discussed in Section 3.3. We compare aggregate system capacity and user-perceived latency under the different strategies, using the topology shown in Figure 3.4.

4.1.1 Optimal Static Replication

The static replication schemes (R-CHash, R-HRW, and their variants) use a configurable (but fixed) number of replicas, and this parameter's value influences their performance.

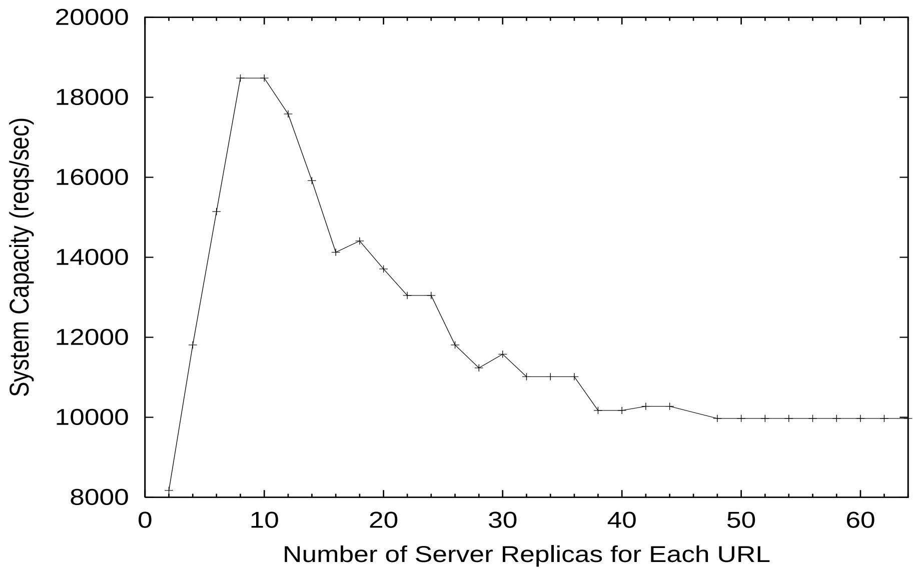

Figure 4.1: Finding Optimal Number of Replicas for R-HRW in 64 Server Case

Using a single replica per URL perfectly partitions the file set, but can lead to early failure of servers hosting popular URLs. Using as many replicas as available servers degenerates to the Random strategy. To determine an appropriate value, we varied this parameter between 2 and 64 replicas for R-HRW when there are 64 servers available, with the results shown in Figure 4.1. The results for R-CHash are similar. Increasing the number of replicas per URL initially helps to improve the system's throughput as the load gets more evenly distributed. Beyond a certain point, throughput starts decreasing due to the fact that each server is presented with a larger working set, causing more disk activity. In the 64-server case—the scenario we use throughout the rest of this section—10 server replicas for each URL achieves the optimal system capacity. For all of the remaining experiments, we use this value in the R-CHash and R-HRW schemes and their variants.

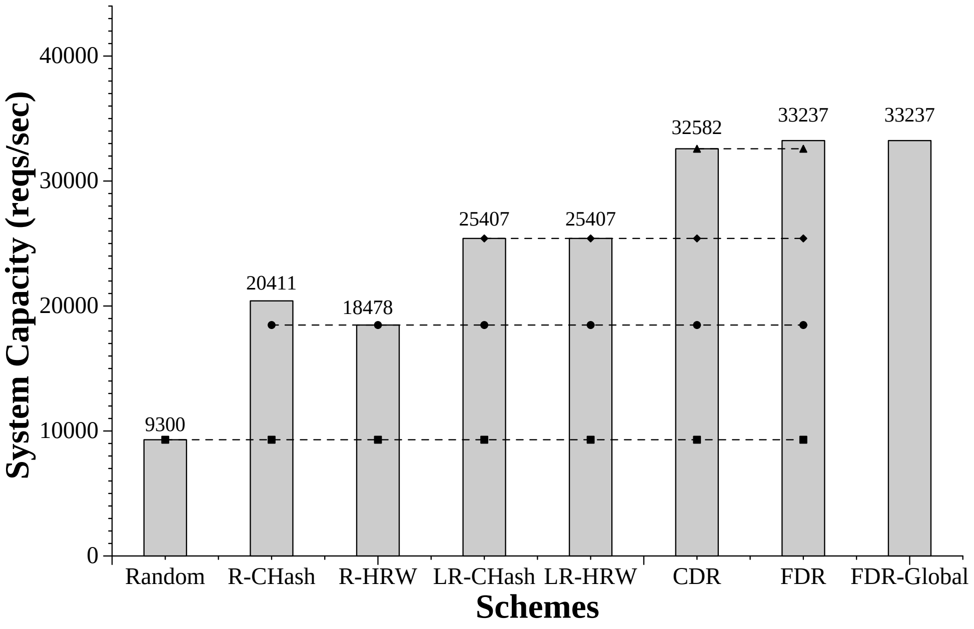

Figure 4.2: Capacity Comparison under Normal Load

4.1.2 System Capacity

The maximum aggregate throughput of the various strategies with 64 servers are shown in Figure 4.2. To determine the capacity numbers, we use the failure threshold of the server request queue length discussed at the end of Section 3.3. Here we do not plot all the strategies and variants, but focus on those impacting throughput substantially. Random shows the lowest throughput at 9,300 req/s before overload. The static replication schemes, R-CHash and R-HRW, outperform Random by 119% and 99%, respectively. Our approximation of static schemes' best behaviors, LR-CHash and LR-HRW, yields 173% better capacity than Random. The dynamic replication schemes, CDR and FDR, show over 250% higher throughput than Random, or more than a 60% improvement over the static approaches and 28% over static schemes with fine-grained load control.

The difference between Random and the static approaches stems from the locality benefits of the hashing in the static schemes. By partitioning the working set, more documents are served from memory by the servers. Note, however, that absolute minimal replication can be detrimental, and in fact, the throughput for only two replicas in Figure 4.1 of Section 4.1.1 is actually lower than the throughput for Random. The difference in throughput between R-CHash and R-HRW is 10% in our simulation. However, this difference should not be over emphasized, because changes in the number of servers or workload can cause their relative ordering to change. Considering load helps the static schemes gain about 25% better throughput, but they still do not exceed the dynamic approaches.

The performance difference between the static (including with load control) and dynamic schemes stems from the adjustment of the number of replicas for the documents. FDR also shows 2% better capacity than CDR. By assigning a small number of replicas for most (unpopular) URLs, the working set size on each server is effectively reduced. As a result, the memory cache on each server has a higher hit rate and documents are served faster directly from memory than from disks. On the other hand, when the load on one server becomes too high due to some popular URLs, dynamic schemes also intelligently migrate the load to other servers by increasing the number of replicas. Dynamic schemes thus achieve a good balance between cache locality and server load.

Interestingly, the difference between our dynamic schemes (with only local knowledge) and the FDR-Global policy (with perfect global knowledge) is minimal. These results suggest that request distribution policies not only fare well with only local information, but that adding more global information may not gain much in system capacity.

Examination of what ultimately causes overload in these systems reveals that, under normal load, the server's behavior is the factor that determines the performance limit of the system. None of the schemes suffers from saturated network links in these non-attack simulations. For Random, due to the large working set, the disk performance is

Utilization Scheme	*CPU (%)*		*DISK (%)*	
	Mean	Stddev	Mean	Stddev
Random	21.03	1.36	**100.00**	0.00
R-CHash	57.88	18.36	**99.15**	3.89
R-HRW	47.88	15.33	**99.74**	1.26
LR-CHash	59.48	18.85	**97.83**	12.51
LR-HRW	58.43	16.56	**99.00**	5.94
CDR	**90.07**	11.78	36.10	25.18
FDR	**93.86**	7.58	33.96	20.38
FDR-Global	**91.93**	11.81	17.60	15.43

Table 4.1: Server Resource Utilization at Overload under Normal Load

the limit of the system, and before system failure, the disks exhibit almost 100% activity while the CPU remains largely idle. The R-CHash, R-HRW and LR-CHash and LR-HRW exhibit much lower disk utilization at comparable request rates; but by the time the system becomes overloaded, their bottleneck also becomes the disk and the CPU is roughly 50-60% utilized on average. In the CDR and FDR cases, at system overload, the average CPU is over 90% busy, while most of the disks are only 10-70% utilized. Table 4.1 summarizes resource utilization of different schemes before server failures (not at the same time point).

These results suggest that the CDR and FDR schemes are the best suited for technology trends, and can most benefit from upgrading server capacities. From past experiences, the speed of CPUs and networks generally increases faster that that of disks, and the performance gap between CPUs and disks is expected to become even larger in the future. Shifting system bottlenecks from disks to CPUs clearly fits this trend. In fact, the throughput of our simulated machines is lower than what can be expected from state-of-the-art machines, but this decision to scale down resources was made to keep the simulation time manageable. *With faster simulated machines, we expect the gap between the dynamic schemes and the others to grow even larger.*

4.1.3 Response Latency

Along with system capacity, the other metric of interest is user-perceived latency, and we find that our schemes also perform well in this regard. To understand the latency behavior of these systems, we use the capacity measurements from Figure 4.2 and analyze the latency of all of the schemes whenever one category reaches its performance limit. For schemes with similar performance in the same category, we pick the lower limit for the analysis so that we can include numbers for the higher-performing scheme. In all cases, we present the cumulative distribution of all request latencies as well as some statistics about the distribution.

Figure 4.3–4.6 plot the cumulative distribution of latencies at four request rates: the maximums for Random, R-HRW, LR-HRW, and CDR (the algorithm in each category with the smallest maximum throughput). The *x*-axis is in log scale and shows the time needed to complete requests. The *y*-axis shows what fraction of all requests finished in that time. The data in Table 4.2 gives mean, median, 90th percentile and standard deviation details of response latencies at our comparison points.

The response time improvement from exploiting locality is most clearly seen in Figure 4.3. At Random's capacity, most responses complete under 4 seconds, but a few responses take longer than 40 seconds. In contrast, all other strategies have median times almost one-fourth that of Random, and even their 90th percentile results are less than Random's median. These results, coupled with the disk utilization information, suggest

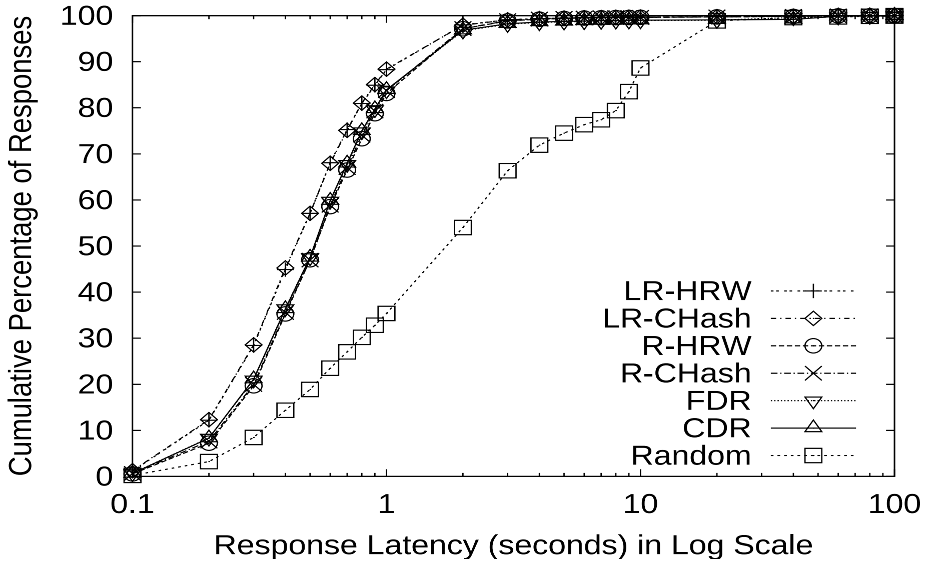

Figure 4.3: Response Latency under Normal Load, [Random's limit: 9,300 req/s]

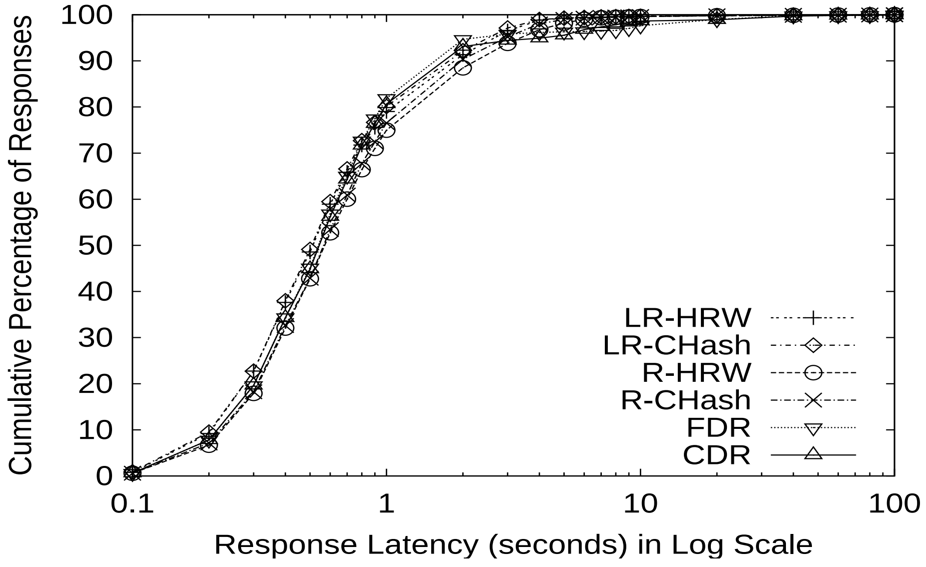

Figure 4.4: Response Latency under Normal Load, [R-HRW's limit: 18,478 req/s]

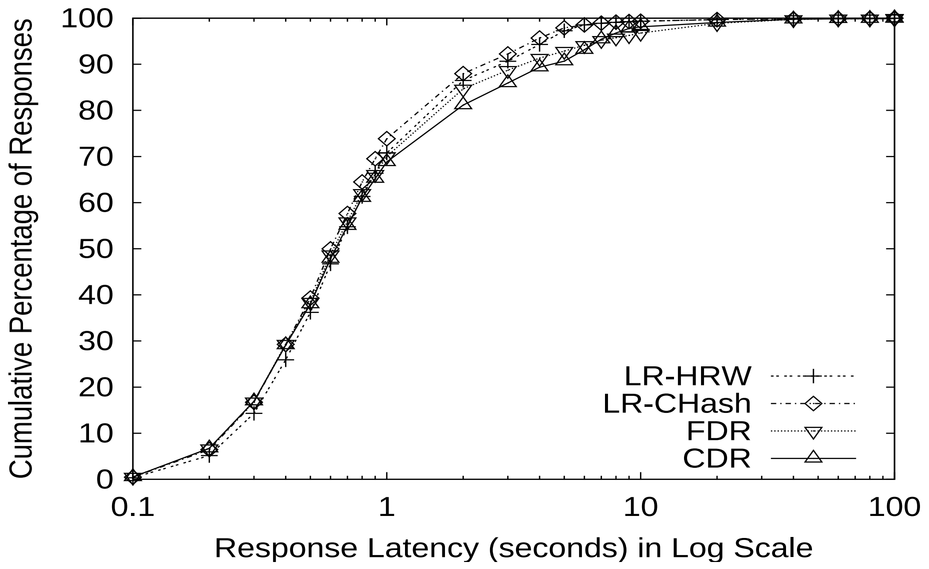

Figure 4.5: Response Latency under Normal Load, [LR-HRW's limit: 25,407 req/s]

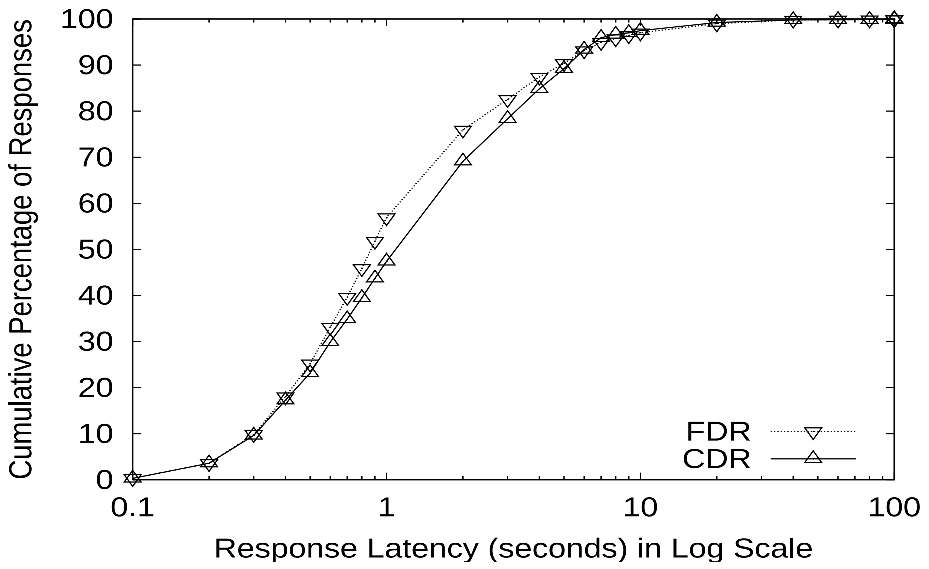

Figure 4.6: Response Latency under Normal Load, [CDR's limit: 32,582 req/s]

that most requests in the Random scheme are suffering from disk delays, and that the locality improvement techniques in the other schemes are a significant benefit.

The benefit of FDR over CDR is visible in Figure 4.6, where the plot for FDR lies to the left of CDR. The statistics also show a much better median response time, in addition to better mean and 90th percentile numbers. This is because FDR avoids unnecessary replication in CDR, further reducing the working set size and increasing the cache locality on each server. FDR-Global has better numbers in all cases than CDR and FDR, due to its perfect knowledge of server load status.

An interesting observation is that when compared to the static schemes, dynamic schemes have worse mean times but comparable (sometimes better) medians and 90th percentile results. We find that this behavior stems from the time required to serve the largest files. Since generally and also verified in our trace, these large files are less popular (being accessed less), the dynamic schemes replicate them less than the static schemes do. As a result, these files are served from a smaller set of servers, causing them to be served more slowly than if they were replicated more widely. This can also been seen from the relatively large latency standard deviations of dynamic schemes, which suggest that their latency distributions have a fairly large tail. We do not consider this behavior to be a significant drawback, and note that some research explicitly aims to achieve this effect [22, 23]. We will revisit large file issues in Section 4.4.2.

Req Rate	*9,300 req/s*				*18,478 req/s*				*25,407 req/s*				*32,582 req/s*			
Latency	μ	50%	90%	σ	μ	50%	90%	σ	μ	50%	90%	σ	μ	50%	90%	σ
Random	3.95	1.78	11.32	6.99												
R-CHash	0.79	0.53	1.46	2.67	1.01	0.57	1.98	3.58								
R-HRW	0.81	0.53	1.49	2.83	1.07	0.57	2.28	3.22								
LR-CHash	0.68	0.44	1.17	2.50	0.87	0.51	1.82	2.74	1.19	0.60	2.47	3.79				
LR-HRW	0.68	0.44	1.18	2.50	0.90	0.51	1.89	3.13	1.27	0.64	2.84	3.76				
CDR	1.16	0.52	1.47	5.96	1.35	0.55	1.75	6.63	1.86	0.63	4.49	6.62	2.37	1.12	5.19	7.21
FDR	1.10	0.52	1.48	5.49	1.35	0.54	1.64	6.70	1.87	0.62	3.49	6.78	2.22	0.87	4.88	7.12
FDR-Global	0.78	0.50	1.42	2.88	0.97	0.54	1.58	5.69	1.11	0.56	1.86	5.70	1.35	0.66	2.35	6.29

Table 4.2: Response Latency of Different Strategies under Normal Load. μ — Mean, σ — Standard Deviation.

4.1.4 Scalability

Robustness not only comes from resilience with certain resources, but also from good scalability with increasing resources. We repeat similar experiments with different number of servers, from 8 to 128, to test how well these strategies scale. The number of server-side routers is not changed, but instead, more servers are attached to each server router as the total number of servers increases.

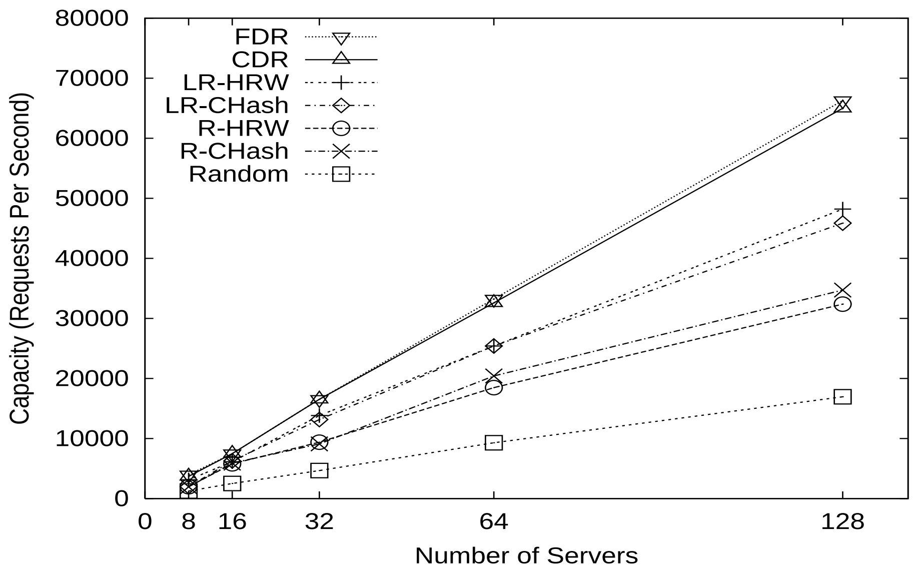

Figure 4.7: System Scalability under Normal Load

We plot system capacity against the number of servers in Figure 4.7. They all display near-linear scalability, implying all of them are reasonably good strategies when the system becomes larger. Note, for CDR and FDR with 128 servers, our original network provision is not enough. The bottleneck in that case is the link between the server router and backbone router, which is 622Mbps. In this scenario, each server router is handling 16 servers, giving each server on average only 39Mbps of traffic. At 600 reqs/s, even an average size of 10KB requires 48Mbps. Under this bandwidth setup, CDR and FDR yield similar system capacity as LR-CHash and LR-HRW, and all these 4 strategies saturate server-router-to-backbone links. To remedy this situation, we run simulations of 128 servers for all strategies with doubled bandwidth on both the router-to-backbone and backbone links. Performance numbers of 128 servers under these faster links are plotted in the graph instead. This problem can also be solved by placing fewer servers behind each pipe and instead spreading them across more locations.

4.2 Behavior Under Flash Crowds

Having established that our new algorithms perform well under normal workloads, we now evaluate how they behave when the system is under a flash crowd or DDoS attack. To simulate a flash crowd, we randomly select 25% of the 1,000 clients to be *intensive* requesters, where each of these requesters repeatedly issues requests from a small set of pre-selected URLs with an average size of about 6KB, but still at the same request rate as normal clients. In most of the experiments below, we use 10 URLs. The total size of these URLs is about 60KB, which roughly compares to a small web page with several embedded objects.

4.2.1 System Capacity

Figure 4.8 depicts the system capacity of 64 servers under a flash crowd. In general, it exhibits similar trends as the no-attack case shown in Figure 4.2. Importantly, the CDR and FDR schemes still yield the best throughput, making them most robust to flash crowds or attacks. Two additional points deserve more attention.

First, FDR now has a similar capacity with CDR, but still is more desirable as it provides noticeably better latency, as we will see later. FDR's benefit over R-CHash and

R-HRW has grown to 91% from 60% and still outperforms LR-CHash and LR-HRW by 22%.

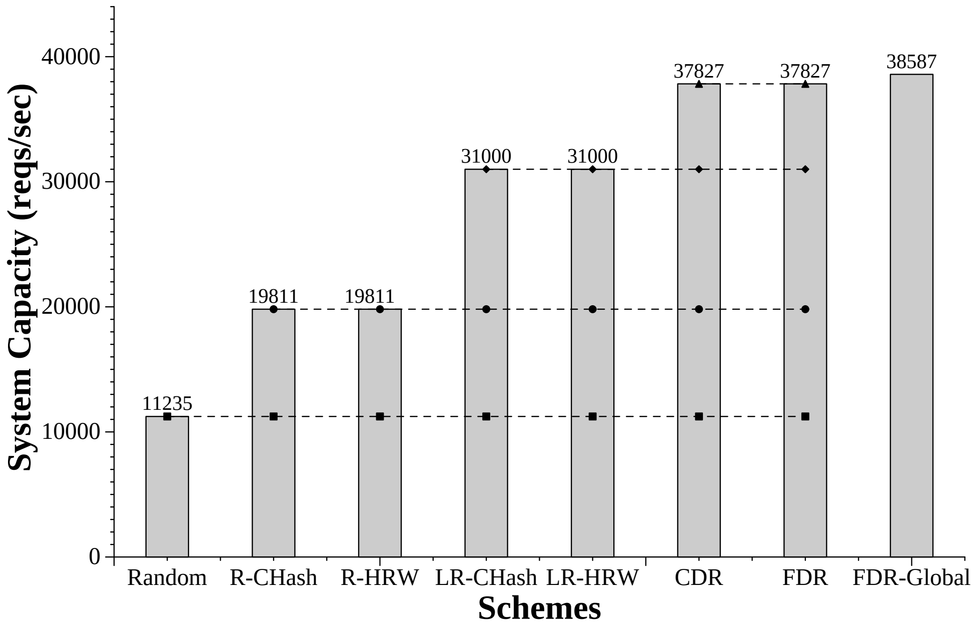

Figure 4.8: Capacity Comparison Under Flash Crowds

Second, the absolute throughput numbers tend to be larger than the no-attack case, because the workload is also different. Here, 25% of the traffic is now concentrated on 10 URLs, and these attack URLs are relatively small, with an average size of 6KB. Therefore, the relative difference among different strategies within each scenario yields more useful information than simply comparing performance numbers across these two scenarios.

Table 4.3 summarizes resource utilization of different schemes before server failures at their individual maximum capacities. Again, our dynamic schemes shift the system bottleneck from disks to CPUs, which is desirable for improving system capacity.

Utilization Scheme	*CPU (%)*		*DISK (%)*	
	Mean	Stddev	Mean	Stddev
Random	19.60	2.52	**100.00**	0.00
R-CHash	51.27	20.30	**93.01**	12.16
R-HRW	51.86	16.47	**91.70**	12.94
LR-CHash	57.38	18.64	**96.91**	13.86
LR-HRW	57.90	17.98	**98.97**	5.85
CDR	**94.49**	8.23	57.94	27.03
FDR	**92.61**	10.27	23.85	19.61
FDR-Global	**91.98**	11.04	7.98	13.81

Table 4.3: Server Resource Utilization at Overload under Flash Crowds

4.2.2 Response Latency

The cumulative distribution of response latencies for all seven algorithms under attack are shown in Figure 4.9–4.12. Also, the statistics for all seven algorithms and FDR-Global are given in Table 4.4. As seen from the figures and table, R-CHash, R-HRW, LR-CHash, LR-HRW, CDR, and FDR still have far better latency than Random, and static schemes are a little better than CDR and FDR at Random, R-HRW's and LR-HRW's failure points; and LR-CHash and LR-HRW yields slightly better latency than R-CHash and R-HRW.

As we explained earlier, CDR and FDR adjust the server replica set in response to request volume. The number of replicas that serve attack URLs increases as the attack ramps up, which may adversely affect serving non-attack URLs. However, the differences in the mean, median, and 90-percentile are not large, and all are probably acceptable to users. The small price paid in response time for CDR and FDR brings us higher system capacity, and thus, stronger resilience to various loads.

Again, due to more concentrated traffic on small URLs, under flash crowds, latency numbers are slightly smaller than those under normal load.

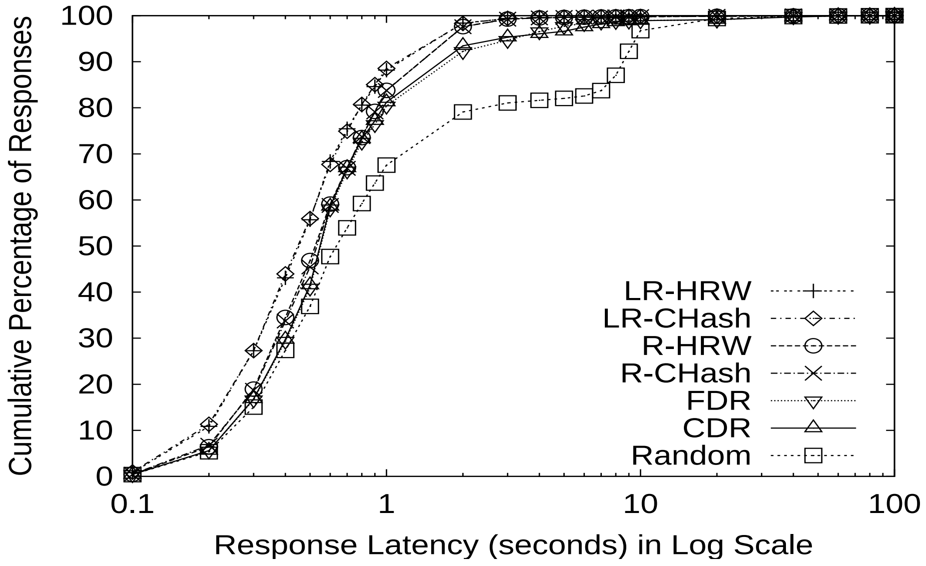

Figure 4.9: Response Latency under Flash Crowds, [Random's limit: 11,235 req/s]

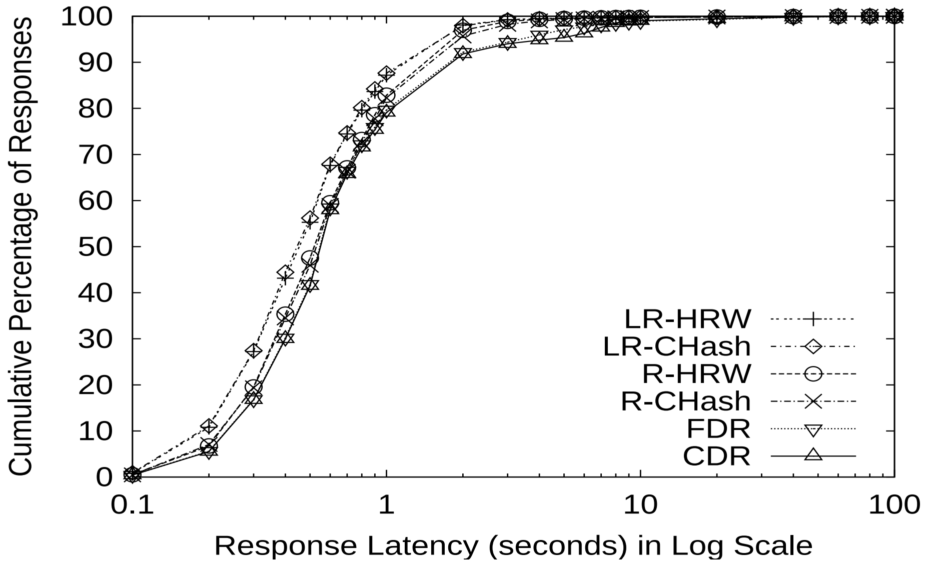

Figure 4.10: Response Latency under Flash Crowds, [R-HRW's limit: 19,811 req/s]

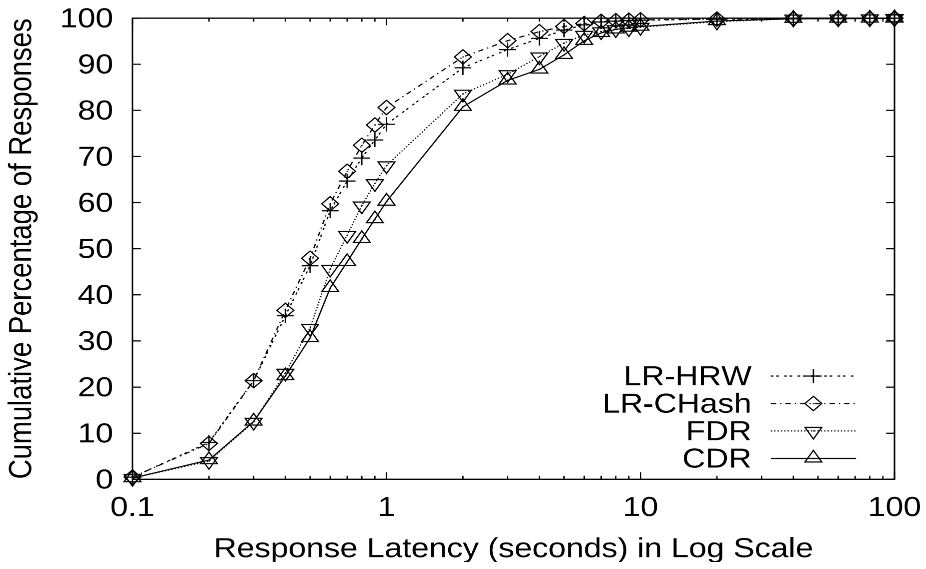

Figure 4.11: Response Latency under Flash Crowds, [LR-HRW's limit: 31,000 req/s]

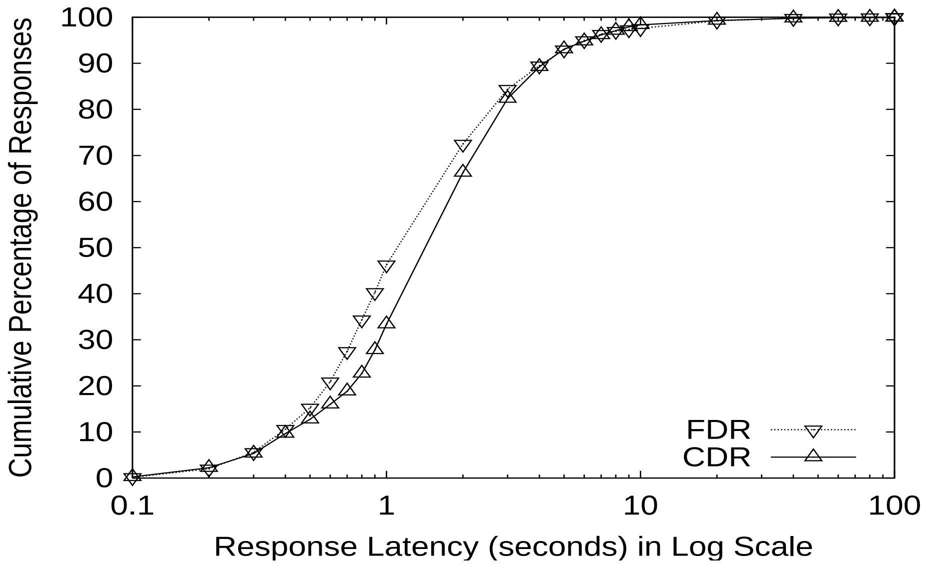

Figure 4.12: Response Latency under Flash Crowds, [CDR's limit: 37,827 req/s]

Req Rate	*11,235 req/s*				*19,811 req/s*				*31,000 req/s*				*37,827 req/s*			
Latency	μ	50%	90%	σ	μ	50%	90%	σ	μ	50%	90%	σ	μ	50%	90%	σ
Random	2.37	0.64	8.57	5.29												
R-CHash	0.73	0.53	1.45	2.10	0.81	0.53	1.57	2.59								
R-HRW	0.73	0.52	1.45	2.11	0.76	0.52	1.51	2.51								
LR-CHash	0.62	0.45	1.15	1.70	0.67	0.45	1.23	2.42	0.96	0.52	1.86	3.55				
LR-HRW	0.63	0.45	1.18	1.80	0.67	0.46	1.26	2.65	1.07	0.53	2.19	3.52				
CDR	1.19	0.55	1.72	5.40	1.25	0.55	1.86	5.51	1.80	0.76	4.35	6.08	2.29	1.50	4.20	6.41
FDR	1.22	0.55	1.81	5.71	1.18	0.55	1.83	5.27	1.64	0.66	3.57	5.95	2.18	1.14	4.15	6.63
FDR-Global	0.91	0.55	1.66	4.09	0.90	0.53	1.60	4.59	0.98	0.54	1.74	5.08	1.20	0.56	1.99	5.53

Table 4.4: Response Latency of Different Strategies under Flash Crowds. μ — Mean, σ — Standard Deviation.

4.2.3 Scalability

We also repeat the scalability test under a flash crowd or attack, where 250 clients are *intensive* requesters that repeatedly request 10 URLs. As shown in Figure 4.13, all strategies scale linearly with the number of servers. Again, in the 128-server cases, we use doubled bandwidth on the router-to-backbone and backbone links.

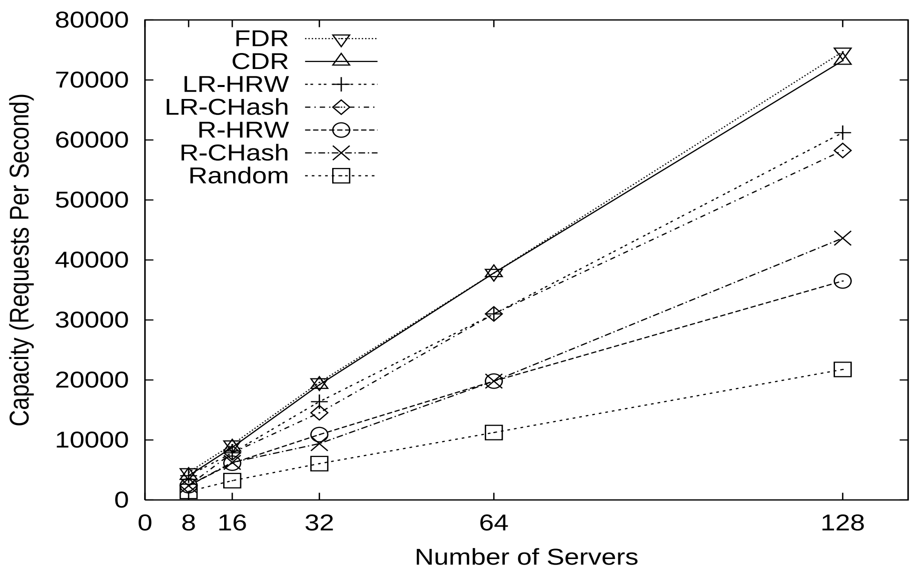

Figure 4.13: System Scalability under Flash Crowds

4.2.4 Various Flash Crowds

Throughout our simulations, we have seen that a different number of *intensive* requesters, and a different number of hot or attacked URLs, have an impact on system performance. To further investigate this issue, we carry out a series of simulations by varying both the number of intensive requesters and the number of hot URLs. Since it is impractical to exhaust all possible combinations, we choose two classes of flash crowds. One class has a single hot URL of size 1KB. This represents a small icon or image file. The other class has 10 hot URLs averaging 6KB, as before. In both cases, we vary the percentage of the 1000 clients that are intensive requesters from 10% to 80%. The results of these two experiments with 32 servers are shown in Figures 4.14 and 4.15, respectively.

In the first experiment, as the portion of *intensive* requesters increases, more traffic is concentrated on this one URL, and the request load becomes more unbalanced. Random, CDR and FDR adapt to this change well and yield increasing throughput. This benefit comes from their ability to spread load across more servers. Random performs well simply because the hot URL has a small size, and on each server, it can be kept in memory cache due to frequent requests. As the request load becomes more focused on this URL, Random utilizes all the servers to serve it from memory caches and gets increasing throughput. However, CDR and FDR behave better than Random because they not only adjust the server replica set on demand, but also maintain server locality for less popular URLs. In contrast, R-HRW, R-CHash, LR-HRW and LR-CHash suffer with more intensive requesters or attackers, since their fixed number of replicas for each URL cannot handle the high volume of requests for one URL. In the 10-URL case, the change in system capacity looks similar to the 1-URL case, except that due to a larger amount of data being intensively requested or attacked, CDR and FDR cannot sustain the same high throughput. We continue to investigate the effects of more attack URLs and other strategies.

Another possible DDoS attack scenario is to randomly select a wide range of URLs. In the case that these URLs are valid, the dynamic schemes “degenerate” into one server for each URL. This is the desirable behavior for this attack as it increases the cache hit rates for all the servers. In the event that the URLs are invalid, and the servers are actually reverse proxies (as is typically the case in a CDN), then these invalid URLs are forwarded

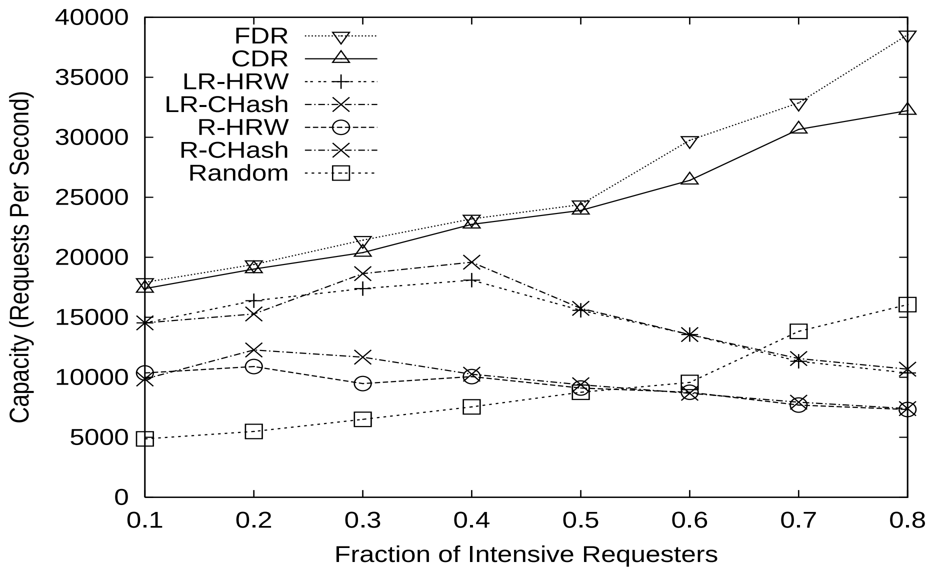

Figure 4.14: 1 Hot URL, 32 Servers, 1000 Clients

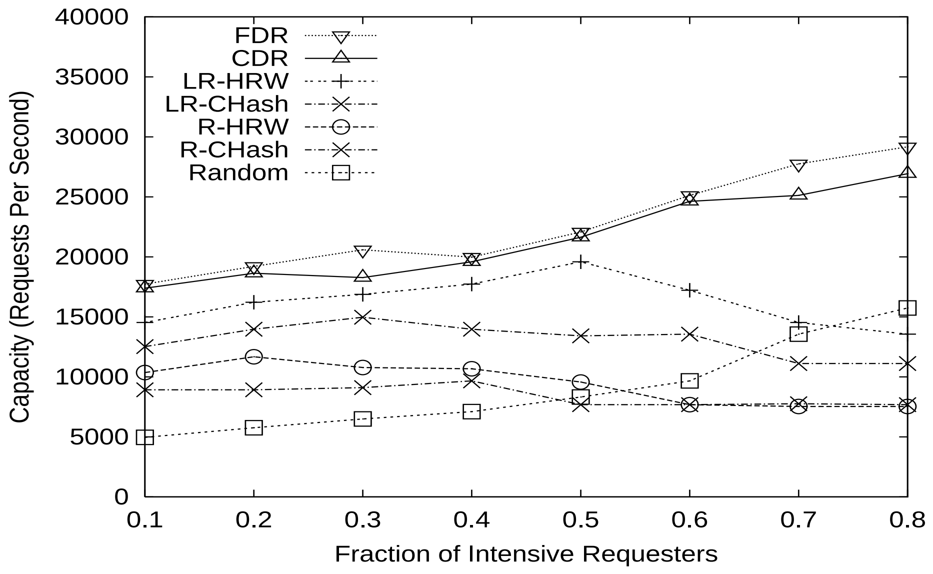

Figure 4.15: 10 Hot URL, 32 Servers, 1000 Clients

to the server-of-origin, effectively overloading it. Servers must address this possibility by throttling the number of URL-misses they forward.

To summarize, under flash crowds or attacks, CDR and FDR sustain very high request volumes, making overloading the whole system significantly harder and thereby greatly improving the CDN system's overall robustness.

4.3 Proximity

The previous experiments focus on system capacity under different loads. We now compare the strategies that factor network closeness into server selection—Static (NPR-CHash), Static+Load (NPLR-CHash), and Dynamic (NP-FDR)—with their counterparts that ignore proximity. We test the 64-server cases in the same scenarios as in Section 4.1 and 4.2. Since commercial CDNs typically use Consistent Hashing based static strategies, we do not evaluate those corresponding HRW based proximity variants.

	System Capacity (reqs/sec)		
Category	Scheme	Normal	Flash Crowds
Static	*NPR-CHash*	14409	14409
	R-CHash	20411	19811
Static +Load	*NPLR-CHash*	24173	30090
	LR-CHash	25407	31000
Dynamic	*NP-FDR*	31000	34933
	FDR	33237	37827

Table 4.5: Proximity's Impact on Capacity

Table 4.5 shows the capacity numbers of these strategies under both normal load and flash crowds of 250 intensive requesters with 10 hot URLs. As we can see, adding network proximity into server selection slightly decreases the system capacity in the case of NPLR-CHash and NP-FDR. However, the throughput drop of NPR-CHash compared with R-CHash is considerably large. Part of reason is that in LR-CHash and FDR, server load information already conveys the distance of a server. However, in the R-CHash case, the redirector randomly choosing among all replicas causes the load to be evenly distributed, while NPR-CHash puts more burden on closer servers, resulting in unbalanced server load.

We further investigate the impact of network proximity on response latency. In Table 4.6 and 4.7, we show the latency statistics under both normal load and flash crowds. As before, we choose to show numbers at the capacity limits of Random, NPR-CHash, NPLR-CHash and NP-FDR.

In Figure 4.16–4.19, we show the latency distribution of these schemes under normal load, and the results under flash crowds are shown in Figure 4.20–4.23. Instead of plotting relevant strategies together all the way, we use a different format. For example, under normal load, we first plot all relevant strategies against the Random scheme at Random's limit of 9,300 req/s, then we plot the six schemes pair-wise at the lower capacity in each category. We want to find out the impacts of proximity on response latency.

We can see from the tables and graphs that when servers are not loaded, all schemes with network proximity taken into consideration—NPR-CHash, NPLR-CHash and NP-FDR—yield better latency. When these schemes reach their limit, NPR-CHash and NP-FDR still demonstrate a significant latency advantage over R-CHash and FDR, respectively.

Interestingly, NPLR-CHash under-performs LR-CHash in response latency at its limit of 24,173 req/s under normal load and 30,090 req/s in the case of flash crowds. NPLR-CHash is basically LR-CHash using effective load. When all the servers are not loaded, it redirects more requests to nearby servers, thus shortening the response time. However, as the load increases, in order for a remote server to get a share of load, a local server has to be much more overloaded than the remote one, inversely proportional to their distance ratio. Unlike NP-FDR, there is no load threshold control in NPLR-CHash, so it is possible that some close servers get significantly more requests, resulting in slow processing and longer responses. In summary, considering proximity may benefit latency, but it can also impact capacity. NP-FDR, however, achieves a good balance of both.

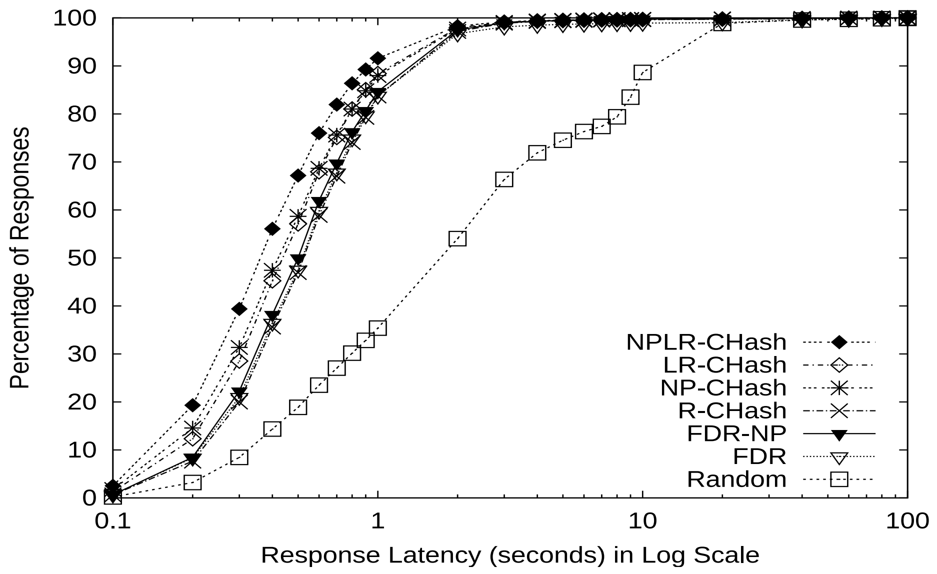

Figure 4.16: Proximity Impact on Response Latency under Normal Load, [Random limit: 9,300 req/s]

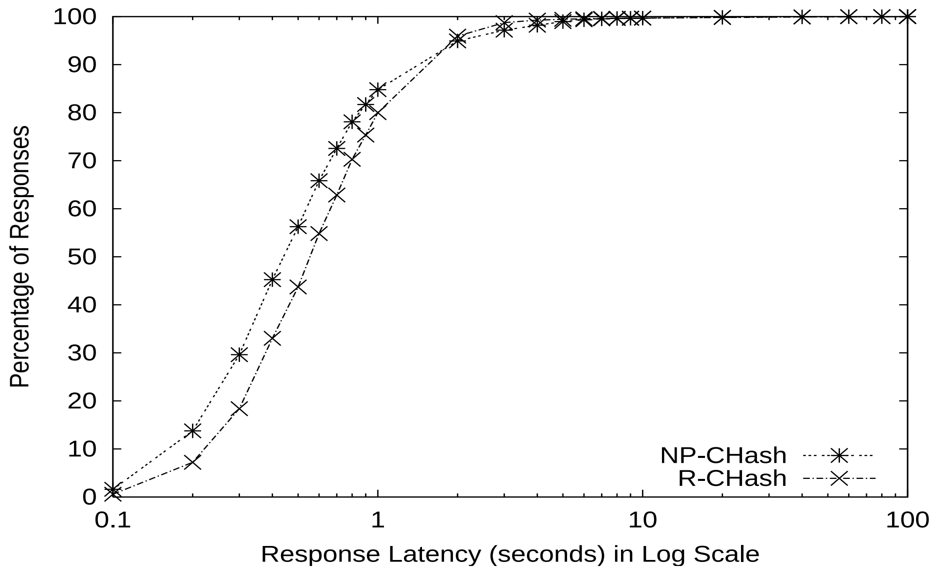

Figure 4.17: Proximity Impact on Response Latency under Normal Load, [NPR-CHash limit: 14,409 req/s]

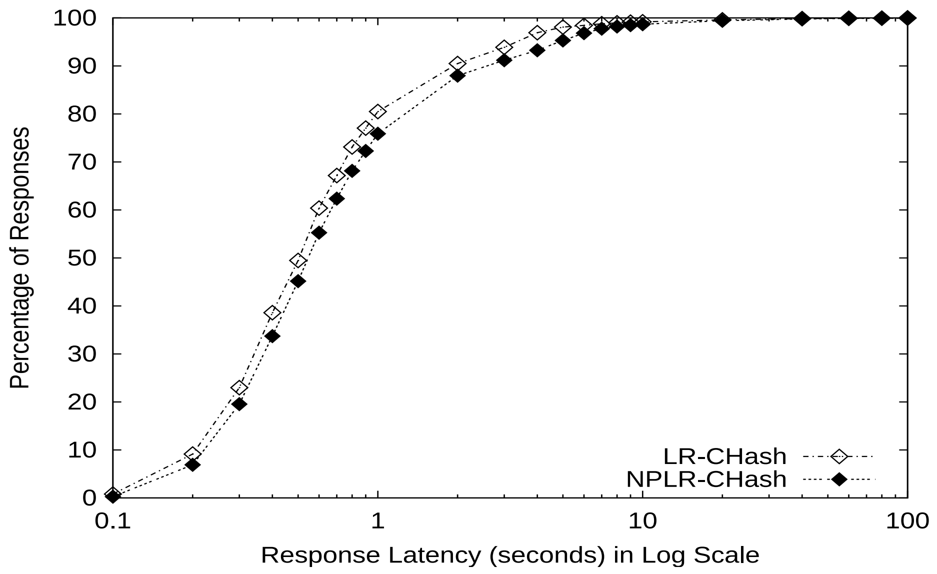

Figure 4.18: Proximity Impact on Response Latency under Normal Load, [NPLR-CHash limit: 24,173 req/s]

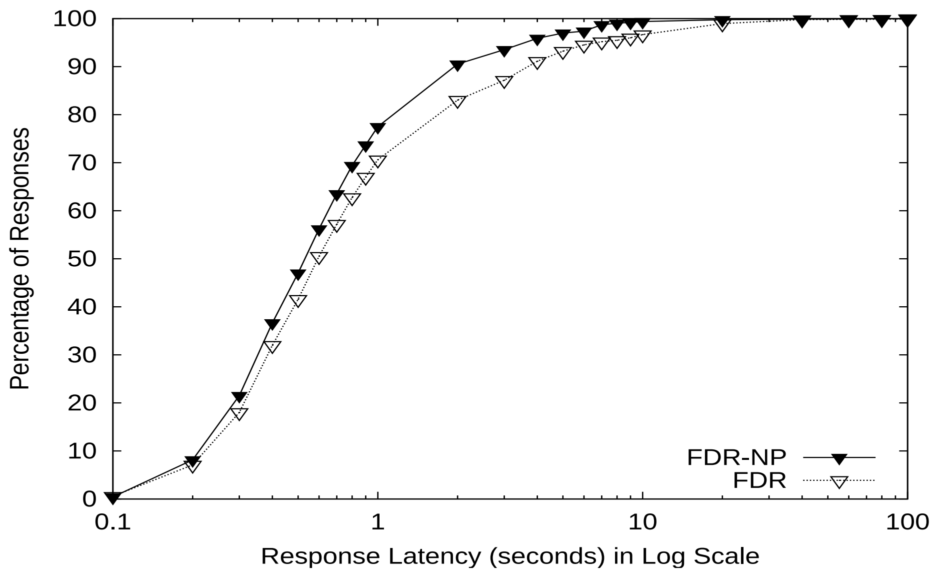

Figure 4.19: Proximity Impact on Response Latency under Normal Load, [NP-FDR limit: 31,000 req/s]

Req Rate	*9,300 req/s*				*14,409 req/s*				*24,173 req/s*				*31,000 req/s*			
Latency	μ	50%	90%	σ	μ	50%	90%	σ	μ	50%	90%	σ	μ	50%	90%	σ
Random	3.95	1.78	11.32	6.99												
NPR-CHash	0.66	0.42	1.21	2.20	0.76	0.44	1.51	2.30								
R-CHash	0.79	0.53	1.46	2.67	0.82	0.56	1.63	2.50								
NPLR-CHash	0.57	0.36	0.93	2.00	0.68	0.39	1.33	2.34	1.34	0.55	2.63	4.73				
LR-CHash	0.68	0.44	1.17	2.50	0.71	0.48	1.43	2.19	1.04	0.50	1.95	3.44				
NP-FDR	0.70	0.50	1.42	1.63	0.67	0.49	1.33	1.56	0.80	0.49	1.55	2.82	1.08	0.53	1.96	3.54
FDR	1.10	0.52	1.48	5.49	1.25	0.54	1.71	5.87	1.60	0.57	2.10	6.84	1.88	0.59	3.72	7.25

Table 4.6: Proximity's Impact on Response Latency under Normal Load. μ — Mean, σ — Standard Deviation.

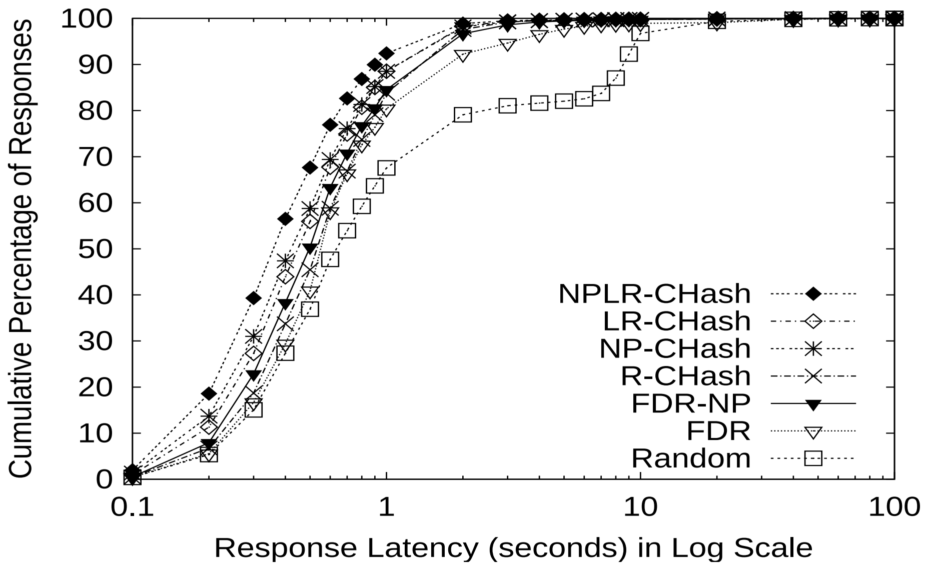

Figure 4.20: Proximity Impact on Response Latency under Flash Crowd, [Random limit: 11,235 req/s]

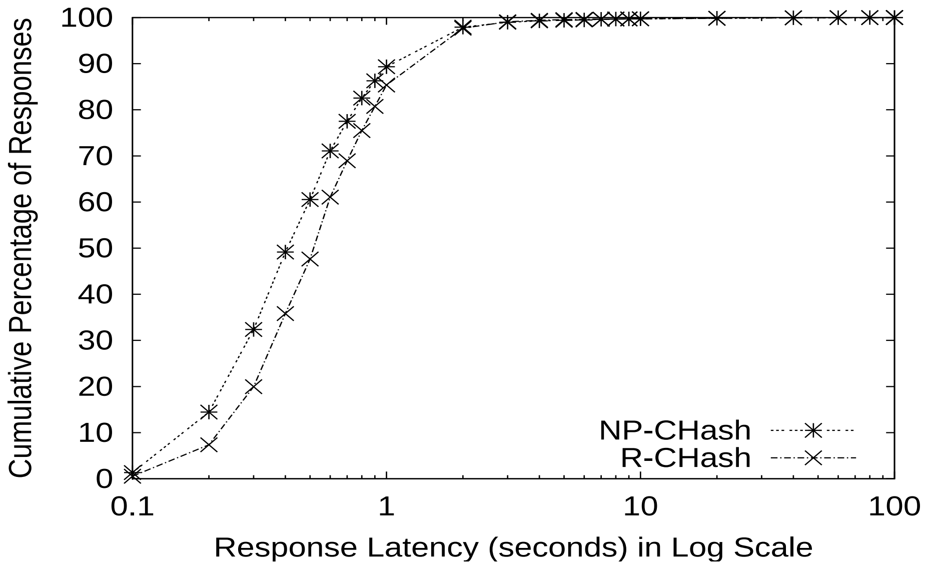

Figure 4.21: Proximity Impact on Response Latency under Flash Crowd, [NPR-CHash limit: 14,409 req/s]

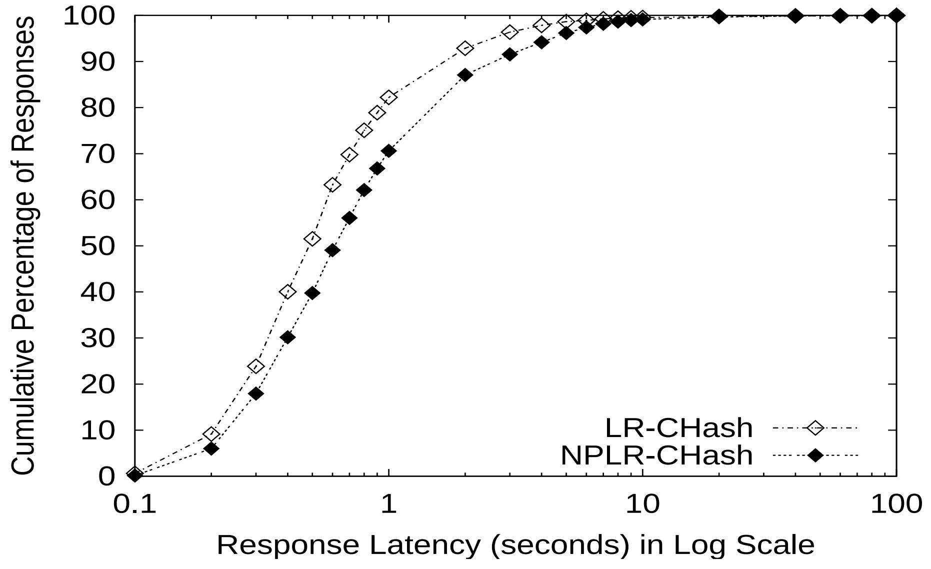

Figure 4.22: Proximity Impact on Response Latency under Flash Crowd, [NPLR-CHash limit: 30,090 req/s]

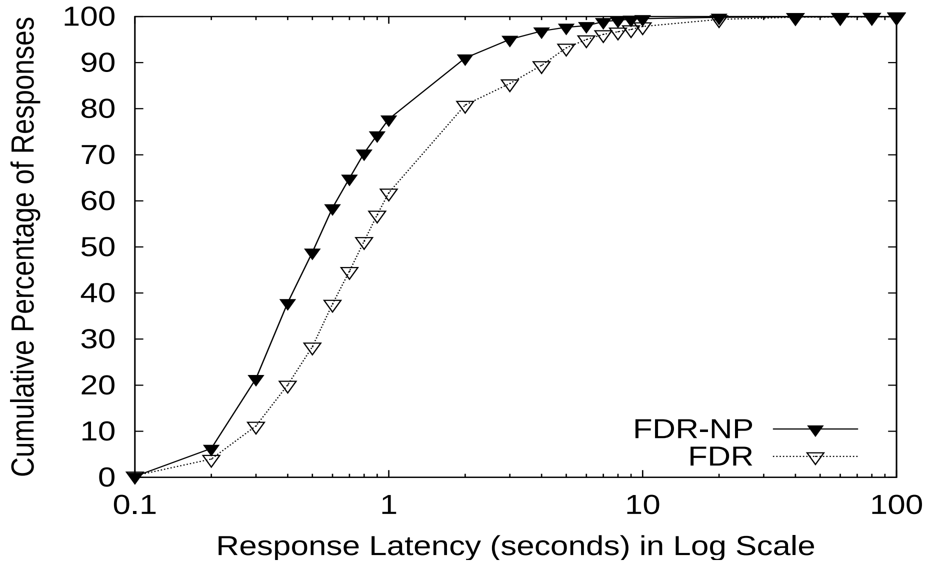

Figure 4.23: Proximity Impact on Response Latency under Flash Crowd, [NP-FDR limit: 34,933 req/s]

Req Rate	*11,235 req/s*				*14,409 req/s*				*30,090 req/s*				*34,933 req/s*			
Latency	μ	50%	90%	σ	μ	50%	90%	σ	μ	50%	90%	σ	μ	50%	90%	σ
Random	2.37	0.64	8.57	5.29												
NPR-CHash	0.61	0.42	1.15	1.76	0.63	0.41	1.08	2.34								
R-CHash	0.73	0.53	1.45	2.10	0.73	0.52	1.38	2.50								
NPLR-CHash	0.53	0.36	0.90	1.75	0.55	0.35	0.91	2.29	1.29	0.61	2.65	3.94				
LR-CHash	0.62	0.45	1.15	1.70	0.64	0.44	1.13	2.56	0.90	0.49	1.73	3.44				
NP-FDR	0.70	0.50	1.45	1.68	0.66	0.45	1.34	1.63	0.81	0.47	1.64	2.55	0.99	0.51	1.92	3.26
FDR	1.22	0.55	1.81	5.71	1.07	0.54	1.67	5.47	1.60	0.66	3.49	5.90	1.84	0.78	4.15	6.31

Table 4.7: Proximity's Impact on Response Latency under Flash Crowds. μ — Mean, σ — Standard Deviation.

4.4 Other Factors

4.4.1 Heterogeneity

To determine the impact of network heterogeneity on our schemes, we explore the impact of non-uniform server network bandwidth. In our original setup, all first-mile links from the server have bandwidths of 100Mbps. We now randomly select some of the servers and reduce their link bandwidth by an order of magnitude, to 10Mbps. We want to test how different strategies respond to this heterogeneous environment. We pick representative schemes from each category: Random, R-CHash, LR-CHash and FDR and stress them under both normal load and flash crowd similar to network proximity case. Table 4.8 summarizes our findings on system capacities with 64 servers. Again, for static strategies, we focus on CHash based schemes, which are typically used in commercial CDN systems.

Redirection Schemes	Portion of Slower Links					
	Normal Load			Flash Crowds		
	0%	10%	30%	0%	10%	30%
Random	9300	8010	8010	11235	8449	8449
R-CHash	20411	7471	7471	19811	7110	7110
LR-CHash	25407	23697	19421	31000	26703	22547
FDR	33237	31000	25407	37827	34933	29496

Table 4.8: Capacity (reqs/sec) with Heterogeneous Server Bandwidth,

From the table we can see, under both normal load and flash crowds, Random and R-CHash are hurt badly because they are load oblivious and assign requests to servers with slower links without explicit monitoring their status, thereby overload them early. In contrast, LR-CHash and FDR only suffer a slight performance downgrade. However, FDR still maintains an advantage over LR-CHash, due to its dynamic expanding of server set for hot URLs.

4.4.2 Large File Effects

As we discussed at the end of section 4.1.3, the worse mean response times of dynamic schemes come from serving large files with a small server set. Our first attempt to remedy this situation is to handle the largest files specially. Analysis of our request trace indicates that 99% of the files are smaller than 530KB, so we use this value as a threshold to trigger special large file treatment. For these large files, there are two simple ways to redirect requests for them. One is to redirect these requests to a random server, which we call T-R (tail-random). The other is to redirect these requests to a least loaded member in a server set of fixed size (larger than one), which we call T-S (tail-static). Both of these approaches enlarge the server set serving large files. However, we do not achieve this through imposing new strategies for large files, we instead simply use a random and fixed server set for them. Therefore, for large files. T-R and T-S are comparable to random and static schemes. We repeat experiments of the 64 server cases in Section 4.1 and 4.2 using these two new approaches, where T-S employs a 10-replica server set for large files in the distribution tail. Handling the tail specially yields slightly better capacity than standard CDR or FDR, but the latency improves significantly. Table 4.9 summarizes latency results under normal load. As we can see, the T-R and T-S versions of CDR and FDR usually generate better latency numbers than LR-CHash and LR-HRW, and also show a much smaller latency standard deviation than plain CDR and FDR, indicating a reduced latency distribution tail. Results under flash crowds are similar. This confirms our assertion about large file effects.

Req Rate	*9,300 req/s*				*18,478 req/s*				*25,407 req/s*				*32,582 req/s*			
Latency	μ	50%	90%	σ	μ	50%	90%	σ	μ	50%	90%	σ	μ	50%	90%	σ
LR-CHash	0.68	0.44	1.17	2.50	0.87	0.51	1.82	2.74	1.19	0.60	2.47	3.79				
LR-HRW	0.68	0.44	1.18	2.50	0.90	0.51	1.89	3.13	1.27	0.64	2.84	3.76				
CDR	1.16	0.52	1.47	5.96	1.35	0.55	1.75	6.63	1.86	0.63	4.49	6.62	2.37	1.12	5.19	7.21
CDR-T-R	0.78	0.52	1.43	2.77	0.76	0.52	1.40	2.80	1.05	0.57	1.90	3.06	1.58	0.94	3.01	3.55
CDR-T-S	0.74	0.52	1.43	2.17	0.72	0.52	1.38	2.44	1.01	0.56	1.93	2.96	1.53	0.68	3.69	4.18
FDR	1.10	0.52	1.48	5.49	1.35	0.54	1.64	6.70	1.87	0.62	3.49	6.78	2.22	0.87	4.88	7.12
FDR-T-R	0.78	0.52	1.43	2.77	0.75	0.52	1.40	2.82	1.01	0.57	1.87	2.98	1.39	0.77	2.82	3.68
FDR-T-S	0.74	0.52	1.43	2.17	0.72	0.52	1.37	2.55	0.98	0.56	1.84	2.95	1.41	0.63	2.88	3.88

Table 4.9: Response Latency with Special Large File Handling, Normal Load. μ — Mean, σ — Standard Deviation.

4.4.3 Dynamic Strategy with Different Hashing Schemes

At the end of Section 2.2.4, we explain that CHash-based CDR and FDR do not perform as well as HRW-based CDR and FDR. Here, we present some of our test results. Figure 4.24 and 4.25 compare the throughput of CHash-based CDR and HRW-based CDR under both normal workload and flash crowds, with the number of servers ranging from 8 to 64. The results for FDR are similar.

From these graphs, we can see using Consistent Hashing to implement our dynamic strategies such as CDR and FDR does not yield as good as performance as using HRW hashing, and sometimes much worse. This is mainly because in HRW hashing, different URLs have a different order of accessing the servers. However, in CHash, the load migration among servers will take a uniform pattern for different URLs; and the less-popular URLs that may have overlapping servers with popular URLs are unlikely to have some other less-loaded nodes in their replica sets. Meanwhile, in CHash, uniform migration not only spreads load to different servers but also increases the working set size on each server. As each server keeps seeing a larger and larger working set size uniformly, cache locality on these servers is reduced and eventually it turns into a round-robin scheme.

4.5 Summary

In this chapter, we present our simulation results on different request redirection strategies. Through these simulations, we demonstrate that our newly proposed dynamic strategies can yield 60%-91% throughput benefit compared to the existing best published schemes, client perceived latency remains low and the system scales well with the number of server surrogates. These improvements make CDNs using our dynamic schemes not

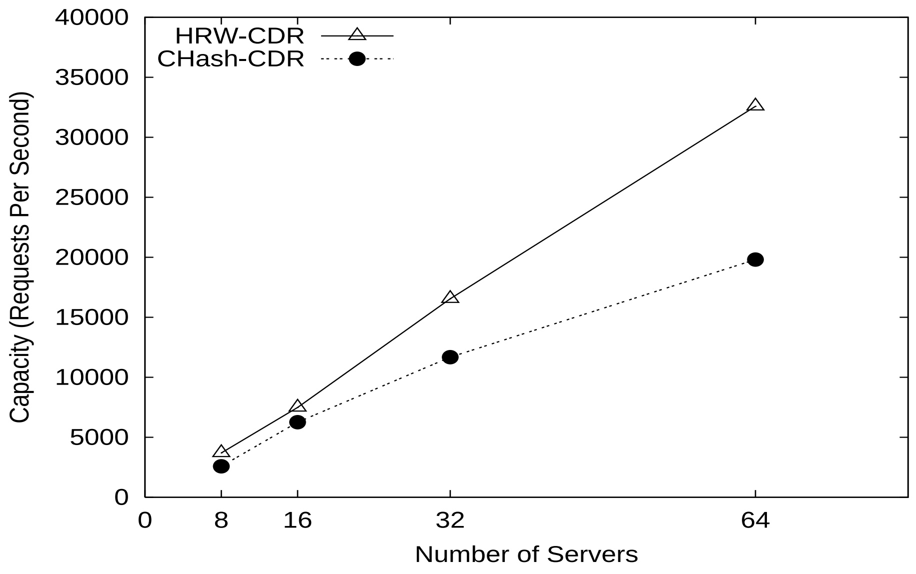

Figure 4.24: CHash-based CDR vs. HRW-based CDR: Throughput under Normal Load

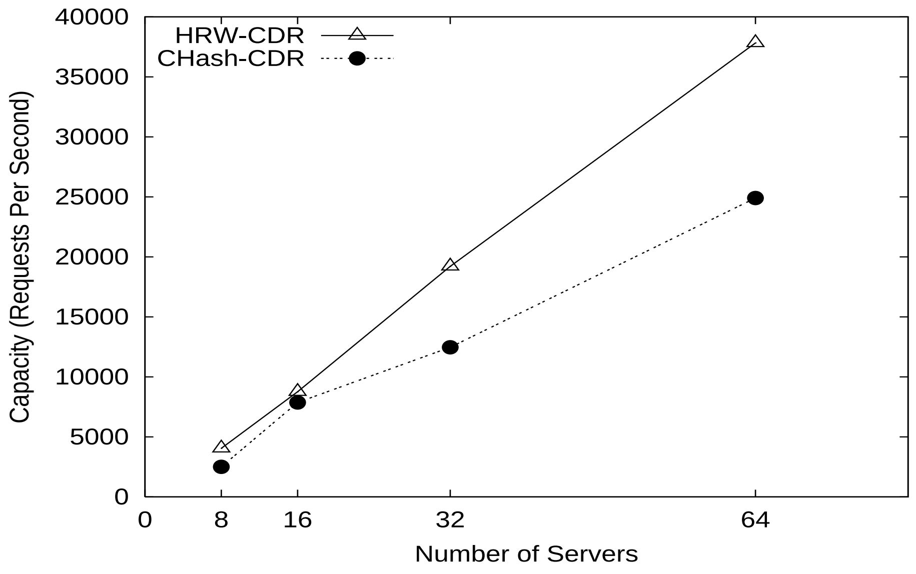

Figure 4.25: CHash-based CDR vs. HRW-based CDR: Throughput under Flash Crowd

only more responsive across a wide range of loads but also more robust in the face flash crowds or DoS attacks.

These improvements come from our success in shifting the system bottleneck from disks to CPUs. This shift fits the technology trends well, where the performance gap between CPUs and disks is expected to become even larger in the future. It suggests that CDR and FDR benefit most from upgrading server capacities. Due to the limitation of our simulation environment, currently we can only simulate 128 servers with relatively small capacity. We expect that CDR and FDR can scale easily to thousands of servers, and with faster simulated machines, the performance gap between these dynamic schemes and the others will grow even larger. When the CDN system grows beyond tens of thousands of servers, we can leverage hierarchy by dividing the servers into geographical or topological regions.

We also investigate the behavior of different strategies under network heterogeneity and how intra-region proximity information can impact the request redirection performance. Again, our dynamic strategies yield the best performance.

Our dynamic redirection strategies improve CDN robustness through efficiently managing CDN resources and maintaining a balance between load, locality and proximity. We believe these successful experiences are not limited to just CDNs. In similar systems, such as peer-to-peer systems and distributed storage systems, there are also constrained resources and conflicting factors. Efficiently managing system resources will be a key in making these systems robust, and we think our algorithms can naturally be extended to those areas.

Chapter 5

CDN Deployment — CoDeeN

Recent developments in building large-scale network testbed [61] and emulation or simulation environments [80, 84] provide us with the opportunities to build an experimental CDN. We choose to construct our prototype CDN, named *CoDeeN*, on top of PlanetLab [61, 63]—an open testbed for developing, deploying and accessing planetary-scale services. PlanetLab provides an overlay network that can be used to both experiment with new network services and let users access them, which meets our goals of building CoDeeN. This chapter first briefly introduces PlanetLab testbed, describes how we built CoDeeN atop it and discusses the problems we encountered.

5.1 PlanetLab

Recently, with the emergence of a new class of geographically distributed network services, such as network-embedded storage, peer-to-peer file sharing and robust routing overlays, testbed environments for designing, evaluating and deploying these services are gaining importance. In [61], it is advocated that the most effective way to facilitate such a new *service-oriented network architecture* is to use an overlay-based testbed, which supports both researchers that want to develop new services and clients that want to use them. The rationale is that the Internet is now less and less influenced by research but increasingly shaped by commercial interests. It becomes harder to test and introduce new services, especially disruptive ones, directly into the Internet. Overlays provide the right opportunity for innovation because they can be quickly programmed without having to compete with the Internet for performance and reliability, etc.

PlanetLab, an open, globally distributed overlay testbed for developing, deploying and accessing planetary-scale network services, is now being built through a multiple-institution effort [63]. As of June 2003, PlanetLab has about 160 nodes dispersed to 65 sites in 16 countries [42]. They span widely across North America, Europe, Asia, and Australia. These nodes are Linux-based PCs and have a diversity of connectivity, not only located in universities and research institutions, but also in co-location and routing centers, and homes (DSL lines and cable modems). Figure 5.1 shows a snapshot of the current nodes on PlanetLab.

We choose to build our academic prototype CDN, named *CoDeeN*, on top of PlanetLab. PlanetLab provides us with realistic network conditions and diverse perspectives on the Internet in terms of connection properties, network presences, and geographical locations. It allows us to build a CDN at a large scale, and conduct different research experiments under various situations. However, CoDeeN has to share the same set of distributed resources on PlanetLab with other services. Therefore, unlike commercial CDNs, CoDeeN does not have fully dedicated and highly reliable resources.

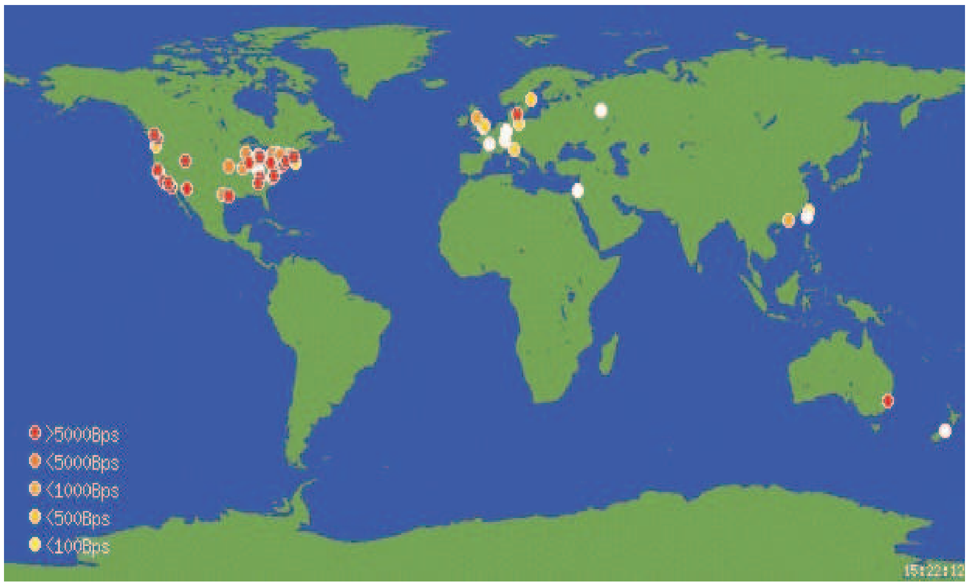

Figure 5.1: Nodes on PlanetLab

5.2 CoDeeN

CoDeeN is an academic prototype Content Distribution Network built on top of PlanetLab. Different from commercial CDNs, CoDeeN is both an experimental system for research projects and also a useful service for users on PlanetLab and beyond. This section first explains our design choices in building CoDeeN, then describes the proxy API we use and how we construct CoDeeN. After explaining how we monitor CoDeeN status, we also discuss a few pragmatic issues and some ongoing projects based on CoDeeN.

5.2.1 Design Choices

A Content Distribution Network has two main components—server surrogates and request redirectors. Server surrogates need to be widely spread and request redirectors need to be able to "intercept" client requests. To build a CDN, we have to implement both of these elements. Nodes on PlanetLab form an overlay network with diverse connection and location properties. These nodes are widely dispersed, and become a natural choice to place CoDeeN server surrogates that cache the content from the origin servers. However, on PlanetLab, we do not have the luxury of controlling the authoritative DNS servers. Therefore, the augmented DNS approach used by commercial CDNs to implement request redirectors does not apply here. We have to use nodes on PlanetLab as CoDeeN request redirectors as well, but in a classical forward proxy setup. Such a proxy-based request redirector scheme implies that we will adopt an opt-in service model, where a user has to point to his browser to one of the proxies of CoDeeN (preferably close to him) if he wants to take advantage of CoDeeN. This minor configuration change does not cause much inconvenience in particular, because most services built atop PlanetLab take some form of an overlay application and probably need some adjustments on the client side if the client wants to use them.

We decided to use PlanetLab nodes to implement both CoDeeN surrogates and redirectors, and then the next problem is how. We can certainly partition all the PlanetLab nodes we employ for CoDeeN into two separate sets, one for server surrogates and the other for request redirectors. However, to better utilize these nodes and consider the fact that each node has plenty of resources, we can also make some nodes both a surrogate and a redirector. We can implement both these functionalities using the same proxy software—configuring it as a reverse (caching) proxy for a server surrogate and as a forward proxy with redirecting capability for a request redirector. For those hybrid nodes, they are then both a redirecting proxy and a caching proxy. Their dual use allows them to map local client requests missed in their own cache to other caching proxies and at the same time to satisfy the redirected requests from remote clients if desired. However, we have to be careful about deploying hybrid proxies. For example, a PlanetLab node with a relatively low bandwidth connection can be used as a request redirector (forward proxy) for local clients, but is not suitable for being a server surrogate (reverse proxy) to serve requests of clients from other locations with faster links.

CoDeeN thus consists of a network of Web caching proxies. This configuration, especially using proxies as redirectors, also conforms to the model we used in our simulations. In addition, proxies can keep track of different activities and generate useful logs/profiles of traffic patterns. Moreover, deploying a network of open proxies at a large scale opens the doors to a lot of interesting research issues such as security and proxy monitoring and management. So far, little has been published in the literature about managing such networks of open proxies.

The remaining problem is to find a proper proxy server to build CoDeeN. One possible choice is to use an open-source proxy server such as Squid [74]. However, the overhead to change proxies like Squid is very high, which requires blending our redirection logic into the proxy core. This implies not just a large amount of work (Squid has 60,000 lines of code), but also means we have to engage in tuning the proxy performance after adding our extensions. Since our focus is not on building fast proxy servers, and we do not want to delve into the details of HTTP processing, what we need is a proxy that has both high performance and a high-level programmable interface allowing us to easily modify the request handling process. Fortunately, iMimic donated its DataReactor [38] proxy servers to PlanetLab, which possess exactly these two properties. Below, we discuss how we use DataReactor proxy servers to construct CoDeeN.

5.2.2 CoDeeN Implementation

As noted above, a proxy-based approach to build CoDeeN requires a high performance proxy server that can be easily customized according to our needs, such as redirecting requests and caching content. iMimic's DataReactor proxy server is such a candidate. It is a full-fledged proxy capable of handling multiple protocols, which relieves us from dealing with the gory details of HTTP. More importantly, it has a flexible and efficient application programming interface (API) for customizations [59].

This API facilitates the development of customized content adaptation, content management and specialized administration features. It provides the infrastructure required to process HTTP requests and responses at a high level. This API allows user modules (add-on functions) to be directly loaded into the core of the proxy cache and run services either on the proxy system or on a separate server as desired. With an API-enabled proxy server, the client sends a request to the proxy and the proxy interprets the request and compares it with the content it caches locally. If the request is a cache hit, the proxy gets the content from its local storage system and returns it to the client. Otherwise, the proxy has to contact the remote server to get the most up-to-date version of the requested object and possibly stores a copy for future similar requests. During these interactions, there are many places where special treatments, such as content adaptations, can be launched. This is achieved through the API allowing user specified modules to register *callbacks* to the proxy. These callbacks are special functions with defined inputs and outputs that are invoked by the proxy on events in the HTTP processing flow. Examples of HTTP processing events include the completion of receiving request or response headers, the arrival of request or response body content, the completion of the entire HTTP transaction, and timer events. To handle these events, user modules need to specify a group of callback function pointers to the proxy core. If a module is interested in a certain event (callback point), it can specify a non-NULL function entry for that event. Otherwise, a NULL function pointer means that the module wants to receive no notification for the corresponding event. The API also allows the module to keep its own state in a private data structure and clean it up if desired when the transaction is complete. In addition, the API allows the module to access configuration fields specified in one or multiple configuration files. These configuration fields can be used to set different parameters in the module at runtime without modifying the module's source code.

To implement CoDeeN, we install DataReactor proxy servers on selected PlanetLab nodes, and use them both as request redirectors and server surrogates as much as possible. Figure 5.2 presents a very high level overview of different scenarios. If a proxy node (such as Proxy A and Proxy E) does not have a well-provisioned network connection, we keep it simply as a redirector, a forward proxy that caches recently-accessed pages for local users. For example, a local client sends a request to Proxy A and the requested content is missed in Proxy A's cache, Proxy A redirects the request to some server surrogate on CoDeeN, but other proxies do not forward requests to Proxy A. Otherwise, a proxy without connectivity constraints also acts as a caching surrogate (reverse proxy) for all the CoDeeN users, such as Proxy B, C and D. As seen in Figure 5.2, if a local client's request causes a miss in Proxy B, Proxy B redirects this request to some surrogate, and the chosen surrogate could be a remote one or a local one in Proxy B itself, depending on the URL and the redirection strategy. In other words, a CoDeeN proxy is at least a redirector. CoDeeN users need to connect to one of the CoDeeN proxies, and their requests will then be intercepted and handled by the redirector. Logically, a CoDeeN proxy can include two entities, a redirector and a surrogate. But in practice, they are blended into one single proxy server loaded with our redirector module.

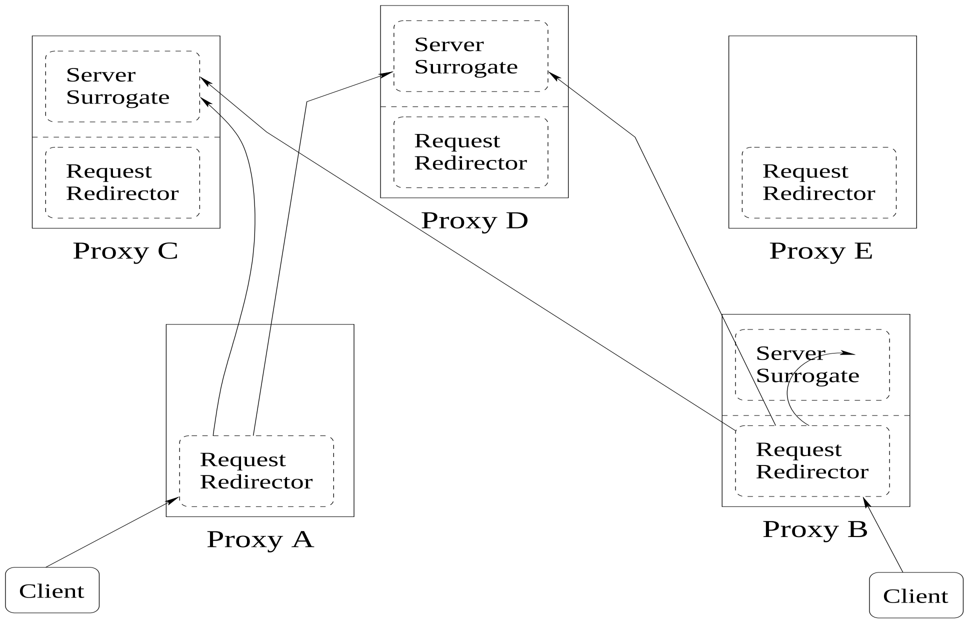

Figure 5.2: CoDeeN Proxy Server Logical Setup

Caching is a built-in functionality of the DataReactor proxy servers. When a surrogate proxy is chosen to be responsible for the requested URL, it is able to get the content from the origin server and cache it if appropriate. Therefore, our main focus here is not on the surrogate, but on the redirector's logic. To separate concerns, we choose to develop a general redirector infrastructure, and keep the redirection strategies independent of the redirector mechanisms. We present to the redirector an abstraction of a redirection strategy in the form of a server selection function. This function takes a URL and a list of surrogates as inputs and returns an appropriate server surrogate for that URL. By doing this, we can easily replace different redirection strategies without changing the general redirector infrastructure. In fact, we make the redirection strategy as a configuration parameter, allowing us to experiment with different strategies we have studied in Section 2.2 conveniently.

With separate redirection strategies and a flexible proxy API, the redirector logic seems simple to construct at first. However, our real implementation turns out more involved. As pointed out in Section 5.1, CoDeeN does not have dedicated and reliable resources, therefore, the proper functioning of CoDeeN relies more on our ability to monitor the health of CoDeeN proxy servers and detect abnormal conditions. We spend quite some effort on building effective monitoring facilities for redirectors. On the other hand, CoDeeN is a network of open proxies that do not restrict their client population to PlanetLab participant sites. This feature opens a number of security issues for CoDeeN. We also employ different mechanisms to keep CoDeeN from being abused. More detailed information about CoDeeN monitoring can be found in Section 5.2.3 and security issues are covered in Section 5.2.4. Here, we present the general control flow of CoDeeN redirectors.

When a redirector module gets loaded, its initialization function first starts the bootstrap process, with the help of several configuration files. It prepares for CoDeeN monitoring facilities, which includes opening a UDP socket for heartbeat messages and spawning a helper process to test HTTP/TCP level connectivity to other redirectors. In order to get notified when the UDP socket is ready, it needs to register the interest in that socket to the proxy core. Meanwhile, it also initializes request rate estimation/control and other logging data structures. The actual bookkeeping and monitoring activities will be addressed more thoroughly later on. To control these monitoring and logging behaviors, an API-provided timer is employed to perform various activities at different time intervals. Another task of the initialization is to get the list of the server surrogates currently available on CoDeeN. This list is now obtained through reading from a configuration file distributed among all CoDeeN redirectors. We are also investigating other ways of disseminating and updating this information. The server list also implicitly decides whether to use this local proxy as a server surrogate, depending on whether or not it appears on the list. Besides knowing all the available server surrogates, a redirector also needs to select its server selection function, which is decided through reading the redirection strategy option from its local configuration file.

On receiving the full URL and headers of a request, the redirector module does various sanity checks on the client IP address and the URL, mainly for security and robustness reasons. We first check to see if the client IP address is from some known attackers or bad users. If that is the case, an error response will be returned to the client. This protects CoDeeN proxies from being disrupted by malicious traffic. The request header function also decides if the client is from the local domain, other PlanetLab sites or outside world. This classification information will later be used for making request redirection decisions. Next, the module examines the request headers and ignores any ill-formed request. If the IP address and URL are normal, the redirector decides if the request has been forwarded from a peer proxy on CoDeeN. The way requests get forwarded from one proxy to another on CoDeeN is through passing along with the modified request two extra header fields starting with an `''x''` that bear the original request information. When a redirector decides that a particular request needs to be redirected, it rewrites the URI and the "host" header field of the request such that the changed request is destined to the target proxy. It also adds to the forwarded request an extra field to indicate the original URI and another extra field to indicate the original host. According to HTTP protocol [31], any header field starting with an `''x''` is not a standard HTTP header. So these special header fields will not interfere with normal HTTP transactions. On the target proxy side, if it sees these two special fields, it knows that the request has been forwarded to it and recovers the original request from these two special headers. As long as the target proxy behaves as a server surrogate, it has to fulfill this request on behalf of its peer proxy and contact the origin server if needed. If there are no such special headers, the request must be directly sent from a CoDeeN user connected to this proxy. The redirector then takes further actions based on the client address classification information and the target website. Currently, a client outside of PlanetLab is given less freedom, only allowed for HTTP GET requests and more constrained on what sites it can access and how much bandwidth it can consume. Some restricted or private sites will only be accessible to local users. These measures are used to protect CoDeeN from being abused, and we discuss more about these rationales in Section 5.2.4. If the request does not need any special treatment, the server selection (request redirection) function will be called at this time to decide which surrogate is the best for serving this request. If the chosen surrogate is a remote one, this redirector needs to modify the request, replacing the first line and "host" field with new ones that destined to the target surrogate proxy and adding extra `''x''`-starting fields to preserve the original request. Using these extra header fields proves to be fairly easy to implement request redirection among CoDeeN proxies, however, we are also working on forwarding schemes with stronger authentications.

When a request is sent out to a remote node on a local cache miss, we do some load status bookkeeping. The remote node can be either a remote proxy in the case of a request redirection, or the origin server if the local proxy is the server surrogate responsible for the request. We count the number of requests sent to each remote proxy and use it as a local load approximation as we did in our simulations.

We keep track of the bandwidth usage for each request when the response header and body come back. If the initial connection with the remote node fails, we can discover it inside the response header function through a special status flag set by the API.

We have a 1-second timer in each CoDeeN redirector module. In the timer handler function, from time to time, we dump performance statistics to files and re-read configuration files if there is any change. We also update request rate accounting data, and launch heartbeat messages and HTTP/TCP connectivity tests. Redirector specific, coarser-grained timers are thus implemented using this 1-second timer.

Our entire redirector module now consists of about 5,000 lines of code, including the monitoring routines discussed in next section. Currently, DataReactor proxies loaded with our redirector module have been installed on over 40 PlanetLab nodes, using a slice (princeton9) of PlanetLab resources. CoDeeN is already fully operational and evolving quickly. Table 5.1 gives a list of 40 nodes that are public to CoDeeN users as we announced CoDeeN's alpha test in July 2003. Proxies on these nodes are configured as both redirectors and surrogates.

5.2.3 Monitoring CoDeeN

As we see from Table 5.1, CoDeeN now has one instance running per site for most of the North American educational sites that host PlanetLab, and it keeps expanding to other PlanetLab nodes. Operating and managing such a network of web proxies requires careful monitoring of the health of each node, especially when CoDeeN shares the same set of distributed resources with other services on PlanetLab. An overloaded or dead proxy should certainly be avoided by other peers. Monitoring current status of each CoDeeN node provides useful information for CoDeeN's operation, moreover, accumulated history of status can be a good indicator of future behaviors. To avoid a centralized scheme, each CoDeeN instance operates independently, and uses heartbeat messages to try to determine what other nodes are alive and worth using. The load information is gathered from time to time, and stored into the local disk on each proxy every 30 seconds. The stored information can later be reloaded when the proxy reboots, avoiding losing all the performance history after a node becomes off-line for some time. The summary of this information is also published and updated automatically on the CoDeeN central status page [53]. This section discusses how each CoDeeN redirector monitors its own proxy state, the state of the node, and gathers information about other nodes.

A redirector in a CoDeeN proxy collects and maintains some information about the proxy itself, such as how long it has been alive (proxy uptime). In the API timer handler function, we also count the average number of requests per hour the proxy handles. This number gives an approximation of how busy the proxy is recently.

To collect the load information of the local CoDeeN node, a redirector uses several system tools, such as “uptime”, “top” and “vmstat’, and also gets help from some specially-designed self-testing routines. Using “uptime”, we know how long the actual node has been alive. We obtain the system load averages for the past 1, 5 and 15 minutes from “top”. We then use the maximum of these 3 numbers as the representative system load value, which indicates how many active processes are competing for CPU. Another load indicator is how much CPU time is spent in the operating system. We gather readings of system CPU utilization through “vmstat” and report the maximum number for the last 3 minutes. A value over 95% implies that the system is not stable at that time. This is based on our empirical experiences that while some applications do spend most of their time in the OS, few spend much more than 90%. Even with some PlanetLab special accounting routines in the kernel, the system CPU time can be high, but over 95% is very rare. Besides recording these numbers reported by the system, a redirector also conducts its own tests on some of the local node resources. For example, since the proxy needs to open a number of sockets for accepting and redirecting client requests, free sockets (file descriptors) become an important resource. Meanwhile, since each proxy shares

Node Name	IP address
planetlab1.cs.ubc.ca	142.103.2.1
planetlab1.cs.arizona.edu	150.135.65.2
planetlab2.millennium.berkeley.edu	169.229.51.251
planetlab-1.cmcl.cs.cmu.edu	128.2.198.188
planlab1.cs.caltech.edu	131.215.45.71
planetlab1.comet.columbia.edu	128.59.67.200
planetlab1.cs.cornell.edu	128.84.154.49
planetlab1.cs.duke.edu	152.3.136.1
lefthand.eecs.harvard.edu	140.247.60.123
planetlab2.cnds.jhu.edu	128.220.231.3
kupl1.ittc.ku.edu	129.237.123.250
planetlab1.lcs.mit.edu	18.31.0.190
planetlab2.cs.northwestern.edu	129.105.44.81
planet1.scs.cs.nyu.edu	216.165.109.81
planetlab-1.cs.princeton.edu	128.112.152.122
planetlab2.cs.purdue.edu	128.10.19.53
ricepl-1.cs.rice.edu	128.42.6.143
planet2.cs.rochester.edu	128.151.65.102
planet1.ecse.rpi.edu	128.113.50.102
planetlab2.rutgers.edu	165.230.49.115
planetlab-1.stanford.edu	171.64.64.216
pl1.ece.toronto.edu	128.100.241.67
planetlab1.cs.ucla.edu	131.179.112.70
planetlab1.cs.uchicago.edu	128.135.11.149
planet1.cs.ucsb.edu	128.111.52.61
planetlab1.ucsd.edu	132.239.17.224
planetlab1.cs.uiuc.edu	192.17.239.250
planetlab1.netlab.uky.edu	206.240.24.20
planetlab1.cs.umass.edu	128.119.247.210
planetlab1.eecs.umich.edu	141.213.4.201
planetlab1.cis.upenn.edu	158.130.6.254
planetlab2.flux.utah.edu	155.98.35.3
planetlab1.csres.utexas.edu	128.83.143.152
pl1.cs.utk.edu	160.36.57.172
planetlab1.cs.virginia.edu	128.143.137.249
planetlab01.cs.washington.edu	128.95.219.192
planetlab1.cs.wayne.edu	141.217.16.210
planetlab1.cs.wisc.edu	198.133.224.145
vn1.cs.wustl.edu	128.252.19.20
planet1.cc.gt.atl.ga.us	199.77.128.193

Table 5.1: Current Public CoDeeN Nodes on PlanetLab

resources with other services multiplexed on the same PlanetLab node, it is possible that the system runs out of free file descriptors sometimes. If that happens, the proxy should not be regarded as a good target for request redirection. To determine whether or not a proxy is suitable to be a recipient of requests, we test to see if we can open a number of sockets (e.g. 50) from time to time. If there is a problem during this test, we will record it as a file descriptor exhaustion incident, otherwise it is simply a success. The interval between file descriptor tests is configurable and currently set to be 2 seconds. We keep a history of last 32 tests. If all tests succeed for the past 32 tests, we determine that the proxy is fit, at least from the free sockets point of view (FdTst). Otherwise, the proxy is deemed to be not good for receiving requests due to running out of file descriptors. At this point, additional local system resource testing mechanisms are also under investigation.

To monitor the reachability and gather load status of other peer proxies, each redirector in a CoDeeN proxy employs the following two mechanisms—a UDP-based heartbeat to exchange load information and a wget helper process to test HTTP/TCP level connectivity. Here, the peer proxies refer to those server surrogates listed in the CoDeeN surrogate configuration file. We use UDP-based heartbeat because even when local file descriptors are exhausted, the UDP port still works, and we can still exchange information with peers. On the other hand, UDP packets can be dropped or lost, we use this loss or drop rate as an estimate of congestion. However, TCP tries to send packets reliably and we cannot gather packet loss information.

- **UDP-based heartbeat.** As discussed earlier, at initialization, each proxy maintains an open UDP socket for sending and receiving heartbeat messages. When a timer expires (every second), a proxy (*A*) sends a heartbeat request message to one of its peer proxies (*B*). When the peer (*B*) receives this request message, *B* responds to *A* with a UDP acknowledgment indicating it is alive and also piggybacking *B*'s own load information. The piggybacked load information includes *B*'s average load (LoadAvgs), system time CPU percentage (SysPtCPU), whether file descriptors have been exhausted recently (NoFd), proxy and node uptime (NdUptime, PxUptime), and average requests handled per hour (ReqsHour). If *B* is configured as or believes itself that it should not handle redirected requests, it can also explicitly say so in the acknowledgment. However, this acknowledgment heartbeat message can get delayed or lost. To account for these events, we record the sending time of each heartbeat request message and also maintain a history of late ACKs (LateAcks) and missed ACKs (MissAcks) for each peer. Over time, proxy *A* sends heartbeat messages to all its peers and learns whether or not each peer proxy is alive or overloaded. By doing this, *A* obtains a view of all CoDeeN proxies and decides whether or not to avoid redirecting requests to certain peers. This UDP-based heartbeat scheme also permits a summary request, which will trigger the receiver to send a summary of its local view of the entire CoDeeN network to the sender. This summary can be useful for both quick load information exchanges and external monitoring. Figure 5.3 depicts the format of heartbeat messages.

- **WGET helper process.** The above heartbeat scheme provides an internal ping facility for peer proxy monitoring. In addition to this UDP-based approach, we also use external tools to test HTTP/TCP connections and provide this information to our proxy modules. This is because UDP-based heartbeat cannot capture certain types of failures. For example, sometimes, due to site administrators' port filtering on TCP connections, although UDP packets can be exchanged without obstruction, HTTP/TCP level connection to the proxies cannot be established. The external tool we choose is "GNU wget" [34], which is a free software package for retrieving

```
#define HBI_QUERY        0x000 /* must be zero */
#define HBI_ACK          0x001
#define HBI_FDEXHAUST    0x002
#define HBI_DONOTSEND    0x004
#define HBI_SUMMARYQUERY 0x100
#define HBI_VERSION      11

typedef struct HeartbeatInfo {
  int hi_statusVec;   /* HBI_xxx code defined above */
  int hi_versionNum;
  int hi_loadAvg;     /* load from uptime, rounded up */
  int hi_sysPctCPU;   /* how much cpu is system using */
  int hi_proxUptime;  /* how many seconds proxy has been up */
  int hi_nodeUptime;  /* how many seconds node has been up */
  int hi_reqsPerHour; /* not a true count - just an approx */
} HeartbeatInfo;
```

Figure 5.3: Heartbeat Message Format

files using protocols like HTTP, HTTPS and FTP. Here, we only use wget's HTTP capability. As described earlier, in each CoDeeN proxy's initialization phase, we spawn a helper process from our redirector module. The helper process then communicates with the main proxy module through a UNIX socket [75]. It reads wget-test instructions from the main proxy module through the socket and launches another child process of its own executing the corresponding wget command. The helper process uses a timer to keep track of the progress of its wget child process and kills it if it is still trying when the timer expires. Whether the wget command is successful or not, the helper process reports the result to the main proxy module. The way we use wget to test whether a proxy works properly is as follows. We maintain a fake webserver listening on a special port in each CoDeeN proxy, and this fake webserver is used solely for the wget measurement. Instead of launching wget directly toward the fake webserver of a target peer proxy, we use a level of indirection and configure wget to use one of CoDeeN's proxies as its "http-proxy". This can be easily achieved through setting the corresponding environment variable of wget. For example, we can have wget on *A* to get a web page from a target fake webserver on *C* through an "http-proxy" *B*. Here, *A*, *B* and *C* are all CoDeeN proxies. And the webpage we ask for from *C* is not cacheable, so it will not be directly satisfied or forwarded at *B* and has to be fetched from *C* by *B*. Thus, without disturbing other outside nodes, we are able to test from *A* whether *B*'s proxy is working and whether *C* is reachable. Both the wget proxy and the wget target will be iterated through the server surrogate list over time. We keep a history of the failed past wget tests for each peer both as a wget proxy (WgetProx) and as a wget target (WgetTarg). Combining this wget test information with the previous heartbeat information, we then determine whether or not a peer proxy should be used for receiving redirected requests.

To summarize, we present a snapshot of the collected information at one of the CoDeeN proxies (planetlab-1.cs.princeton.edu) below. The first part includes its own load status and its local view of other CoDeeN proxies, and the second part is how other proxies on CoDeeN think of this proxy.

```
planetlab-1.cs.princeton.edu
FdTstHst: 0x0
ProxUptm: 36707
NodeUptm: 111788
LoadAvgs: 0.18 0.24 0.33
ReqsHrly: 5234 3950 0 788 1004 275 2616
SysPtCPU: 2 2 1 3 2 4
```

```
planetlab-1.cs.princeton.edu’s view of CoDeeN
Liveness: ..X.. ..X.. ..... .X.XX ..... ...X. ..... .....
MissAcks: 10w00 00001 00000 0w066 00010 000v0 00020 00000
LateAcks: 00000 00000 00000 00000 00000 00000 00000 00000
NoFdAcks: 00000 00000 00000 00000 00000 00000 00000 00000
VersProb: 00000 00000 00000 00000 00000 00000 00000 00000
MaxLoads: 41022 11111 11141 20344 11514 14204 11111 11011
SysMxCPU: 81011 11111 11151 10656 11615 15564 11111 11111
NdUptime: kk0mh lmkmk mhemk m0kmm mllkm kmmfl mmmmj mmllm
PxUptime: gg0gg ggggg ggdgg g0ggg ggggg gggdg gdggg ggggg
ReqsHour: ed0dd ddded ddccd d0dde dcdcd ecdbe cccdc ccccc
WgetProx: 00w00 00100 00010 0w110 00000 000s0 00010 00001
WgetTarg: 11w11 10301 01021 1w220 00111 101t0 11121 00011

CoDeeN’s view of planetlab-1.cs.princeton.edu
Liveness: ..... ....X ..... ..... .X.X. ..... ..X.. ...
MissAcks: 10000 00000 01000 00000 02061 00000 00v00 000
LateAcks: 00000 00000 00000 00000 00000 00000 00000 000
NoFdAcks: 00000 00000 00000 00000 00000 00000 00000 000
WgetProx: 00000 00000 00000 00000 00120 00000 00t00 000
WgetTarg: 00111 11013 11111 10010 10221 11100 01u01 101
```

Most of the fields are explained earlier. In order to have a compact representation of the status of many nodes, sometimes, we use base-32 (0–9a–w) numbers, with 0 being the lowest value and w being the highest. For example, MissAcks, LateAcks, NoFdAcks, MaxLoads, WgetProx and WgetTarg are shown as a base-32 number. NdUptime, PxUptime and ReqsHour for peer proxies are first converted in to log base 2 numbers and then also reported in base-32 format. For SysMxCPU, values below 90% are rounded to the higher multiple of 10% and shown divided by 10. So, a value of 6 means that 51-60% of the CPU time is being spent in the system. Values that are 90% or higher are shown starting with the letter "a", incremented by one character for each percent, with 90%=a, 91%=b, 92%=c, etc.

The Liveness row has one character for each CoDeeN node, and shows whether this node considers that node to be a viable redirector. If the node is considered unusable, an "X" is shown in its column. Otherwise, a "." indicates liveness. The reasons for a node being considered unavailable are multiple. Nodes that have not responded to heartbeats recently are considered dead. Likewise, nodes that are behind others in terms of recent wget tests (WgetProx and WgetTarg failure counts exceed other nodes by more than 2) are also considered not viable. Finally, nodes that show a system CPU time above 95% are similarly skipped.

As we can see from the snapshot, the liveness status is consistent with other metrics. Those nodes that are considered bad by planetlab-1.cs.princeton.edu, typically either have a high number of missed ACKs, or fail to meet the safe threshold on other aspects such as WgetProx and WgetTarg. The bottom half of the snapshot shows other's view of planetlab-1.cs.princeton.edu. Since at the time we took the snapshot, two nodes were physically down without any connections, we can only show 38 nodes here. Although planetlab-1.cs.princeton.edu seems OK locally, some of other nodes are still avoiding it.

This load/performance information is not only critical to CoDeeN's smooth operation, but also valuable for monitoring the status of PlanetLab nodes in general. For example, we have discovered that wide disparity exists between nodes. At the snapshot time, out of the total 40 nodes running on CoDeeN, most nodes considered about 33–35 other nodes to be viable. However, some nodes saw much less—only about 9 viable nodes. Load status on different nodes also differed a lot, with most of the live nodes on CoDeeN seeing load averages of less than 1, but some seeing a load average of 10–11. And some nodes also fluctuated between fine and bad frequently. Some nodes had few problems responding to the UDP heartbeat, but failed the wget tests. These observations on one hand prove that our multiple-test approach is quite useful in discovering different abnormalities (CoDeeN performs well and successfully avoids bad nodes with our monitoring facilities in place), but on the other hand also motivates us to study the problem of node monitoring more thoroughly. We keep collecting more performance data and developing tools to analyze and visualize them. Currently, these monitoring activities incur negligible overhead on CoDeeN. However, when the number of CoDeeN nodes further increases, this application-level n-to-n ping scheme may not scale very well. One possible solution is to divide CoDeeN nodes into regional cliques and organize them into a hierarchy. Within a clique, we can still use frequent heartbeats to monitor each proxy. We then can use a dissemination tree to pass load summaries among different cliques and eventually propagate the complete load information to all proxies. We are now working on improving our monitoring routines and reducing their overhead.

5.2.4 Pragmatic Issues and Ongoing Research

CoDeeN uses a network of programmable proxies running on PlanetLab nodes to intelligently route web traffic requests to offload traffic from origin servers and balance load across the network. In deploying the initial prototype of CoDeeN, we found several security-related vulnerabilities, where CoDeeN nodes were used to help carry out malicious activities. We also received complaints from PlanetLab administrators indicating CoDeeN was abused. This is mainly because CoDeeN nodes operate as “open” proxies, which do not restrict their client population to PlanetLab participant sites. During the seven weeks of CoDeeN's trial period, although we did not publicly announced it, CoDeeN attracted over 24 million requests from more than 59,000 unique IP addresses, and our request rate is growing, as shown in Figure 5.4. This was well beyond our expectations. Part of the reason is that some users kept probing for open proxies. Meanwhile, some web sites listed open proxies and sold additional software to make testing and using proxies easier and CoDeeN nodes were found to be advertised on these sites. However, some of the outside users of CoDeeN tried to abuse CoDeeN in a number of different ways.

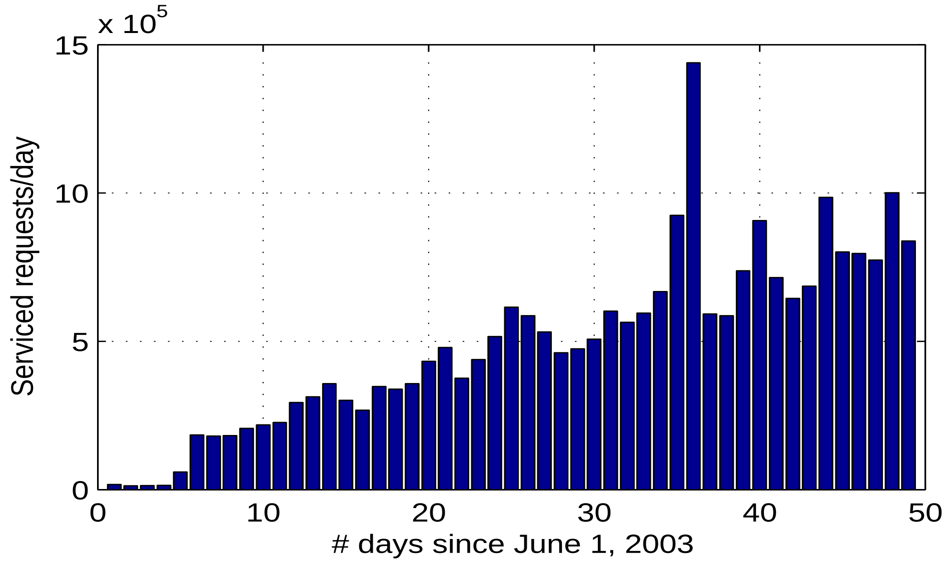

Figure 5.4: Daily traffic on CoDeeN.

We analyzed the security issues and overhauled the initial CoDeeN system. However, the list of the problems is still growing and our work on handling these issues is still underway. Here we only discuss how we address some of issues and mitigate the corresponding problems. It also helps to understand some of the implementation choices described in previous section. In the end, we describe some of the ongoing research based on CoDeeN to conclude this section.

Denial of Most HTTP CONNECT Requests

The CONNECT method in HTTP is used to tunnel TCP for end-to-end services such as SSL when proxies also serve as firewalls. The client specifies a remote machine and a port number, and the proxy creates a new TCP connection and forwards data in both directions. Spammers search for proxies that allow this TCP tunneling via the HTTP CONNECT method. Examples include connecting to port 25 of open SMTP (Simple Mail Transfer Protocol) relays or to port 6667 of IRC (Internet Relay Chat) network servers. In both cases, the spammer sends spam through a weakly-authenticated system, and hides his/her identity by going through the proxy. To the victim, the proxy appears to originate the connection.

To address this problem, we classify clients by their IP addresses, and determine which clients are from within PlanetLab-affiliated sites and which are not. Clients from outside PlanetLab are not allowed to use the HTTP CONNECT method. Clients attempting to connect to port 25 are optionally sent to a local SMTP honeypot, where their activities are logged.

More Restrictive Handling of Non-GET Requests

The POST request is used for submitting web-based forms, including cgi-to-email gateways. These gateways use a CGI program called *formmail* to allow users to mail web-based forms to the site's operators, usually for feedbacks. And *formmail* often encodes the destination email address in one of the form's hidden input. Although regular web browsers will send along only the email address specified in the form, it is easy to forge requests with different email addresses. We observed the proxy being used for high rates of such formmail submissions by spammers targeted to their victim email addresses. In reaction to this problem, clients from outside of PlanetLab are only allowed to issue requests using the HTTP GET method. This will limit their ability to complete certain forms, including those used for authenticated login to sites such as eBay. Clients from within PlanetLab are not restricted.

Rate-limiting of Non-PlanetLab Clients

We have observed suspiciously high rates of requests from some clients, seemingly from automated services. We do not know that these are malicious, but one can imagine that some web site receiving high rates of requests over a long time may feel threatened.

We can specify the maximum number of requests per minute, per hour, and per day from any IP address. We currently default to use the following thresholds: 3 reqs/sec per minute, 0.5 reqs/sec per hour, and 0.1 reqs/sec per day. These limits are configurable on a per-node basis, and are designed to allow bursts of activity over short periods of time.

Protection of ”Private” Information on PlanetLab Sites

Many organizations such as universities have web pages that are not accessible from outside the range of IP addresses associated with the university. While the PlanetLab AUP (Acceptable Use Policy) [63] explicitly forbids using PlanetLab nodes to access such information, few technical barriers are present to enforce such behavior. As a result, connections from the proxy appear to originate from acceptable IP addresses, and such content may be visible via CoDeeN.

We now have information on the range of IP addresses affiliated with each PlanetLab site, and use this information to determine if a client is ”local” to that site. If such a client attempts to access a local page, it is permitted. If a non-local client attempts to access such a page, the request is forwarded to a CoDeeN node at another site, thereby de-escalating its privilege.

Protection of Weakly-authenticated Subscription Services

In the past, open proxies have been used to access large amounts of content at subscription-only online journals and databases. These sites typically have the IP address ranges of the universities that have subscribed to them, and use only this information in authenticating clients. A few accesses to these sites via the CoDeeN prototype occurred, and such behavior indicated that a more coordinated large-scale access may have been planned.

We have a database of such sites, and they can only be accessed by clients local to each PlanetLab site. Currently, if a client from outside of PlanetLab attempts to access such sites, their access is denied. In the future, we may forward such requests to sites without subscriptions, de-escalating them similar to the way we handle ”private” pages.

Ongoing Research

In summary, many of the issues listed above are related to CoDeeN’s nature as a network of open proxies. And these open proxies reside in a semi-hostile environment, with PlanetLab local users more trustworthy. How to manage this network of open proxies in such a unique environment presents a very interesting and challenging research problem. Although our immediate goal is to secure CoDeeN from the problems associated with open proxies, our mechanisms may be also helpful for other research projects that allow “open” access to Web resources.

Our current effort solves or alleviates some of the CoDeeN problems through rate-limiting and privilege separation. We expect to see more security and management related issues and plan to address them in a more systematic and automatic fashion. Currently, there are a couple of research projects carried out along this direction. For example, we are looking for ways to distinguish between requests sent by human beings or those generated by scripts, in an effort to early detect some organized and automated spamming or attacking activities. We are also investigating how to use SMTP honeypot on CoDeeN to defend against mail spams.

Other interesting issues include to improve and better utilize our CoDeeN monitoring facilities and see how they can cooperate with other monitoring schemes on PlanetLab. Certainly, we can use CoDeeN to test different redirection strategies under various load conditions to further verify our simulation results. We plan to use other PlanetLab nodes as clients to stress CoDeeN proxies. However, we still need to exercise caution not to disturb other services/projects on PlanetLab too much, since PlanetLab is a shared testbed. In order to create sufficient amount of request traffic, we may need to scale-down CoDeeN proxy servers capacity or conduct the experiments at a less busy time with the coordination of other PlanetLab services. Another factor to consider is that on CoDeeN, most of its proxies behave both as a redirector and a surrogate, so we can not control what content will be served from the surrogates (content is cached on-demand) with unconstrained client populations. While on commercial CDNs, surrogates are only responsible for content belongs to certain CDN's customers (content providers) and redirectors simply map requests.

This section describes how we deploy CoDeeN on top of PlanetLab, and reports the current status of CoDeeN including the problems we have encountered. Although some of the problems stem from our using open proxies to build CoDeeN, our solutions can potentially benefit other "open" systems as well. We have already gained valuable experience on managing and monitoring an operational CDN from CoDeeN. As CoDeeN becomes more stable, we are confident to have more opportunities to carry out different experiments and expand our knowledge about CDN and open proxies.

Chapter 6

Conclusions

To conclude this thesis, we summarize our three main technical contributions in this research. First, we explore CDN redirection strategy design space and propose a new class of dynamic redirection schemes that yield much better performance. Second, we develop a hybrid network-server simulator and conduct large scale detailed end-to-end simulations to study various redirection strategies. Third, we deploy an academic prototype CDN on PlanetLab. These contributions support our claim that through efficient resource management we can make network services more robust in the face of different traffic. In the end, we also outline our plans for continuing research.

6.1 Technical Contributions

This thesis addresses the problem of improving network service robustness from the standpoint of efficient resource management. We study two main robustness challenges—flash crowds and Denial of Service (DoS) attacks. We find that in order to be immune from these challenges, a network service needs to possess two important properties—

completeness and generality. In other words, we have to protect all the resources of a network service and should handle faults or attacks within the same framework used during normal operations. Following these two principles, we use Content Distribution Networks as an example and demonstrate that through a unified resource management scheme, we can build a network service that not only yields better performance under regular traffic, but also becomes more robust in the face of abnormal events such as flash crowds and DoS attacks.

More specifically, in designing a request distribution mechanism that manages all the CDN resources efficiently to improve CDN robustness, we make the following three main contributions. First, we explore the design space of redirection strategies employed by CDN request redirectors, and define a class of new algorithms that carefully balance load, locality, and proximity. Through detailed end-to-end simulations, these new algorithms are demonstrated to be much more effective on improving CDN robustness when compared with published CDN technology. Second, we develop a novel hybrid network-server simulator that provides detailed modeling at both packet level and operating system level. We optimize the simulator and use it to conduct extensive large-scale simulations to evaluate various redirection strategies. Third, we build an academic prototype CDN—CoDeeN on PlanetLab. CoDeeN provides us with valuable experience on building and managing an operational CDN. We use CoDeeN to conduct many research projects and plan to further verify our simulation results on CoDeeN.

Request Redirection Strategies

In this thesis, we have studied a number of redirection strategies, from the baseline case Random, the best published CDN technology based on a fixed set of servers to service each URL, to our newly proposed schemes that dynamically adjust the servers responsible for each URL according to the system's load, locality and proximity.

In exploring the design space of request redirection strategies, we identify the benefits and disadvantages of different strategies, and also demonstrate that our improved dynamic request redirection strategies can effectively improve CDN robustness by balancing locality, load and proximity. Detailed large-scale end-to-end simulations show that even when redirectors have imperfect information about server load, algorithms that dynamically adjust the number of servers selected for a given object, such as FDR, allow the system to support a 60-91% greater load than best published CDN systems. Moreover, this gain in capacity does not come at the expense of response time, which is essentially the same both when the system is under flash crowds and when operating under normal conditions. We also test these strategies under various request traffic patterns and different hot-spot situations. Again, using our new schemes, CDN systems show better flexibility and adaptiveness under various conditions. Finally, we evaluate several network proximity based redirection strategies. We believe we are the first to study how to explicitly factor intra-region proximity information into server selection algorithms.

These results demonstrate that the proposed algorithm results in a system with significantly greater capacity than published CDNs, which should improve the system's ability to handle legitimate flash crowds. The results also suggest a new strategy in defending against DDoS attacks: each server added to the system multiplicatively increases the number of resources an attacker must marshal in order to have a noticeable affect on the system.

Our algorithms and techniques developed in this research are not only limited to CDN systems, they can also be applied to other systems, such as distributed storage systems and peer-to-peer systems, to provide them with protection against abnormal traffic.

Hybrid Network-Server Simulator

To evaluate various CDN redirection strategies at a large scale under overload situations, such as flash crowds and DoS attacks, we build a hybrid network-server simulator by combining the best of existing simulators that individually models either networks or OS/server well. We merge NS-2 and LogSim simulators into a hybrid simulator, which provides detailed modeling at both packet level and operating systems level. We also optimize some modules in these two simulators, eliminating some bottlenecks that restrict us from conducting simulations at large scales. Currently we can afford to simulate a CDN system with over 1000 clients and 128 servers, and the sustainable request rate can reach to more than 70,000 requests per second.

The simulator proves to be very useful in our evaluations and its detailed accounting helps us to understand why our new redirection schemes work better. Our new dynamic strategies achieve better performance because we successfully shift the system bottleneck from disks to CPUs. Since CPU speed increases faster than that of disks, with faster CPUs, we expect the performance gap between our strategies and others will become even more pronounced. Therefore, this shift not only explains why our schemes work, but also predict that they can become even better in the future.

This hybrid simulator helps us to conduct large-scale simulations and understand a lot of performance issues in our CDN study. We believe it can also be used in other areas, such as web caching, peer-to-peer systems and distributed file storage system. We plan to use it for more experiments and also prepare to release it to the research community.

CoDeeN Deployment on PlanetLab

We are now building an academic testbed CDN, named CoDeeN, on top of PlanetLab. PlanetLab provides an overlay network with rich diversity of link behavior and widespread geographical coverage that can be used to both experiment with new network services and let users access them. These properties of PlanetLab fit CoDeeN's goals very well. We use CoDeeN both as a research testbed but also as a continuous service. We implement CoDeeN as a network of web proxies using iMimic's DataReactor proxy servers that have an efficient and flexible API allowing user specified special processing in HTTP transactions. Each proxy on CoDeenN is configured either as a request redirector, or most often, a hybrid request redirector and server surrogate. We follow an opt-in service model letting users point their browsers to one of the proxies. Currently, CoDeeN has been running on over 40 sites on PlanetLab and has been open to the public for alpha test. During the deployment process, we devise a number of monitoring schemes to watch the health of CoDeeN proxies and nodes. These monitoring routines prove to be very useful for the smooth operations of CoDeeN, and will be potentially beneficial to the monitoring of PlanetLab as a whole. We also encounter various security and management problems when operating CoDeeN as a network of open proxies. We take actions to handle these issues and believe these experiences are valuable for managing networks of open proxies in general. Meanwhile, we are preparing for using other PlanetLab nodes to stress CoDeeN and further test various redirection strategies. We also plan to conduct other research projects such as anti-spam mail honeypots.

In summary, in this thesis, we propose improved CDN request redirection strategies, evaluate them through detailed large-scale end-to-end simulations, and deploy a testbed CDN on PlanetLab overlay network. We demonstrate that the resilience of large wide area network services can be improved through efficient management of networked resources. By adapting unified resource management schemes, we present a practical way to build network systems that not only handle a larger volume of regular traffic more easily, but also absorb flash crowds and deter DoS attacks as a natural part of their operations.

6.2 Future Work

Although we believe this thesis identifies important trends, much work remains to be done.

First, we have conducted the largest detailed simulations as current simulation environment allows. We also find that approximate load information works well. We expect our new algorithms scale to very large systems with thousands of servers, but it requires a lot more resources and time to evaluate. Also, to reach a scale beyond tens of thousands of servers, we can take a hierarchical region approach, where servers within a region adapt our dynamic redirection schemes directly. In the future, we would like to run simulations at an even larger scale, with faster, more powerful simulated servers. This includes further optimizing our hybrid simulator and exploring how to integrate the hierarchical region approach. We would also like to experiment with more topologies such as those generated by power-law based topology generators, use more traces, real or synthetic (such as SPECweb99).

Second, we want to continue improving our new dynamic redirection schemes. For example, to further explore how to combine proximity information into redirection strategies, both intra-region and inter-region wise. Meanwhile, although our wide-area dynamic redirection strategies, such as FDR, are inspired by previous cluster-based schemes, their distributed, non-centralized control makes them candidates for being used in clusters and LAN environments as well. As discussed earlier, these dynamic schemes can also be adapted in other similar systems, such as distributed storage systems and peer-to-peer systems, to improve their resource utilization and service robustness. We want to explore the possible use of our redirection schemes in other contexts.

Third, we will keep working on improving our testbed CDN—CoDeeN's monitoring and managing infrastructure. We plan to stress CoDeeN from other PlanetLab nodes to calibrate our simulation results on different redirection strategies. We have also started many other interesting research projects on top CoDeeN, in particular, how to improve the reliability and security of a network of open proxies like CoDeeN.

We believe these future projects will be promising. They will not only further our knowledge on how to make network services more robust, but also possibly inspire new research directions.

Bibliography

[1] Akamai. Akamai content delivery network. http://www.akamai.com.

[2] D. G. Anderson. Mayday: Distributed filtering for internet services. In *Proceedings of the 4th USENIX Symposium on Internet Technologies an d Systems (USITS'03)*, Seattle, WA, Mar. 2003.

[3] D. Andresen, T. Yang, V. Holmedahl, and O. H. Ibarra. SWEB: Towards a scalable world wide web server on multicomputers. In *Proceedings of 10th Int. Parallel Processing Symposium (IPPS'96)*, Honolulu, HA, Apr. 1996. IEEE.

[4] M. Balazinska, H. Balakrishnan, and D. Karger. Ins/twine: A scalable peer-to-peer architecture for intentional resource discovery. In *Proceedings of International Conference on Pervasive Computing*, 2002.

[5] A. Barbir, B. Cain, F. Douglis, M. Green, M. Hofmann, R. Nair, D. Potter, and O. Spatscheck. Known CN Request-Routing Mechanisms, Feb. 2002. Work in Progress, draft-ietf-cdi-known-request-routing-00.txt.

[6] P. Barford and M. Crovella. Generating representative Web workloads for network and server performance evaluation. In *Proceedings of the SIGMETRICS '98 conference*, June 1998.

[7] S. M. Bellovin. ICMP Traceback Messages, Mar. 2000. Work in Progress, Internet Draft draft-bellovin-itrace-00.txt.

[8] T. Brisco. DNS support for load balancing. Request for Comments 1794, Rutgers University, New Brunswick, New Jersey, Apr. 1995.

[9] R. Brown. Calendar queues: A fast o(1) priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31(10):1220–1227, Oct. 1988.

[10] K. Calvert, M. Doar, A. Nexion, and E. Zegura. Modeling internet topology. *IEEE Communications Magazine*, June 1997.

[11] P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. In *Proceedings of the USENIX Symposium on Internet Technologies an d Systems (USITS)*, Monterey, CA, Dec. 1997.

[12] V. Cardellini, M. Colajanni, and P. Yu. Geographic load balancing for scalable distributed web systems. In *Proceedings of the International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Aug. 2000.

[13] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *Journal of Computer and Systems Scineces*, 18:143–154, 1979.

[14] J. Challenger, P. Dantzig, and A. Iyengar. A scalable system for consistently caching dynamic web data. In *Proceedings of IEEE INFOCOM*, New York, New York, 1999.

[15] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A hierarchical internet object cache. In *USENIX Annual Technical Conference*, pages 153–164, 1996.

[16] M. Charikar and S. Guha. Improved combinatorial algorithms for the facility location and k-median problems. In *IEEE Symposium on Foundations of Computer Science*, pages 378–388, 1999.

[17] M. Chatel. Classical versus Transparent IP Proxies. Request for Comments 1919, Mar. 1996.

[18] CNN technology news. Computer worm grounds flights, blocks atms, Jan. 2003. http://www.cnn.com/2003/TECH/internet/01/25/internet.attack/.

[19] J. Cohen, N. Phadnis, V. Valloppillil, and K. W. Ross. Cache array routing protocol v1.1. http://ds1.internic.net/internet-drafts/draft-vinod-carp-v1-01.txt, September 1997.

[20] M. Colajanni, P. S. Yu, and V. Cardellini. Dynamic load balancing in geographically distributed heterogeneous web servers. In *International Conference on Distributed Computing Systems*, pages 295–302, 1998.

[21] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1989.

[22] M. Crovella, R. Frangioso, and M. Harchol-Balter. Connection scheduling in web servers. In *USENIX Symposium on Internet Technologies and Systems*, 1999.

[23] M. Crovella, M. Harchol-Balter, and C. D. Murta. Task assignment in a distributed system: Improving performance by unbalancing load (extended abstract). In *Measurement and Modeling of Computer Systems*, pages 268–269, 1998.

[24] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of 18th ACM Symposium on Operating Systems Principles*, pages 202–215, Oct. 2001.

[25] O. Damani, P. Y. Chung, Y. Huang, C. M. R. Kintala, and Y. M. Wang. ONE-IP: Techniques for hosting a service on a cluster of machines. In *Proceedings of the Sixth International World-Wide Web Conference*, 1997.

[26] Digital Island. http://www.digitalisland.com.

[27] M. Doar. A better model for generating test networks. In *Proceedings of Globecom '96*, Nov. 1996.

[28] Edge Side Includes (ESI). http://www/esi.org.

[29] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. In *Proceedings of ACM/SIGCOMM '99*, pages 251–262, Sept. 1999.

[30] Z. Fei, S. Bhattacharjee, E. W. Zegura, and M. H. Ammar. A novel server selection technique for improving the response time of a replicated service. In *INFOCOM (2)*, pages 783–791, 1998.

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

[32] S. Gadde, J. Chase, and M. Rabinovich. Web caching and content distribution: A view from the interior. In *Proceedings of the 5th International Web Caching and Content Delivery Workshop*, Lisbon, Portugal, May 2000.

[33] L. Garber. Technology news: Denial-of-service attacks rip the Internet. *Computer*, 33(4):12–17, Apr. 2000.

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

[35] J. D. Guyton and M. F. Schwartz. Locating nearby copies of replicated internet servers. In *SIGCOMM*, pages 288–298, 1995.

[36] A. Harrison. Cyberassaults hit buy.com, ebay, cnn and amazon. *Computerworld*, Feb. 2000.

[37] G. Hunt, E. Nahum, and J. Tracey. Enabling content-based load distribution for scalable services. Technical report, IBM T.J. Watson Research Center, May 1997.

[38] iMimic Networking. DataReactor. http://www.imimic.com.

[39] R. Jain. *The Art of Computer Systems Performance Analysis*. John Wiley and Sons, Inc., 1996.

[40] C. Jin, Q. Chen, and S. Jamin. Inet: Internet topology generator, 2000.

[41] K. L. Johnson, J. F. Carr, M. S. Day, and M. F. Kaashoek. The measured performance of content distribution networks. In *Proceedings of The 5th International Web Caching and Content Delivery Workshop*, Lisbon, Portugal, May 2000.

[42] M. Kanellos. Intel, universities create world network. *The New York Times*, June 2003. http://www.nytimes.com/cnet/CNET_2100-1035_3-1020157.html.

[43] 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 Eighth International World-Wide Web Conference*, 1999.

[44] 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*, pages 654–663, 1997.

[45] A. D. Keromytis, V. Misra, and D. Rubenstein. SOS: Secure overlay services. In *Proceedings of the ACM SIGCOMM 2002*, pages 61–72, Aug. 19–23 2002.

[46] B. Li, M. Golin, G. Italiano, X. Deng, and K. Sohraby. On the optimal placement of Web proxies in the Internet. In *Proceedings of the INFOCOM '99 conference*, Mar. 1999.

[47] Q. Luo and J. F. Naughton. Form-based proxy caching for database-backed web sites. In *The VLDB Journal*, pages 191–200, 2001.

[48] E. P. Markatos. On caching search engine query results. In *Proceedings of The 5th International Web Caching and Content Delivery Workshop*, Lisbon, Portugal, May 2000.

[49] A. Medina, I. Matta, and J. Byers. BRITE: An Approach to Universal Topology Generation. In *Proceedings of MASCOTS 2001*, Cincinnati, OH, Aug. 2001.

[50] Mirror Image. http://www.mirror-image.com.

[51] P. Mockapetris. Domain names—implementation and specification. Request for Comments 1035, Nov. 1987.

[52] D. Moore, G. Voelker, and S. Savage. Inferring internet denial of service activity. In *Proceedings of 2001 USENIX Security Symposium*, Aug. 2001.

[53] Network Systems Group, Princeton University. CoDeeN—A CDN on PlanetLab. http://codeen.cs.princeton.edu.

[54] L. Niven. *The Flight of the Horse*. Ballantine Books, 1971.

[55] NS. (Network Simulator). http://www.isi.edu/nsnam/ns/.

[56] J. Ousterhout. Scripting: Higher-level programming for the 21st century. *IEEE Computer*, 31(3):23–30, Mar. 1998.

[57] V. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable web server. In *USENIX Annual Technical Conference*, June 1999.

[58] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. M. Nahum. Locality-aware request distribution in cluster-based network servers. In *Architectural Support for Programming Languages and Operating Systems*, pages 205–216, 1998.

[59] V. S. Pai, A. L. Cox, V. S. Pai, and W. Zwaenepoel. A Flexible and Efficient Application Programming Interface (API) for a Customizable Proxy Cache. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS 03)*, Seattle, WA, Mar. 2003.

[60] V. Paxson. An analysis of using reflectors for distributed denial-of-service attacks. *Computer Communications Review*, 31(3), July 2001.

[61] 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.

[62] L. L. Peterson and B. S. Davie. *Computer Networks A Systems Approach, Third Edition*. Morgan Kaufmann, May 2003.

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

[64] L. Qiu, V. Padmanabhan, and G. Voelker. On the placement of web server replicas. In *Proceedings of the Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM-01)*, pages 1587–1596, Los Alamitos, CA, Apr. 22–26 2001. IEEE Computer Society.

[65] M. Rabinovich and O. Spatscheck. *Web Caching and Replication*. Addison-Wesley Press Inc., Jan. 2002.

[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] 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.

[68] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of 18th ACM Symposium on Operating Systems Principles*, pages 188–201, Oct. 2001.

[69] A. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. Scribe: The design of a large-scale event notification infrastructure. In *Networked Group Communication (NGC'2001)*, Nov. 2001.

[70] S. Savage, D. Wetherall, A. Karlin, and T. Anderson. Practical network support for IP traceback. In *Proceedings of the 2000 ACM SIGCOMM Conference*, Aug. 2000.

[71] B. Smith, A. Acharya, T. Yang, and H. Zhu. Exploiting result equivalence in caching dynamic web content. In *USENIX Symposium on Internet Technologies and Systems*, 1999.

[72] A. C. Snoeren, C. Partridge, L. A. Sanchez, C. E. Jones, F. Tchakountio, S. T. Kent, and W. T. Strayer. Hash-based ip traceback. In *Proceedings of ACM SIGCOMM'01*, Aug. 2001.

[73] N. Spring, R. Mahajan, and D. Wetherall. Measuring isp topologies with rocketfuel. In *Proceedings of ACM/SIGCOMM '02*, Aug. 2002.

[74] Squid. The Squid Web Proxy Cache. http://www.squid-cache.org.

[75] W. R. Stevens. *UNIX Network Programming, 2nd Edition*. Prentice Hall, 1998.

[76] 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.

[77] H. Tangmunarunkit, R. Govindan, S. Shenker, S. Jamin, and W. Willinger. Network topology generators: Degree-based vs. structural. In J. Wroclawski, editor, *Proceedings of the ACM SIGCOMM 2002*, pages 147–160, Aug. 19–23 2002.

[78] D. G. Thaler and C. V. Ravishankar. Using name-based mappings to increase hit rates. *IEEE/ACM Transactions on Networking*, 6(1):1–14, Feb. 1998.

[79] USC. Transmission control protocol. Request for Comments 793, USC Information Sciences Institute, Marina del Ray, Calif., Sept. 1981.

[80] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, J. Chase, and D. Becker. Scalability and Accuracy in a Large-Scale Network Emulator. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, Boston, MA, December 2002.

[81] L. Wang, V. Pai, and L. Peterson. The Effectiveness of Request Redirecion on CDN Robustness. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, Boston, MA, December 2002.

[82] B. M. Waxman. Routing of multipoint connections. *IEEE Jour. Selected Areas in Communications (Special Issue: Broadband Packet Communications)*, 6(9):1617–1622, Dec. 1988.

[83] D. Wetherall and C. J. Lindblad. Extending Tcl for dynamic object-oriented programming. In *Proceedings of the Tcl/Tk Workshop*, Ontario, Canada, July 1995.

[84] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Envionment for Distributed Systems and Networks. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, Boston, MA USA, December 2002.

[85] 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*, pages 16–31, 1999.

[86] L. Zhang, S. Floyd, and V. Jacobson. Adaptive web caching. In *Proceedings of the 1997 NLANR Web Cache Workshop*, June 1997.

[87] G. K. Zipf. *Human Behavior and the Principle of Least Effort: An Introduction to Human Ecology*. Addison-Wesley Press Inc., Cambridge 42, MA, USA, 1949.