Towards Autonomic Computing: Service Discovery and Web Hotspot Rescue

Weibin Zhao

Submitted in partial fulfillment of the

requirements for the degree

of Doctor of Philosophy

in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2006

ABSTRACT

Towards Autonomic Computing: Service Discovery and Web Hotspot Rescue

Weibin Zhao

Autonomic computing is a vision that addresses the growing complexity of computing systems by enabling them to manage themselves without direct human intervention. This thesis studies two related problems, service discovery and web hotspot rescue, which can serve as a building block and a prototype for autonomic networking and distributed systems, respectively.

Service discovery allows end systems to discover desired services on networks automatically, eliminating administrative configuration. We made four enhancements to the Service Location Protocol (SLP): mesh enhancement, remote service discovery, preference filters, and global attributes. These enhancements improve SLP efficiency and scalability, and enable SLP to better support new and advanced discovery scenarios. The SLP mesh enhancement (mSLP), remote service discovery, and preference filters are now experimental RFCs (Request for Comments). We expect that similar techniques can be applied to other service discovery systems.

During the development of mSLP, we designed selective anti-entropy, a generic mechanism for high availability partial replication. Traditional anti-entropy only supports full replication. We enhanced it to support partial replication by allowing two replicas to selectively reconcile inconsistent data in a session.

Web hotspots are short-term dramatic load spikes. We developed DotSlash, a self-configuring and scalable rescue system for handling web hotspots effectively. DotSlash

works autonomously. It uses service discovery to allocate resources dynamically from a server pool distributed globally, and uses adaptive overload control to automate the whole rescue process. As a comprehensive solution, DotSlash enables a web site to build an adaptive distributed web server system on the fly, replicate application programs dynamically, and set up distributed query result caching on demand. DotSlash relieves a spectrum of bottlenecks ranging from access network bandwidth to web servers, application servers, and database servers.

As part of DotSlash, we developed a prediction algorithm for estimating the upper bound of future web traffic volume, which is simple and effective for short-term bursty web traffic. This algorithm provides insight into characterizing traffic of web hotspots, and is useful for web server overload prevention.

Contents

1 Introduction **1**

1.1 Problem Statement and Our Approach 2

1.1.1 Service Discovery 2

1.1.2 Web Hotspot Rescue 4

1.2 Thesis Contributions 5

1.3 Thesis Outline 7

2 Enhancements to the Service Location Protocol **8**

2.1 Introduction 8

2.2 Background 11

2.2.1 Service Discovery 11

2.2.2 Systems Related to Service Discovery 13

2.2.3 Service Location Protocol 14

2.3 Mesh Enhancement 17

2.3.1 Motivation 17

2.3.2 Design Overview 18

2.3.3 Peer Relationship Management 20

2.3.3.1 Learning about New Peers 20

2.3.3.2 Establishing a Peering Connection 20

2.3.3.3 Exchanging Peer Information 21
2.3.3.4 Maintaining a Peer Relationship 21
2.3.3.5 Tearing Down a Peer Relationship 22
2.3.4 Registration Propagation Control 22
2.4 Remote Service Discovery 25
2.4.1 Motivation 25
2.4.2 Design 26
2.5 Preference Filters 27
2.5.1 Motivation 27
2.5.2 Design 28
2.6 Global Attributes 30
2.6.1 Motivation 30
2.6.2 Basic Design 31
2.6.3 Advanced Usages 32
2.7 Implementation 33
2.8 Evaluation 35
2.8.1 Mesh Enhancement 35
2.8.2 Preference Filters 39
2.8.3 Global Attributes 41
2.9 Related Work 42
2.10 Summary 44

3 Selective Anti-Entropy **45**

3.1 Replication 45
3.1.1 Anti-Entropy 48

3.2 Motivation 49
3.3 Design 51
3.3.1 Safe Sessions 52
3.3.2 Parallel Sessions 56
3.4 Implementation 57
3.5 Evaluation 57
3.5.1 Performance Analysis 58
3.5.2 Experimental Results 58
3.6 Related Work 59
3.7 Summary 60

4 DotSlash: An Automated Web Hotspot Rescue System 61

4.1 Introduction 61
4.2 Related Work 64
4.3 DotSlash Overview 67
4.3.1 Usage Models 68
4.3.2 Rescue Examples 70
4.4 DotSlash Design 73
4.4.1 Dynamic Virtual Hosting 73
4.4.2 Request Redirection 75
4.4.3 Workload Monitoring 78
4.4.4 Rescue Control 80
4.4.4.1 Rescue Protocol 80
4.4.4.2 Rescue Control Overview 82
4.4.4.3 Rescue Actions and State Transitions 84

4.4.5 Service Discovery 89
4.5 Implementation 90
4.6 Evaluation 92
4.6.1 Workload Generation 93
4.6.2 Experimental Setup 93
4.6.3 Experimental Results on PlanetLab 95
4.6.4 Experimental Results in Local Area Networks 97
4.7 Summary 100

5 Hotspot Rescue for Dynamic Content by Replicating Application Programs Dynamically 101

5.1 Introduction 102
5.2 Related Work 104
5.3 Dynamic Script Replication 104
5.3.1 Operations at the Rescue Server 106
5.3.2 Operations at the Origin Server 107
5.3.3 File Inclusions in Replicated Scripts 107
5.3.4 Implementation 108
5.4 Evaluation 109
5.4.1 Experimental Setup 109
5.4.2 Effectiveness 110
5.4.3 Workload Control and Migration 112
5.5 Summary 115

6 Hotspot Rescue for Dynamic Content by Using On-demand Dis-

tributed Query Result Caching **116**

6.1 Introduction 117

6.2 System Design 118

6.2.1 Design Goals 118

6.2.2 Scalability Mechanisms 118

6.2.3 Application Model 119

6.2.4 System Architecture 119

6.2.5 Caching Features 122

6.2.6 Data Driver 124

6.2.7 Query Result Cache 126

6.3 Evaluation 128

6.3.1 Benchmark Description 128

6.3.2 Experimental Setup 129

6.3.3 Caching TTL 131

6.3.4 Results for the RUBBoS Read-only Mix 132

6.3.5 Results for the RUBBoS Submission Mix 138

6.4 Related Work 144

6.5 Summary 145

7 Web Traffic Prediction for Overload Prevention **147**

7.1 Introduction 147

7.2 Related Work 148

7.3 Motivation 149

7.4 Prediction Algorithm 150

7.5 Parameter Selection 151

7.6 Experimental Results 152
7.7 Summary 155

8 Conclusions **157**

8.1 Thesis Summary 157
8.2 Thesis Contributions 160
8.3 Future Work 161

Bibliography **164**

List of Figures

2.1 SLP system architecture: User Agents (UAs) initiate service discovery on behalf of service users by querying all SAs via multicast or a DA (if available) via unicast; Service Agents (SAs) work on behalf of service providers by responding directly to UA queries, and registering with DAs (if exist); and Directory Agents (DAs) serve as centralized information repositories by accepting SA registrations and answering UA queries. 15

2.2 An example of the mSLP scope-based fully-meshed peering DA architecture for four DAs (DA_1 to DA_4) and three scopes (S_1 to S_3), where an edge between two DAs means that they are peers. 18

2.3 Steps for establishing a peering connection between DA_1 and DA_2: (1) DA_1 gets DA_2's `DAAdvert`, (2) DA_1 initiates a peering connection to DA_2, and (3) DA_1 sends its `DAAdvert` along this peering connection to DA_2. 21

2.4 The processing of a `SrvRqst` that has a search filter and a preference filter 28

2.5 Without using mSLP: consistency between two peer DAs at host *ankara* and *ottawa* after host *ottawa* recovered from two failures during 90 to 440 seconds and 1150 to 1320 seconds 36

2.6 The relationship between μ and ρ for $\rho \in [0, 1]$, where μ is the average rate of missing entries during the first R seconds after a DA recovered from failures, and ρ is the ratio of failure duration over recovery interval R. 38

2.7 Using mSLP: consistency between two peer DAs at host *ankara* and *ottawa* after host *ottawa* recovered from two failures during 90 to 440 seconds and 1150 to 1320 seconds 39

2.8 Using preference filters versus without using preference filters: the response time measured from two PlanetLab nodes, *gtidsl1* and *ucla1*, where node *gtidsl1* was behind a DSL line, and node *ucla1* connected to Internet2. 40

2.9 Using global attributes versus without using global attributes: the time used for performing location-based queries from a PlanetLab DSL node *gtidsl1* 42

3.1 An example of $\mathcal{P}(6, 3)$, a partial replication system with 6 replicas and 3 scopes, where replicas are R_1 to R_6, scopes are S_1 to S_3, and an edge between two replicas means that they share scopes. 46

3.2 An example to show how selective anti-entropy (here we use select-one-direct anti-entropy) differs from complete anti-entropy 53

3.3 The performance of parallel select-one-direct sessions compared to that of sequential complete sessions 59

4.1 An example of DotSlash rescue relationships for eight web servers S_1 to S_8, where an arrow from S_y to S_x denotes that S_y provides rescue services to S_x. In this figure, S_1 and S_2 are origin servers; S_3, S_4, S_5, and S_6 are rescue servers; and S_7 and S_8 are not involved with rescue services yet. 68
4.2 Two examples for DotSlash rescue services 71
4.3 DotSlash rescue protocol (DSRP) 81
4.4 DotSlash closed-loop rescue control system 83
4.5 DotSlash rescue actions and state transitions 84
4.6 Algorithm for adjusting P_r at an origin server 85
4.7 Adjusting P_r at an origin server for two different workloads by using the algorithm shown in Figure 4.6. In this figure, ρ_n is the filtered value of the raw measurement of network utilization which changes at control interval 21: from 0.8 to 0.2 for workload1, and from 0.9 to 0.1 for workload2. 86
4.8 Algorithm for adjusting λ_{rd}^a at a rescue server 87
4.9 Adjusting λ_{rd}^a at a rescue server by using the algorithm shown in Figure 4.8. In this figure, ρ_n is the filtered value of the raw measurement of network utilization which changes from $\lambda_{rd}^a/\lambda_d^m$ to 0.9 at control interval 21. 88
4.10 Rescue server discovery via a set of fully-meshed DotSlash registries (based on mSLP DAs), where R/S denotes registration and search operations, and P denotes a peer relationship. 89
4.11 DotSlash software architecture 90

4.12 The data rate and request rate for a PlanetLab DSL node *gtidsl1* in two cases, namely without using DotSlash verses using DotSlash. Note that figure (a) and (b) have different scales of ordinates, and 1 kB = 1000 bytes. 96
4.13 The request rates and data rates at the origin server *bjs* and its rescue servers 98

5.1 The three-tier architecture for dynamic content web sites 102
5.2 The LAMP configuration for dynamic content web sites 103
5.3 An example for dynamic script replication 105
5.4 The request rate and timeout rate for the origin web server *Orig_HC* in two cases, namely without using DotSlash verses using DotSlash. . 111
5.5 The request rate and redirect rate at the origin server *Orig_LC* and the rescue rate at the 9 rescue servers (*Resc_LC1*, ..., *Resc_LC9*) 113
5.6 The CPU utilization for the origin server *Orig_LC*, the 9 rescue servers (*Resc_LC1*, ..., *Resc_LC9*), and the database server *DB_HC*. 114

6.1 DotSlash Application Model 119
6.2 Enabling query result caching in DotSlash 120
6.3 DotSlash on-demand query result caching, where caching is activated (cache on) or de-activated (cache off) based on the web server's DotSlash state (normal, SOS, or rescue) and load region (desired load, heavy load, or light load). 123
6.4 The relationship between the caching TTL and query result cache hit ratio in a set of 10-minute experiments for the RUBBoS read-only mix 131

6.5 Experimental results for the RUBBoS read-only mix when rescue servers are not available 134
6.6 Experimental results for the RUBBoS read-only mix when rescue servers are available 136
6.6 Experimental results for the RUBBoS read-only mix when rescue servers are available (Continued) 137
6.7 Experimental results for the RUBBoS submission mix when rescue servers are not available 140
6.8 Experimental results for the RUBBoS submission mix when rescue servers are available 142
6.8 Experimental results for the RUBBoS submission mix when rescue servers are available (Continued) 143

7.1 Prediction results for day74 of the 1998 World Cup data set, where $n = 10$, $H = 0.85$, and the percentage of correct predictions is computed as the percentage of prediction intervals in which the real traffic volume falls below the predicted upper bound. 153
7.2 Detailed prediction results for server41 on day65 of the 1998 World Cup data set, where $n = 10$, $H = 0.85$, and $T = 100$ seconds. 154
7.3 Prediction results for server41 on day65 of the 1998 World Cup data set, where $n = 10, 20, 50, 100$, $H = 0.5, 0.65, 0.8, 0.85, 0.9$, and $T = 100$ seconds. 155

List of Tables

3.1 The scopes and accept-ids for three updates, u_6^1, u_6^2, and u_6^3, at replica R_6 that are accepted by R_6 from clients, and $t_6^1 < t_6^2 < t_6^3$. 50

3.2 Four types of selective anti-entropy sessions: select-one-direct, select-one-indirect, select-multiple, and select-all, where the total number of replicas is r. 52

4.1 Major DotSlash parameters, where type C, O, I, and D denote configurable parameters, measured outputs, control inputs, and derived parameters, respectively, and 1 kB = 1000 bytes. 78

6.1 Three configurations in using DotSlash 121

6.2 DotSlash caching-enhanced data driver, which handles database queries based on the web server's query result caching state (on or off), the client request HTTP Cache-Control header (bypass caching or not), and the client request type (rescue or regular). 126

6.3 Summary of experimental results for the RUBBoS read-only mix . . . 133

6.4 Summary of experimental results for the RUBBoS submission mix . . 139

Acknowledgments

This thesis would not have been possible without the help of many people. I am deeply grateful to my advisor Prof. Henning Schulzrinne, who is very patient in helping me to develop my thesis topic and provides great guidance throughout my thesis research. Many thinks to Prof. Jason Nieh, Prof. Vishal Misra, Prof. Dan Rubenstein, and Dr. Chatschik Bisdikian for serving on my dissertation committee and providing helpful suggestions for my thesis.

All members of our research group, Xiaotao Wu, Wenyu Jiang, Xin Wang, Jonathan Lennox, Maria Papadopouli, Kundan Singh, Charles Shen, Sangho Shin, Knarig Arabshian, Stelios Sidiroglou-Douskos, Ping Pan, Ashutosh Dutta, Jonathan Rosenberg, Lisa Amini, and Santosh Krishnan, are great sources for ideas and fun. I thank them for making my Ph.D. study productive and enjoyable.

I thank Dr. Christoph Schuba for being my mentor and making my summer internship at Sun Microsystems Research Labs a wonderful experience. Many thanks to Dr. Chatschik Bisdikian and William Jerome for offering me two summer internships at IBM T. J. Watson Research Center and providing a great help with my thesis research. Thanks to Dr. Michah Lerner for helping me in my summer internship at AT&T Research Labs.

Last, but not least, I am grateful to my family for their love, encouragement, and constant support. This thesis is dedicated to the memory of my father and to my mother.

To the Memory of My Father

and

To My Mother

Chapter 1

Introduction

Autonomic computing [69, 79] is a vision that addresses the growing complexity of computing systems by enabling them to manage themselves without direct human intervention. For networking and distributed systems, manual control and configuration are not only time-consuming, expensive, and error-prone, but also difficult in certain situations. For example, mobile devices need to adapt to new environments dynamically and web servers need to handle unpredictable dramatic load spikes. As a result, there is an increasing need to build self-managing systems that self-monitor, self-configure, self-optimize, self-heal, and self-protect.

While realizing the full potentials of autonomic computing is still a grand challenge, we start with two important components, service discovery and web hotspot rescue, which can serve as a building block and a prototype for autonomic networking and distributed systems, respectively.

1.1 Problem Statement and Our Approach

This thesis studies two related problems, service discovery and web hotspot rescue. Service discovery allows end systems to discover desired services on networks automatically, eliminating configuration by system administrators. Web hotspot rescue enables web sites to scale dynamically as needed, handling short-term dramatic load spikes autonomously without human intervention.

1.1.1 Service Discovery

Service discovery is a technology that can dynamically map service descriptions into service access points. It provides a powerful and flexible way for service users to locate available desired services on networks. Services are of various types, such as printing services, computational services, and storage services. Service discovery differs from other resource discovery such as information retrieval by focusing on where desired services are provided. Traditionally, service users depend on a priori knowledge or manual configuration to learn about desired services on networks, involving non-trivial administrative overhead as more devices are network enabled and more services are available on networks. Moreover, administrative configuration becomes difficult or even impossible in certain situations such as ad-hoc networks. By using service discovery technology, service users no longer need to know the access points of desired services via a priori knowledge. Instead, they just need to specify the characteristics of desired services, which will be mapped into available service access points automatically in any network that supports service discovery. In summary, service discovery is a promising technology for building autonomic networking and distributed systems.

Challenges. A major challenge in service discovery is scalability. Currently,

service discovery is often performed at small scales via multicast [130] or within specific domains such as administrative domains [59] or application domains [129] via service registries. Generic service discovery is still an open issue. Another challenge is to support new discovery scenarios effectively. One discovery scenario not well supported so far is to discover the best matching service for a request, such as finding a printer that has the shortest queue or a server that has the minimum load. Another discovery scenario not well addressed yet is to discover services that are provisioned in complex ways, such as services that have multiple access points or services that are replicated at multiple locations.

Our approach. To leverage existing efforts in service discovery and make our proposed techniques more likely to be used in real applications, we choose to enhance an existing service discovery system instead of designing a new system from scratch. We select the Service Location Protocol (SLP) [59] as our base system since SLP is an IETF (Internet Engineering Task Force) proposed standard for service discovery in IP networks, and it is flexible, lightweight, and powerful. We made four enhancements to SLP: mesh enhancement (mSLP) [156], remote service discovery [158], preference filters [157], and global attributes [152]. These enhancements improve SLP scalability and efficiency, and enable SLP to better support new and advanced discovery scenarios. Although our techniques for service discovery are developed in the context of SLP, they can be generally applied to other service discovery systems.

During the development of mSLP, we designed selective anti-entropy [147], a generic mechanism for high availability partial replication. Traditional anti-entropy [96, 54] only supports full replication. We enhanced it to support partial replication by allowing two replicas to selectively reconcile inconsistent data in a session. As a generalization of traditional anti-entropy, selective anti-entropy is flexible and applicable to both full replication and partial replication.

1.1.2 Web Hotspot Rescue

Web hotspots, also known as flash crowds or the Slashdot effect [4], are short-term dramatic load spikes that can seriously degrade the service quality of affected web sites. When a web site experiences a hotspot, its request rate increases dramatically. But the peak load often lasts for a short period, and is usually a one-time event like "15 minutes of fame". To effectively handle web hotspots, researchers are actively seeking solutions such as extending server capacity, adding more network bandwidth, deploying various caches [135, 126], employing commercial Content Delivery Networks (CDNs) [5, 45], reducing content complexity under heavy load [1], performing adaptive admission controls [140], and replicating contents and redirecting client requests [77].

Challenges. For web hotspots, over-provisioning is not only inefficient but also difficult since the peak load is hard to predict [74]. This calls for an automated system that can dynamically extend a web site's capacity as needed to handle dramatic load surges. There are three major challenges here. The first one is how to discover and allocate needed resources dynamically because static configuration is insufficient. The second challenge is how to automate the hotspot handling process so as to react quickly to load spikes and improve a web site's availability during critical periods. The third challenge is how to address different bottlenecks in the web server infrastructure. For static content, the access network bandwidth tends to be the most common bottleneck [92]; but for dynamic content, different applications may have different bottlenecks [7, 33], including web servers, application servers, and database servers. Consequently, we need to migrate workloads across wide area networks and we need a comprehensive solution to relieve different bottlenecks.

Our approach. Based on the framework outlined in [38], we designed *DotSlash* [150, 151, 154, 144], a self-configuring and scalable rescue system for handling web

hotspots effectively. DotSlash supports dynamic collaboration among different web servers, using spare capacity in a mutual-aid community to relieve web hotspots experienced by any individual site. DotSlash provides an automated rescue process, including rescue server discovery, workload monitoring, request redirection, dynamic virtual hosting, and rescue relationship management. DotSlash enables a web site to build a distributed web server system across wide area networks on the fly, effectively removing the bottlenecks at access network bandwidth and web servers. To handle hotspots at dynamic content web sites, DotSlash supports dynamic replication of application programs, eliminating the application server bottleneck. To relieve the database server bottleneck, DotSlash allows a web site to set up on-demand distributed query result caching.

As part of DotSlash, we developed a prediction algorithm for estimating the upper bound of future web traffic volume [148]. We employ a multiple-time-scale approach by using traffic information at a smaller time scale to forecast traffic volume at a larger time scale. Moreover, we utilize traffic statistical properties other than curve fitting to forecast traffic volume.

1.2 Thesis Contributions

This thesis research makes the following contributions.

- **Enhancements to the Service Location Protocol.** We made four enhancements to the Service Location Protocol (SLP): mesh enhancement that simplifies Service Agent registrations and improves the consistency of peer Directory Agents, remote service discovery that enables SLP users to discover services at remote DNS domains, preference filters that facilitate processing of search results at SLP servers, and global attributes that allow using a single

query to search services across multiple types. These enhancements [145, 153] improve SLP efficiency and scalability, and enable SLP to better support new and advanced discovery scenarios. The SLP mesh enhancement (mSLP), remote service discovery, and preference filters are now experimental RFCs (Request for Comments) [157, 156, 158]. We expect that similar techniques can be applied to other service discovery systems as well.

- **Selective anti-entropy.** We developed selective anti-entropy [147], a generic mechanism for high availability partial replication. Traditional anti-entropy [96, 54] only supports full replication. We enhanced it to support partial replication by allowing two replicas to selectively reconcile inconsistent data in a session. As a generalization of traditional anti-entropy, selective anti-entropy is flexible and applicable to both full replication and partial replication.

- **DotSlash—an automated web hotspot rescue system.** We developed DotSlash [150, 151, 154, 144], a self-configuring and scalable rescue system for handling web hotspots effectively. DotSlash works autonomously. It uses service discovery to allocate resources dynamically from a server pool distributed globally, and uses adaptive overload control to automate the whole rescue process. DotSlash is a cost-effective mechanism for small to medium-sized web sites to handle short-term dramatic load spikes. As a comprehensive solution, DotSlash enables a web site to build an adaptive distributed web server system on the fly, replicate application programs dynamically, and set up distributed query result caching on demand. These techniques relieve a spectrum of bottlenecks ranging from access network bandwidth to web servers, application servers, and database servers.

- **Web traffic prediction for overload prevention.** We developed a pre-

diction algorithm for estimating the upper bound of future web traffic volume [148], which is simple and effective for short-term bursty web traffic. We employ a multiple-time-scale approach, and utilize traffic statistical properties to forecast traffic volume. Our prediction algorithm provides insight into characterizing traffic of web hotspots, and is useful for web server overload prevention.

1.3 Thesis Outline

The rest of this thesis is organized as follows. We describe the four enhancements we made to the Service Location Protocol in Chapter 2, and discuss selective anti-entropy for high availability partial replication in Chapter 3. Then, we present DotSlash—an automated rescue system for handing web hotspots effectively: we focus on hotspot rescue for static content in Chapter 4, and address hotspot rescue for dynamic content in Chapter 5 and 6. After a discussion of our web traffic prediction algorithm for overload prevention in Chapter 7, we summarize the thesis in Chapter 8.

Chapter 2

Enhancements to the Service Location Protocol

In this chapter, we first introduce service discovery technology and the motivation for enhancing the Service Location Protocol (SLP) [59]. We then give a brief overview of service discovery, related systems, and SLP. The main body of this chapter describes the four enhancements we made to SLP: mesh enhancement [156], remote service discovery [158], preference filters [157], and global attributes [152]. These enhancements improve SLP scalability and efficiency, and enable SLP to better support new and advanced discovery scenarios [153]. After presenting our implementation and evaluation for these enhancements, we discuss related work and give a summary.

2.1 Introduction

As computing continues moving towards a network-centric model, automatically discovering available services on networks becomes increasingly important. Services are of various types, such as printing services, computational services, and storage ser-

vices. Service discovery differs from other resource discovery such as information retrieval by focusing on where desired services are provided, that is, discovering service access points. In IP networks, a *service access point* can be encoded as a URL (Uniform Resource Locator) or specified by a tuple consisting of IP address, port number, and transport protocol. Traditionally, service users depend on a priori knowledge or manual configuration to learn about the access points of desired services, involving non-trivial administrative overhead as more devices such as personal digital assistants, cellular phones, and digital cameras are network enabled and more services are available on networks. Moreover, administrative configuration becomes difficult or even impossible in certain situations such as ad-hoc networks. Consider the following application scenarios. In pervasive computing, mobile devices are peripheral-poor due to portability considerations and power consumption constraints. Thus, they often rely on services provided by other devices. When a mobile device moves to a new network, it needs to discover and make use of available services in the new environment. In ad-hoc networks such as a disaster rescue setting, devices need to learn about each other dynamically and cooperate, where administrative configuration is unlikely to be possible and effective. For home networking, low cost and ease of use are dominant design considerations, making administrative configuration unsuitable.

In recognizing the need to reduce administrative configuration as much as possible and enable automated discovery of desired services, many companies, standards bodies, consortia, and research institutions are actively developing service discovery technology. As a result, various service discovery systems, protocols, and research prototypes are emerging in recent years, such as the Service Location Protocol (SLP) [59], Jini [134], Universal Plug and Play (UPnP) [130], Universal Description Discovery and Integration (UDDI) [129], Bonjour [25], the Bluetooth Service Discovery Protocol (SDP) [24], the Berkeley Service Discovery Service (SDS) [41], the Inten-

tional Naming System (INS) [3], and VIA [31].

We can categorize the existing service discovery systems along a number of dimensions. In terms of running platforms, Jini runs on Java platforms whereas SDP is for Bluetooth access technology. With respect to application domains, UPnP targets home networking applications whereas UDDI targets web services applications. Regarding design goals, SLP is designed for discovering different types of local services within one administrative domain, whereas UDDI is designed for discovering a single type of web services across different administrative domains. As to discovery mechanisms, UPnP uses multicast, UDDI uses registries, and SLP uses both mechanisms.

Despite their differences, all service discovery systems support the same basic service discovery functionality, namely dynamically mapping service descriptions into service access points. Service discovery provides a powerful and flexible way for locating services on networks. By using this technology, service users no longer need to know the access points of desired services via a priori knowledge. Instead, they just need to specify the characteristics of desired services, which will be mapped into available service access points automatically in any network that supports service discovery. In summary, service discovery is a promising technology for building autonomic networking and distributed systems.

SLP is a widely used service discovery protocol, and an IETF (Internet Engineering Task Force) proposed standard for service discovery in IP networks. As more applications [68, 87, 14, 101] employ SLP for various discovery purposes, we saw a need to improve SLP efficiency and scalability, and enhance it to support new discovery scenarios such as discovering multi-access-point services and multi-function devices. In this chapter, we present four new mechanisms for SLP: mesh enhancement, remote service discovery, preference filters, and global attributes. The SLP mesh enhancement (mSLP) simplifies Service Agent (SA) registrations and improves

consistency among Directory Agents (DAs) by defining an interaction scheme for DAs and supporting automatic registration distribution among peer DAs. Remote service discovery enables SLP users to discover services at remote DNS domains. Preference filters facilitate processing of search results such as finding the best match at SLP servers, either DAs or SAs, to reduce the amount of data transferred to service users for saving network bandwidth. Global attributes allow using a single query to search services across multiple types.

2.2 Background

2.2.1 Service Discovery

In a service discovery system, a common service description framework is needed for service providers, referred to as servers, and service users, referred to as clients, to describe service characteristics so that they can understand each other properly. In general, a service can be described using a set of attribute-value pairs, with each attribute-value pair specifying one property of the service. There are two ways to organize attributes: a flat structure where all attributes are at the same level, and a hierarchical structure where attributes can be at different levels. For example, SLP [59] simply puts attribute-value pairs into a list, whereas UPnP [130] and UDDI [129] use XML to describe a hierarchy of attributes. Although the Resource Description Framework (RDF) [106] has been proposed as the service description format for interoperability among different service discovery systems [107] and for global service discovery [11], so far there is no service description standard.

While service advertisements from servers usually include all attributes of services, service search requests from clients only include attributes of interest, which may just

specify a desired service type, such as "printer", or give additional desired service properties, such as "speed = 15 ppm". In general, any service search request can be specified by a search filter such as those used in SLP and the Lightweight Directory Access Protocol (LDAP) [61], which is a logical expression about attributes of interest and their desired values. The matching of a service search request with a service advertisement leads to a discovery.

Multicast and service registries (also known as directory services) are two widely used mechanisms for service discovery. In multicast-based discovery, servers and clients all listen to a well-known multicast address, and services are discovered in a peer-to-peer fashion in two ways. In passive discovery, servers periodically multicast their service advertisements, and clients listen to these advertisements. A client compares received service advertisements with its desired service requirements to determine matching services. In active discovery, clients multicast their service search requests, and servers listen to these requests. A server compares received service search requests with its service advertisement, and if there is a match, the server unicasts its service advertisement to the client. Multicast-based service discovery is simple, and multicast can enable a device to be fully auto-configured [35] in a network segment. However, multicast usually cannot scale to a large number of devices, and it is not generally supported in wide area networks. One reason is that when a client multicasts a request, it may face a response implosion problem if a large number of servers all answer its request within a very short period of time. In order to scale well, registry-centric discovery is needed, where registries accept service advertisements from servers, and answer service search requests from clients. Servers register their services with registries and clients search services at registries, all using unicast. To discover service registries, multicast can be used for an intranet [59, 134], but well-known registries are often assumed for the Internet [129]. In addition, the

Dynamic Host Configuration Protocol (DHCP) [46] can be used to obtain registry information in local area networks [95], and DNS SRV [56] can be used to obtain registry information for given DNS domains [158].

2.2.2 Systems Related to Service Discovery

Web search engines. Service discovery systems are similar to web search engines [55] in that they both provide matches for search requests. However, they are designed for different purposes and work differently. First, service discovery is used for finding services on networks whereas web search is used for finding documents on the Internet. Secondly, service discovery uses attribute-based matching whereas web search often performs full-text matching. Consequently, a match in service discovery has a specific meaning, but a match in web search may have different meanings in different contexts, and a ranking algorithm is needed to rank web search results. Finally, service registries rely on service providers to register information whereas web search engines use crawlers to collect information automatically.

LDAP. The Lightweight Directory Access Protocol (LDAP) [133] is a widely used general-purpose directory service, which organizes directory entries into a hierarchical structure to scale well. Although LDAP directories can serve as back-end service registries [67], they need a native discovery system to serve as the front-end because LDAP does not provide a mechanism such as multicast discovery for clients to discover directories. Moreover, LDAP directories are often updated by administrators whereas service registries are intended for service providers to publish information themselves.

DNS. The Domain Name System (DNS) [82, 83] is a crucial infrastructure of the Internet. It can be extended by supporting new Resource Records (RRs). DNS SRV [56] has been defined to facilitate locating services. Multicast DNS and DNS service

discovery have been proposed in Bonjour [25]. However, DNS servers do not support service filtering via search filters [61], meaning that all relevant service information needs to be sent to the client who performs service selection. This is inefficient if many services match the search request.

Discovery in P2P systems. Recently, discovery in peer-to-peer (P2P) systems has attracted great interest. In centralized P2P systems such as Napster [86], registries are used for discovery. In distributed P2P systems such as Gnutella [53], guided depth-first searches or restricted breadth-first searches are used for discovery. Recent P2P systems such as JXTA [138] and KaZaA [66] organize nodes into a hierarchy by using specialized nodes called hubs or supernodes, which provide a middle ground between the decentralized Gnutella model and the centralized Napster approach. Second generation P2P systems such as Chord [125], Pastry [109], Tapestry [143], and Content Addressable Networks (CAN) [104] use Distributed Hash Tables (DHTs) to build scalable and symmetric systems that have no centralized control or hierarchical organization. Current DHT-based P2P systems retrieve data based on a unique identifier, which is good for discovering files with a unique name. However, it is an open issue whether DHTs can efficiently support attribute-based searches for generic service discovery.

2.2.3 Service Location Protocol

The Service Location Protocol (SLP) [59] provides a flexible framework for service discovery in IP networks. It supports both registry-centric and peer-to-peer discovery models, and enables powerful service filtering and browsing. SLP uses general URLs [21] and the "service:" URL scheme [58] to specify *service locations* (also known as service access points). Each service has a *service type*, e.g., *ftp://ftp.example.com* is

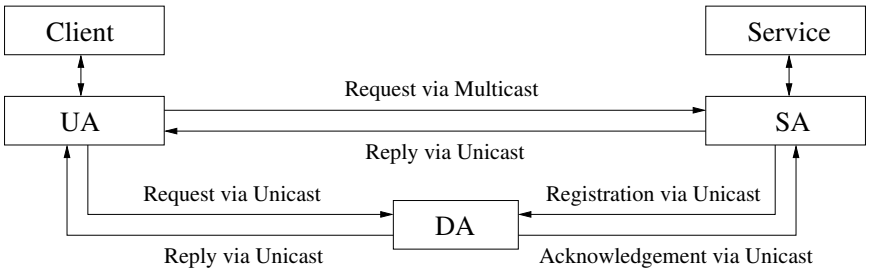


Figure 2.1: SLP system architecture: User Agents (UAs) initiate service discovery on behalf of service users by querying all SAs via multicast or a DA (if available) via unicast; Service Agents (SAs) work on behalf of service providers by responding directly to UA queries, and registering with DAs (if exist); and Directory Agents (DAs) serve as centralized information repositories by accepting SA registrations and answering UA queries.

an FTP service, and *service:printer:lpr://printer.example.com* is a printing service. Service properties are described via a list of attribute-value pairs, such as "speed = 15 ppm, resolution = 1200 dpi" for a printing service. SLP uses *service scopes* to arrange services into groups, which can indicate geographic locations such as "New York", administrative groupings such as "Law School", or other categories such as "Emergency". Each service registration is valid only for its specified *service lifetime* such as 2 hours, and will be removed from service registries when it has expired. In other words, SLP service registrations are soft states, and need to be refreshed periodically.

SLP has three types of entities: User Agents (UAs), Service Agents (SAs), and Directory Agents (DAs). Figure 2.1 illustrates their relationships.

User Agents. UAs initiate service discovery on behalf of service users by querying all SAs via multicast or a DA (if available) via unicast. UAs use three types of SLP messages: a service type request (`SrvTypeRqst`) message to get a list of available service types in a service type reply (`SrvTypeRply`) message, an attribute request (`AttrRqst`) message to get a list of attributes for

a given service type or service instance in an attribute reply (`AttrRply`) message, and a service request (`SrvRqst`) message with a search filter (or attribute predicate) specifying characteristics of the desired service to get a list of URLs giving the locations of matching services in a service reply (`SrvRply`) message. `SrvTypeRqst`, `SrvTypeRply`, `AttrRqst` and `AttrRply` messages allow a client to browse available service types and their attributes, which can be used to construct service queries in `SrvRqst` messages. Given the desired service type, and a set of attributes describing the service, SLP derives the service access points (URLs) for clients.

Service Agents. SAs work on behalf of service providers by responding directly to UA queries, and registering with DAs (if exist) via service registration (`SrvReg`) messages. SAs can also deregister services from DAs using service deregistration (`SrvDeReg`) messages.

Directory Agents. DAs serve as centralized information repositories by accepting SA registrations and answering UA queries. DAs can be discovered in two ways. For passive DA discovery, UAs and SAs simply listen for unsolicited DA advertisement (`DAAdvert`) messages sent periodically by DAs to an administratively scoped multicast address [81]. UAs and SAs can actively discover DAs by multicasting a DA discovery `SrvRqst` message whose service type is "service:directory-agent". DAs answer each DA discovery request with a unicast `DAAdvert` message.

SLP achieves scalability by using DAs and service scopes, and thus efficiently supports service discovery in systems of different scales. In small SLP deployments, DAs are usually not needed. UAs multicast requests to all SAs, and SAs respond via unicast. Since this multicast-based discovery cannot scale to a large number of SAs

and UAs, DAs are introduced in medium-sized SLP deployments, where SAs register services with DAs, and UAs search services at DAs, all using unicast. In large SLP deployments, DAs are arranged into different scopes to provide further scalability, e.g., services in the Law School and Business School of Columbia University can be assigned to different scopes.

2.3 Mesh Enhancement

2.3.1 Motivation

DAs allow SLP to scale to large deployments that may span large geographic regions. To avoid a single point of failure, each scope needs to have multiple DAs. However, SLP DAs do not interact with each other, thus SAs are required to register services with all DAs in their scopes. This simple approach has two disadvantages. First, it places too heavy a burden on SAs since they not only need to discover and register with all existing DAs, but also need to re-register when new DAs are discovered or old DAs are found to have rebooted. In other words, an SA needs to constantly monitor all DAs in its scope. This burden becomes an issue of efficiency and scalability when many devices provide services and each of them uses an SA. Secondly, in large deployments it is hard to guarantee that all SAs can discover all DAs in their scopes, leading DAs in the same scope to have inconsistent registrations.

To remedy this situation, we designed the SLP mesh enhancement (mSLP) [156, 155]. The rationale behind mSLP is that distributing registrations to multiple DAs should be taken care of by DAs instead of by SAs because there are far fewer DAs than SAs. It is more efficient and scalable for SLP to have a smaller number of powerful DAs but many lightweight SAs.

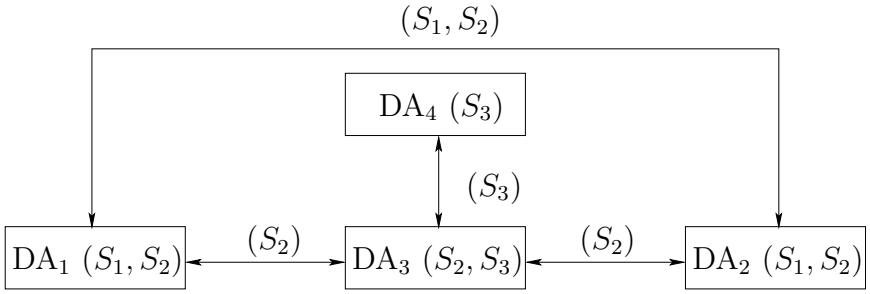


Figure 2.2: An example of the mSLP scope-based fully-meshed peering DA architecture for four DAs (DA$_1$ to DA$_4$) and three scopes (S_1 to S_3), where an edge between two DAs means that they are peers.

2.3.2 Design Overview

mSLP defines a scope-based fully-meshed peering DA architecture, which has two major components: peer relationship management and registration propagation control. In mSLP, DAs that share one or multiple scopes are *peers*. Each pair of peer DAs maintain a single peering connection between them no matter how many scopes they share. A peering connection is a persistent connection (e.g., TCP) that provides reliable and ordered transfers between two peers. For each scope, all DAs that serve the scope form a fully-meshed peer relationship, similar to the Internal Border Gateway Protocol (IBGP) [105]. Figure 2.2 shows an example of this peering DA architecture for four DAs (DA$_1$ to DA$_4$) and three scopes (S_1 to S_3), where an edge between two DAs means that they are peers. To keep consistent registrations for their shared scopes, two peer DAs exchange new registrations via their peering connection by using direct forwarding and anti-entropy; these two mechanisms will be described further in Section 2.3.4.

For simplicity and reliability, mSLP employs a full-mesh topology for its peering DA architecture. We anticipate that each scope has a small number of DAs, thus mSLP should be sufficient for a mesh size on the order of tens or below. Moreover,

large DA meshes can be avoided by splitting scopes. For example, if scope S has n DAs and n is too large, we can split S into two finer scopes S_1 and S_2, with n_1 DAs for S_1 only, n_2 DAs for S_2 only, n_3 DAs for both S_1 and S_2, and $n_1 + n_2 + n_3 = n$. In this way, instead of having a large full mesh of size n, now we have two smaller full meshes of size $n_1 + n_3$ and $n_2 + n_3$, respectively. Accordingly, a service registration that previously targets for scope S now needs to be registered under both S_1 and S_2.

Another important design consideration for mSLP is to be fully backward compatible with SLP. As a lightweight enhancement to SLP, mSLP only defines a new `DAAdvert` attribute "mesh-enhanced", a new message extension called mesh forwarding (`MeshFwd`), and a new message type called anti-entropy request (`AntiEtrpRqst`). An SLP DA can be mesh-enhanced by carrying the "mesh-enhanced" attribute keyword in its `DAAdvert` message and supporting the mSLP functionality without affecting its old functionality. Mesh-enhanced DAs can be deployed incrementally and co-exist with legacy SLP DAs in the same system.

mSLP offers a number of advantages. First, SA registrations can be simplified. No matter how many DAs are present in a scope, an SA only needs to discover, monitor, and register with any one of them for that scope. Registrations will then be propagated automatically to other DAs. Secondly, consistency among peer DAs can be improved as they periodically reconcile their inconsistent registrations. Furthermore, newly booted and rebooted DAs can catch up on all new registrations at once from their peers purely through DA interaction, without involving SAs. Finally, fewer TCP connections are needed when SAs register with DAs via TCP. Note that service registrations in SLP can be performed via either TCP or UDP, but a registration needs to use TCP if it is too large to fit into a UDP packet. Consider a scope that has n SAs and m DAs. In SLP, each SA needs to connect to each DA and register, thus nm TCP connections are needed. But in mSLP, each SA only needs to

connect to one DA in the full mesh of m nodes and register, then registrations are propagated through the DA mesh, therefore only $n + m(m-1)/2$ TCP connections are needed. Given any $n > m \geq 2$, we have $nm > n + m(m-1)/2$. For example, if $n = 100$ and $m = 10$, then 1000 TCP connections are needed in SLP, but only 145 such connections are needed in mSLP.

2.3.3 Peer Relationship Management

In mSLP, a DA maintains a peer relationship to each of its peers. The peer relationship management involves five aspects: learning about new peers, establishing a peering connection, exchanging peer information, maintaining a peer relationship, and tearing down a peer relationship.

2.3.3.1 Learning about New Peers

In mSLP, a DA can learn about its peers via static configuration, DHCP [95], and `DAAdvert` multicast and unicast.

2.3.3.2 Establishing a Peering Connection

To establish a peering connection from DA_1 to DA_2, DA_1 first needs to get DA_2's `DAAdvert`, then initiates a peering connection to DA_2, and then sends its `DAAdvert` along this peering connection to DA_2. Figure 2.3 illustrates these three steps. The last step is important because it ensures that DA_2 will have DA_1's `DAAdvert`, and thus enables DA_2 to identify the peering connection initiated by DA_1 based on that the advertised IP address in the `DAAdvert` is the same as the sender's IP address. Note that there is a small possibility that a pair of peering connections might be created between two peers if they try to initiate a connection to each other almost at the same

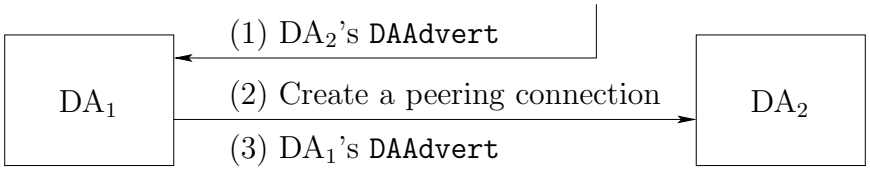


Figure 2.3: Steps for establishing a peering connection between DA_1 and DA_2: (1) DA_1 gets DA_2's `DAAdvert`, (2) DA_1 initiates a peering connection to DA_2, and (3) DA_1 sends its `DAAdvert` along this peering connection to DA_2.

time. To avoid this kind of inefficiency, when a DA identifies a new peering connection initiated by a peer, it should check whether it has already initiated another peering connection to the peer. If this is the case, then the DA with a lower-numbered IP address should terminate the connection it has initiated. In this way, only a single peering connection will be established between DA_1 and DA_2.

2.3.3.3 Exchanging Peer Information

After two DAs have established a peering connection, they exchange information about their existing peers by forwarding `DAAdvert` messages of their existing peers to each other via the peering connection. This enables both DAs to learn about new peers incrementally.

2.3.3.4 Maintaining a Peer Relationship

To maintain a peer relationship, a DA periodically sends its `DAAdvert` message to its peer DA via the peering connection. This keep-alive mechanism enables a DA to detect network partitions and peer crashes.

2.3.3.5 Tearing Down a Peer Relationship

A DA will tear down a peer relationship when its peer's `DAAdvert` message has timed out, when it has received its peer's shutdown message, or when its peer has closed the peering connection.

2.3.4 Registration Propagation Control

mSLP constrains registration propagations among peer DAs in three ways. First, two peer DAs exchange registrations only in their shared scopes. This calls for the support of scope-based partial replication among peer DAs. For example, as DA_1 and DA_3 in Figure 2.2 share one scope S_2, they exchange registrations only in scope S_2. Note that a multi-scoped registration needs to be propagated properly to all corresponding scopes. For example, consider that DA_3 in Figure 2.2 receives a multi-scoped registration in scope S_2 and S_3, then this registration needs to be propagated to DA_1, DA_2, and DA_4 since all of them serve part of the registration scopes. Secondly, only new registrations are exchanged between two peer DAs. To support this functionality, each registration needs to be labeled correctly so that a DA can decide which registrations a peer has already had, and the DA only sends those registrations the peer does not have yet. Finally, for each registration only a new version can overwrite an old one. Note that different versions of the same registration have the same service URL. When registrations are propagated among DAs, their arrival orders at a DA cannot be used to resolve different versions of the same registration. For example, assume that SA_1 sends a registration R_1 to DA_1 first, and a new version of the same registration R_2 to DA_2 later. When R_1 and R_2 are propagated, DA_2 receives R_1 later than R_2, but R_1 should not overwrite R_2 at DA_2 since R_2 is a newer version.

To satisfy the first constraint, a DA just needs to compare the scopes of each reg-

istration with the scopes of each peer DA, and propagates each registration properly to the corresponding peer DAs. To satisfy the last two constraints, additional control information is needed. Thus, we designed the mesh-forwarding (`MeshFwd`) extension, which carries two pieces of control information for each registration: an *accept-id* assigned by its accept DA (the first DA that accepts the registration) and a *version-timestamp* assigned by its originating SA (the SA that initiates the registration). An accept-id has two components: an accept-da (a unique identifier of the accept DA) and an accept-timestamp. All accept-timestamps assigned by the same DA must be monotonically increasing. Therefore, all accept-ids are unique; they define a total order for all registrations accepted by the same DA and a partial order for all registrations accepted by all DAs. mSLP uses accept-ids to control registration propagations so as to ensure that any registration accepted by any DA is distributed to all DAs in the registration scopes exactly once. Specifically, a DA propagates registrations in increasing order of their accept-ids, i.e., registrations accepted by the same DA are propagated in increasing order of their accept-timestamps, and registration accepted by different DAs may be propagated in any order. Similarly, all version-timestamps assigned by the same SA must be monotonically increasing. Since any SLP registration is only updated by one SA, using version-timestamps is sufficient to identify the most recent version for any registration.

mSLP has two goals in propagating registrations among peer DAs. The first one is to make service information highly available. Compared with availability, inconsistency among peer DAs is less of a concern. Therefore, an asynchronous replication model is more appropriate, where a registration update is delivered to one DA first, then it is propagated to other peer DAs later. Another goal is to propagate registration updates received by each DA to other peer DAs as quickly as possible in order to minimize inconsistency among peer DAs. To achieve these two goals, mSLP dis-

tributes registrations among peer DAs in two ways: anti-entropy and direct forwarding. In terms of providing high availability using lazy replication, the anti-entropy [96, 54, 147] mechanism is more efficient than the gossip technique [71].

Anti-entropy is used for exchanging initial registrations when two peer DAs get to know each other for the first time, and for catching up on new registrations after failures. A DA initiates an anti-entropy session by sending an `AntiEtrpRqst` message to a peer. Then the peer replies with all requested new registrations in increasing order of their accept-ids, and sends a service acknowledgment (`SrvAck`) message at the end of the batch of new registrations to indicate that processing of the corresponding `AntiEtrpRqst` message has been completed. While in anti-entropy, new registrations are pulled by the receiving DA via an `AntiEtrpRqst` message and are sent in a batch, in direct forwarding, new registrations are pushed by the sending DA and are sent individually. More specifically, after a DA has sent all new registrations accepted by itself to a peer via anti-entropy, the DA starts to forward any further incoming registration accepted by itself directly to the peer. This direct forwarding continues as long as the peer is alive and there is no failure. Note that the direct forwarding of a registration only goes one hop from its accept DA (say, DA_1) to all DA_1's peers that are in the registration scopes.

For anti-entropy, mSLP supports two types of sessions: complete and selective. Complete anti-entropy [96, 54] is the traditional way to perform anti-entropy, in which all registrations that have an accept-id greater than any specified accept-id in the `AntiEtrpRqst` or have an accept-da not specified in the `AntiEtrpRqst` are solicited. Selective anti-entropy [147] (described further in Chapter 4) is our proposed new way to perform anti-entropy, in which only registrations that have an accept-id greater than any specified accept-id in the `AntiEtrpRqst` are solicited. Selective anti-entropy enables two parties to perform partial anti-entropy in the granularity of one accept-

da, i.e., all registrations accepted by the same DA. Selective anti-entropy generalizes traditional anti-entropy with added flexibility; it supports scope-based replication in mSLP and complex partial replication in general.

Next, we use an example to show how selective anti-entropy differs from complete anti-entropy. Consider a scope that has three DAs: DA_1, DA_2 and DA_3. DA_2 has registrations accepted by DA_1, DA_2, and DA_3. If DA_1 sends a selective `AntiEtrpRqst` to DA_2 using an accept-id list as $\{(DA_2, T_2)\}$, then DA_1 only requests registrations that are accepted by DA_2 and have an accept-timestamp greater than T_2. In contrast, if DA_1 sends a complete `AntiEtrpRqst` to DA_2 using the same accept-id list as before, then DA_1 requests all registrations accepted by DA_1 and DA_3, in addition to those registrations accepted by DA_2 and having an accept-timestamp greater than T_2.

2.4 Remote Service Discovery

2.4.1 Motivation

SLP is designed for local service discovery within one administrative domain. For service discovery beyond the local domain, there are two cases. The first one is remote service discovery, which is to discover services in a given remote (i.e., non-local) DNS domain. For this type of service discovery, we have a specific DNS domain. The second case is to discover services based on geographic locations or other properties that are not specific to any DNS domain, such as finding wireless network access points in the New York metropolitan area. This type of service discovery is more difficult since it spans many DNS domains.

DNS SRV [56] can specify the server locations for a specific service, transport protocol, and DNS domain, which provides good support for remote service discovery.

However, if multiple servers are discovered via DNS SRV for a service, only priority and weight can be used to make a selection. If additional service properties such as cost, speed, and service quality need to be considered in the selection process, DNS SRV becomes insufficient.

Using SLP and DNS SRV together can provide better support for remote service discovery. First, a UA uses DNS SRV to find SLP DAs at a remote DNS domain. Then, the UA uses SLP to query one of those DAs to discover desired services in that domain just as in the local domain. In this way, we can avoid the limitations in using SLP and DNS SRV separately. On one hand, without DNS SRV, an SLP UA needs to depend on static configuration to learn about remote DAs because DHCP and multicast DA discovery are not generally applicable beyond the local administrative domain. On the other hand, without SLP, DNS SRV has limited support for service selection.

2.4.2 Design

To support remote service discovery in SLP, we defined DNS SRV Resource Records (RRs) for SLP DA services [158], which can map a given DNS domain name to remotely accessible (i.e., accessible from the Internet) SLP DAs in that domain.

As SLP scopes are intended to be used only within one administrative domain, they are likely incomprehensible to users outside of the administrative domain. Thus, any remotely accessible service must be registered in the "default" scope, but it may be registered in other scopes at the same time. Similarly, all DAs advertised via DNS SRV must serve the "default" scope, but they may serve other scopes at the same time. As a result, users wishing to discover services at a remote DNS domain should only search the "default" scope.

Due to reasons such as security considerations, load controls, and charging requirements, a domain normally just wants a chosen subset of its services to be accessible from the Internet. This administrative differentiation of service usages is achieved by having chosen services from any scope registered in the "default" scope and exposed to the Internet. The advantage for a domain to put all remotely accessible services in a single "default" scope is that remote (non-local) users do not need to differentiate scopes to discover desired services in that domain. Note that the services discovered via DNS SRV and remote SLP DAs may not necessarily be remotely accessible.

Our proposed remote service discovery using SLP and DNS SRV facilitates discovering services at a remote DNS domain if the domain name is known via a priori knowledge. However, it is not intended to solve the problem of Internet-wide service discovery [11].

2.5 Preference Filters

2.5.1 Motivation

Since an SLP server, either a DA or an SA, does not perform any processing on search results, all matching results are returned from the server to the client with no particular order. This works fine for small SLP deployments, but may not scale to large deployments. Consider the following scenarios. First, if too many service entries match a search request, the search results may overload client network and storage resources. Secondly, a client may just want to find a few results that satisfies its requirements rather than all of them. Sending unneeded results to the client will waste network and server resources. Finally, a client may want to weigh the relative suitability of matching results based on some criteria, which calls for sorting search

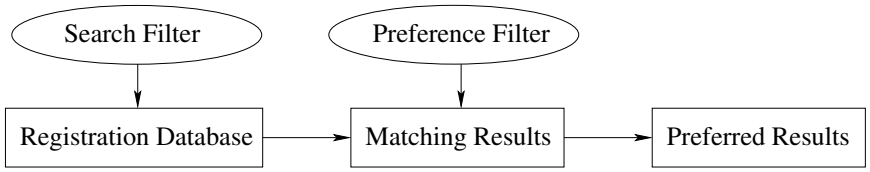


Figure 2.4: The processing of a `SrvRqst` that has a search filter and a preference filter

results. Sorting at the server is more efficient than sorting at the client since the former does not need to pass the attributes of matching results to the client just for sorting purposes. Reducing the amount of data transferred to the client is useful when the client uses a low bandwidth channel, such as a wireless channel. A good example showing the need for processing search results at SLP servers is the best-match search, such as finding a printer that has the shortest queue. For this discovery, an SLP UA needs to get information for all printers, sort them based on the queue length attribute, and choose the one with the shortest queue. This procedure is inefficient when there are many printers.

2.5.2 Design

Preference filters [157, 146] are designed to facilitate flexible processing of search results at the server, based on the client requirements in terms of the size and order of results. Preference filters are specified via SLP extensions attached to `SrvRqst` messages. Figure 2.4 shows the processing of a `SrvRqst` message that has a search filter and a preference filter. First, the search filter is applied to the service registration database, which generates a set of matching results. Then the preference filter is applied to the matching results, which generates a set of preferred results.

Although the format of preference filters could be designed in a way similar to SLP and LDAP search filters [61], we employ a simpler approach based on composition.

We choose *select* and *sort* as two basic preference filters and design the corresponding SLP `Select` and `Sort` extensions [157] for specifying them, then we use these two basic filters to compose generic preference filters.

The `Select` extension is used by a UA in the `SrvRqst` message to limit the maximum number (say, n) of results to be returned, and is used by an SLP server in the corresponding `SrvRply` message to indicate the total number (say, m) of search results. If $n < m$, then only the first n search results are returned, otherwise all m search results are returned. As a special case, a UA may set n to 0 to obtain the number of search results without retrieving the results themselves.

The `Sort` extension carries a sort key list. Each sort key has a key name (i.e., an attribute name), a type specifier ("s" for string and "i" for integer), an ordering specifier ("+" for increasing and "-" for decreasing), and an optional reference value. Although SLP has five attribute types, namely integer, string, boolean, opaque, and keyword, we only consider integer sort and string sort because keyword attributes never need to be sorted as they have no values, and boolean and opaque attributes can be sorted as strings if needed. Integer keys may have a reference value, as in *speed:i:+:12*, causing the sort to be based on the distance to the reference value, here, 12.

A generic preference filter is a list of select and sort filters observing the following rules. First, two basic filters of the same type, whether select or sort, cannot be adjacent to each other. Secondly, if the number of sort and select filters is the same, the last one must be a select filter. Finally, for two select filters s_1 and s_2, if s_1 appears earlier than s_2, then the selected number of results specified in s_1 must be greater than that in s_2.

Next, we show some examples of preference filters, in which *select(number)* denotes a select filter and *sort(attribute:type:ordering:reference)* denotes a sort filter.

Finding the best match is accomplished via a sort filter followed by a select filter, e.g., "*sort(load:i:+), select(1)*" for the least loaded service, "*sort(speed:i:-), select(1)*" for the fastest service, and "*sort(price:i:+:12), select(1)*" for the service with a price closest to 12 charging units. Examples for other complex preference filterings include "*sort(speed:i:-), select(3)*" for the three fastest services, "*sort(speed:i:-,load:i:+), select(1)*" for the least loaded service among the fastest, and "*sort(speed:i:-), select(3), sort(load:i:+), select(1)*" for the least loaded service among the three fastest.

2.6 Global Attributes

2.6.1 Motivation

In SLP, service attributes describe service properties specific to certain service type, which we referred to as *local attributes*. In contrast, *global attributes* describe service properties common to all service types. Local attributes and global attributes differ in how they are defined, named, and used. Currently, SLP only supports local attributes in that each service type defines its own attribute set via a service template [58]; an attribute name is unique only within its service type (i.e., two different service types may use the same attribute name); and an attribute is always used along with its service type. As more service properties are identified as being common to all service types, such as transport protocol, we saw a need to enable global attributes in SLP for efficiency and advanced discovery. For example, without such a mechanism, a UA needs three steps to find all services supporting the Stream Control Transmission Protocol (SCTP) [123]: sending a `SrvTypeRqst` message to obtain a list of service types, then using a separate `SrvRqst` message to search services of each type, and finally combining the search results. As a `SrvRqst` message can only search services

of a single type, $n + 1$ searches are needed for n service types, which is inefficient if n is large.

2.6.2 Basic Design

To enable global attributes [146, 152] in SLP, we need to assign a separate namespace to global attributes, define them via attribute templates, and use them properly in searching services across multiple service types. First, a separate namespace is needed for global attributes. This is because a global attribute can be used with any service type. If it has the same name as a local attribute, then there will be confusion as to which is which. To follow the common practice of prefixing an attribute name with its service type, we use the "*service-*" prefix in global attribute naming. Note that XML [27] also uses prefixes to define its namespaces. Secondly, a global attribute is defined via an attribute template [152]. Any service type that uses a global attribute imports the attribute's definition into its service template, similar to the C `include` and Java `import` mechanisms. In this way, a global attribute only has one definition, and can be used consistently for all service types. Finally, a global attribute can appear in any place where a local attribute is appropriate. In a `SrvRqst` message, when local attributes are used, exactly one service type must be specified; but when only global attributes are used, multiple service types or a service type wildcard can be specified. Thus, using a single `SrvRqst` message can search services across multiple or all service types. For example, to find all services supporting SCTP, we can use a `SrvRqst` message that has a service type wildcard, and a search filter of "*service-transport-protocol=sctp*".

Using global attributes can improve SLP efficiency. First, global attributes only need to be defined once. Afterwards, they can be imported into any service template.

This avoids defining the same attribute repeatedly in different service templates, and ensures a consistent definition. Secondly, by using global attributes, a single `SrvRqst` message can search services across multiple service types, which is more efficient than using multiple `SrvRqst` messages, one for each service type.

2.6.3 Advanced Usages

Using global attributes can accelerate the standardization of common service properties and support advanced discovery scenarios. For example, we can define service identifier and device identifier as global attributes. *Service identifiers* and *device identifiers* are URIs [21] such as UUIDs [73]; each of them *uniquely* and *persistently* identifies a service or a device. We can use service identifiers and device identifiers to support the following discovery scenarios.

Supporting URL changes. While Jini [134] and UDDI [129] use service identifiers as service keys, SLP uses service URLs as service keys. Since a service may change its URLs (e.g., when the service moves), retrieving a service based on its service URLs may not always be feasible. To remedy this situation, we can define service identifier as a global attribute so that a client can always find a service based on its service identifier.

Discovering multi-access-point services. A multi-access-point service provides the same service via different access points that reside at the same device. For example, a multi-protocol printer that supports IPP [60] and LPR access protocols may have two URLs *service:printer:ipp://mpp.example.com* and *service:printer:lpr://mpp.example.com*. We can use service identifiers to discover multi-access-point services as follows. A multi-access-point service advertises each access point separately, but all advertisements use the same service iden-

tifier to indicate that they point to the same service. A client can discover all advertisements of a multi-access-point service by specifying the service identifier and the service type (or a service type wildcard) in a `SrvRqst` message.

Discovering multi-function devices. A multi-function device provides different types of services at the same device. For example, a device that supports printing and scanning services may have two URLs *service:printer://print.example.com* and *service:scanner://scan.example.com*. Using device identifiers, we can discover multi-function devices as follows. A multi-function device advertises each service type separately, but all advertisements use the same device identifier to indicate that they reside at the same device. A client can discover all advertisements of a multi-function device by specifying the device identifier and a wildcard service type (or all the service types the device supports) in a `SrvRqst` message.

Discovering replicated services. A replicated service provides the same service at different devices. Using service identifiers and device identifiers together, we can discover replicated services as follows. A replicated service advertises the same service at each device separately, and all advertisements use the same service identifier but different device identifiers. Note that a replicated service uses different device identifiers in its advertisements whereas a multi-access-point service uses the same device identifier in its advertisements.

2.7 Implementation

We have implemented our enhanced SLP in Java 1.4, which includes a stand-alone DA server and an integrated UA/SA tool. Besides the basic SLP protocol stack,

our open-source implementation [145] supports the SLP mesh enhancement, remote service discovery, preference filters, and global attributes. Next, we outline how these SLP enhancements are implemented.

To support the SLP mesh enhancement, DAs need to manage peer relationships by maintaining a peer table, and need to control registration propagations by maintaining a summary vector for all registrations as well as an accept-id and a version-timestamp for each registration. At the same time, SAs need to use the `MeshFwd` extension in their registrations, but UAs do not need to be changed.

To support remote service discovery, a domain needs to list its remotely accessible DAs via DNS SRV. At the same time, UAs need to perform DNS queries to obtain SRV records so as to discover DAs at given remote DNS domains.

To support preference filters, an SLP server needs to adjust its processing of `SrvRqst` messages as follows. For a `SrvRqst` message with a preference filter, the filter is ignored during the search, and then the filter is applied to the search results. When the filter has multiple select and sort filters, they must be processed in order, with the output of one filter as the input of the next filter. The output of the last filter is returned to the client.

Similarly, to support global attributes, an SLP server needs to adjust its processing of `SrvRqst` messages as follows. For a `SrvRqst` message that uses local attributes, it should have exactly one service type, and is handled as before. For a `SrvRqst` message that uses only global attributes, it may have multiple service types or a service type wildcard. In this case, the service type information is ignored during the search, and then those search results that do not match any of the specified service types are discarded.

2.8 Evaluation

To evaluate the proposed enhancements, we carried out experiments in our local area network and on PlanetLab [99]. In our local area network, we used a cluster of 30 machines; each machine had a 1 GHz Intel Pentium III CPU and 512 MB of memory. They all ran Redhat 9.0 with Linux kernel 2.4.20-20.9, and were connected via 100 Mb/s fast Ethernet. At the time of the experiments, PlanetLab consisted of more than 300 nodes all over the world; each node had a CPU of at least 1 GHz clock rate and had at least 1 GB of memory. They all ran Redhat 9.0 with Linux kernel 2.4.22-r3_planetlab, and used PlanetLab software 2.0. PlanetLab nodes had four types of network connections: DSL, Internet2, North American commodity Internet, and outside North America.

We used our open-source SLP DA implementation written in Java [145], and implemented SLP SAs and UAs using C. We evaluated each enhancement individually by comparing the results when the enhancement was enabled with that when the enhancement was disabled. Note that we did not obtain quantitative results for remote service discovery because this enhancement extended SLP functionality without affecting discovery performance.

2.8.1 Mesh Enhancement

To show the benefits of using mSLP, we measured consistency between two peer DAs after one recovered from failures. In this experiment, we ran two DAs, one SA, and one UA, all at local machines, and they were all in the same scope. The SA performed n different registrations repeatedly with a fixed interval of I_r seconds, and the lifetime of each registration was set to $(n+1)I_r$ seconds. Thus, each registration was refreshed every $R = nI_r$ seconds, and each DA should have n registration entries

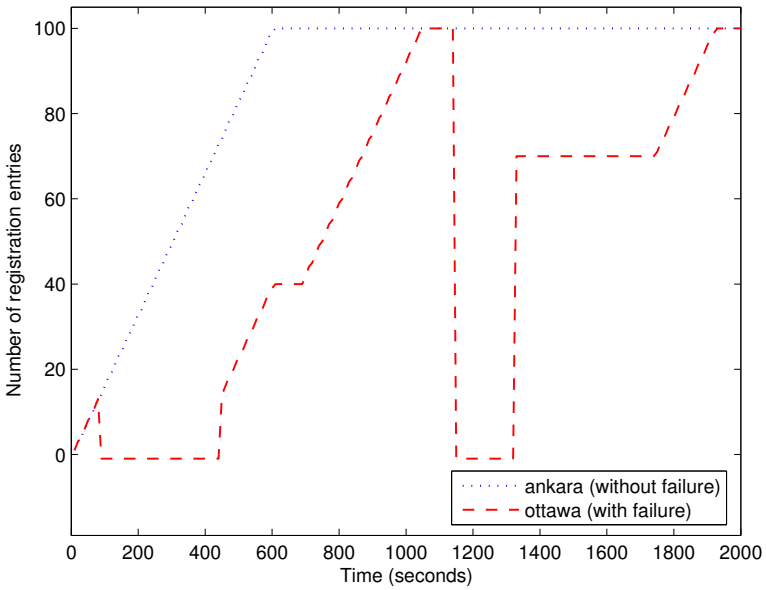


Figure 2.5: Without using mSLP: consistency between two peer DAs at host *ankara* and *ottawa* after host *ottawa* recovered from two failures during 90 to 440 seconds and 1150 to 1320 seconds

at steady state. The UA queried both DAs with a fixed interval of I_q seconds to find out how many registration entries each DA had. Since the UA did not need to retrieve the entries themselves, it attached a preference filter "*select(0)*" to each query. The SA or UA timed out a registration or query if it could not get a response after T seconds, in which case the DA was assumed to have failed, i.e., the DA had crashed or had been disconnected from the network. The experimental parameters were set as follows: $n = 100$, $I_r = 6$ seconds, $R = nI_r = 600$ seconds, $I_q = 10$ seconds, and $T = 1$ second.

In the first experiment, mSLP was disabled, and the SA registered with both DAs. For the run length of 2000 seconds, the DA at host *ankara* worked properly, but the DA at host *ottawa* was unavailable during two time periods, from 90 to 440 seconds and from 1150 to 1320 seconds. Figure 2.5 shows the number of registration

entries at both DAs as sampled by the UA. For each sample, if the response from a DA timed out, then the UA marked the number of registration entries at that DA as -1. We observe that host *ottawa* missed m service registrations when it failed during $[t_0, t_1]$, where $m = n(t_1 - t_0)/R$ assuming $t_1 - t_0 \leq R$. When host *ottawa* recovered from failures, it had m registration entries less than host *ankara* during $[t_1, t_0 + R]$. Without suffering new failures, host *ottawa* caught up with missing entries during $[t_0 + R, t_1 + R]$, and had the same registration entries as host *ankara* after $t_1 + R$. Thus, host *ottawa* took a recovery interval of R seconds to catch up with all missing registrations. When the UA queried host *ottawa* during $[t_1, t_1 + R]$, it got incomplete service information.

To quantify the relationship between missing entries and failure duration, we use μ to denote the average rate of missing entries during the first R seconds after a DA recovered from failures, and use ρ to denote the ratio of the failure duration over recovery interval R. We first consider $\rho \in [0, 1]$. In the above experimental setup, the missing entries in $[t_1, t_0 + R]$ is $n\rho$, and the average missing entries in $[t_0 + R, t_1 + R]$ is $n\rho/2$. Thus, we can compute μ as follows:

$$\mu = \frac{n\rho(1-\rho)R + n(\rho/2)\rho R}{nR} = \rho(2-\rho)/2$$

Figure 2.6 shows the relationship between μ and ρ for $\rho \in [0, 1]$. We observe that μ increases as ρ increases, and $\mu = 0$ when $\rho = 0$, and $\mu = 50\%$ when $\rho = 1$. If $\rho > 1$, the DA would expire all registration entries when it recovered. Thus, $\mu = 50\%$ when $\rho > 1$, which is the same as $\rho = 1$.

In the second experiment, mSLP was enabled. For each registration, the SA only registered with one DA by randomly choosing one DA to register. If the response from the chosen DA timed out, then the SA registered with another DA. Figure 2.7 shows

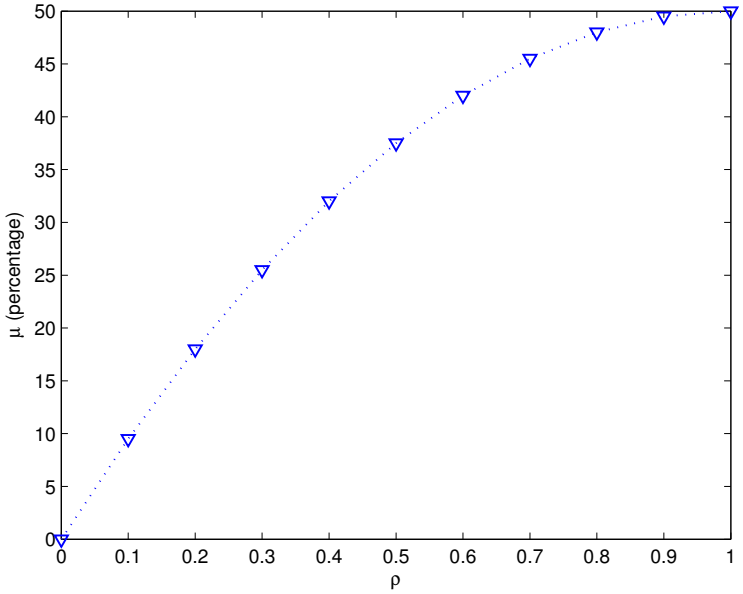


Figure 2.6: The relationship between μ and ρ for $\rho \in [0, 1]$, where μ is the average rate of missing entries during the first R seconds after a DA recovered from failures, and ρ is the ratio of failure duration over recovery interval R.

the experimental results, where the failure scenario was the same as in the first case. We observe that when both DAs were alive, each registration was propagated from one DA to another DA automatically via mSLP. Also, when host *ottawa* recovered from failures, it got all missing registrations at once from host *ankara* via mSLP quickly, which took about 100 ms in our experiments. Thus, when both DAs were alive, the UA could always get the same consistent reply whether it queried host *ottawa* or *ankara*. In other words, mSLP can effectively fix the inconsistency problem among peer DAs after one of them recovers from failures.

To quantify the consistency improvement by using mSLP, consider the probability that a UA gets incomplete service information. Assuming each DA has an availability of p_1, and $\rho \leq 1$, then the probability that a DA has incomplete service information is $p_2 = (1 - p_1)/\rho$. Thus, the probability that a UA gets incomplete service information

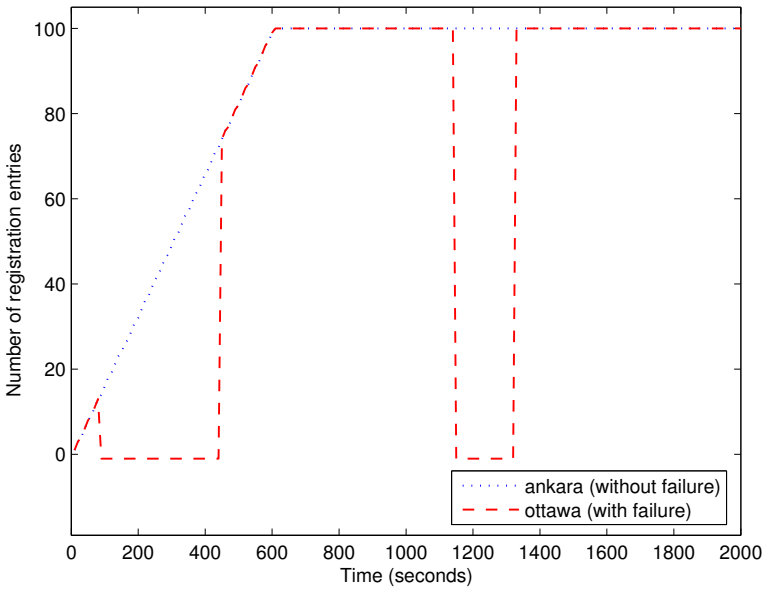


Figure 2.7: Using mSLP: consistency between two peer DAs at host *ankara* and *ottawa* after host *ottawa* recovered from two failures during 90 to 440 seconds and 1150 to 1320 seconds

from two DAs is $p_3 = 50\%(p_2 + (1 - p_1)p_2)2 = p_2 + (1 - p_1)p_2$ when mSLP is not used, but is $p_4 = p_2^2$ when mSLP is used. For example, if $p_1 = 99.9\%$, and $\rho = 0.1$, then $p_2 = 1\%$, $p_3 = 1.001\%$, and $p_4 = 0.01\%$, where the probability that a UA gets incomplete service information has been reduced by two orders of magnitude by using mSLP.

2.8.2 Preference Filters

To show the benefits of using preference filters, consider the response time of a query when many entries match the query. For example, in DotSlash [150], different web servers register with mSLP DAs, and a web server discovers and utilizes spare capacity at other web servers to relieve its load spikes. Although many registered web servers may have spare capacity, a web server only needs to use a few of them in case of load

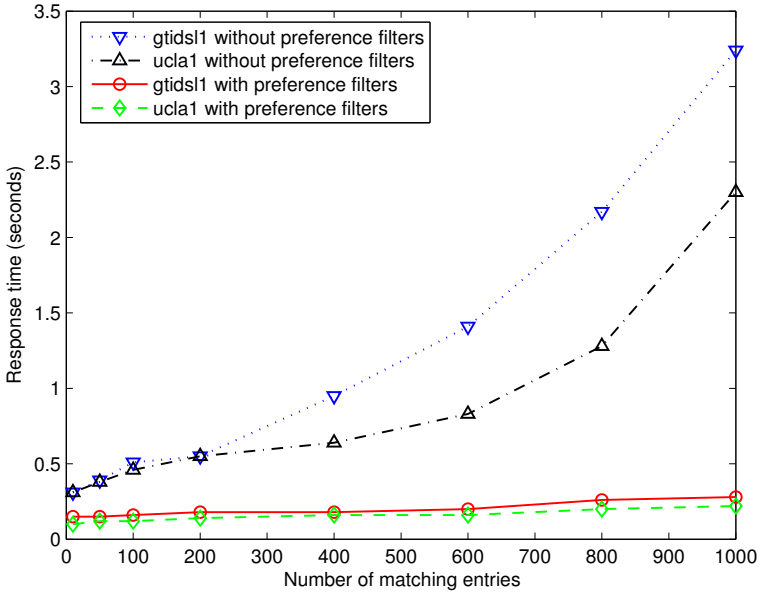


Figure 2.8: Using preference filters versus without using preference filters: the response time measured from two PlanetLab nodes, *gtidsl1* and *ucla1*, where node *gtidsl1* was behind a DSL line, and node *ucla1* connected to Internet2.

spikes. In this experiment, we ran one DA and one SA at local machines, and ran two UAs on PlanetLab nodes, one UA at node *gtidsl1* which was behind a DSL line, and another UA at node *ucla1* which connected to Internet2. The SA registered with the DA to simulate service registrations performed by n web servers with different spare capacities, where n varied from 10 to 1000. The two UAs queried the DA to obtain information about spare capacities at available web servers. For each n, each UA queried in two cases: (1) without using preference filters, all n entries were retrieved, and (2) by using the following preference filters "*sort(spare-capacity:i:-), select(10)*", only entries with the ten largest spare capacities were retrieved. We used TCP in the first case since the reply size could be quite large, but used UDP in the second case since the reply size was small and fixed.

Figure 2.8 shows the experimental results. We observe that without using prefer-

ence filters, the response time increases as the number of matching entries n increases because the DA needs to process more entries and send more data to the UA. Further, when the reply size gets larger, the low bandwidth UA at node *gtidsl1* takes significantly more time to get the reply than the high bandwidth UA at node *ucla1*. In contrast, by using preference filters, the response time increases only slightly as n increases because the reply size is unchanged, but the DA needs sightly more time to process a larger number of matching entries. The big difference in response times between using and without using preference filters is mainly due to the difference of reply sizes in these two cases. When preference filters are used, the reply size is fixed, which is 927 bytes for all $n \in [10, 1000]$. But when preference filters are not used, the reply size increases as n increases, which is 920 bytes when $n = 10$, 9020 bytes when $n = 100$, and 90020 bytes when $n = 1000$.

2.8.3 Global Attributes

To show the benefits of using global attributes, consider the time used for completing a location-based query. For example, consider the problem of finding all services at a given location, where a number of different types of services exist, such as printer, projector, and fax. In this experiment, we ran one DA and one SA at local machines, and ran one UA at the PlanetLab node *gtidsl1*. The SA registered n types of services with the DA, where n varied from 1 to 20. The UA queried the DA to find services at a given location. Without using global attributes, the UA needed to use $n+1$ queries, where one query was needed for obtaining the service type list, and a separate query is needed for each service type. Using global attributes, the UA only needed to use one query by specifying a service type wildcard.

Figure 2.9 shows the experimental results. We observe that without using global

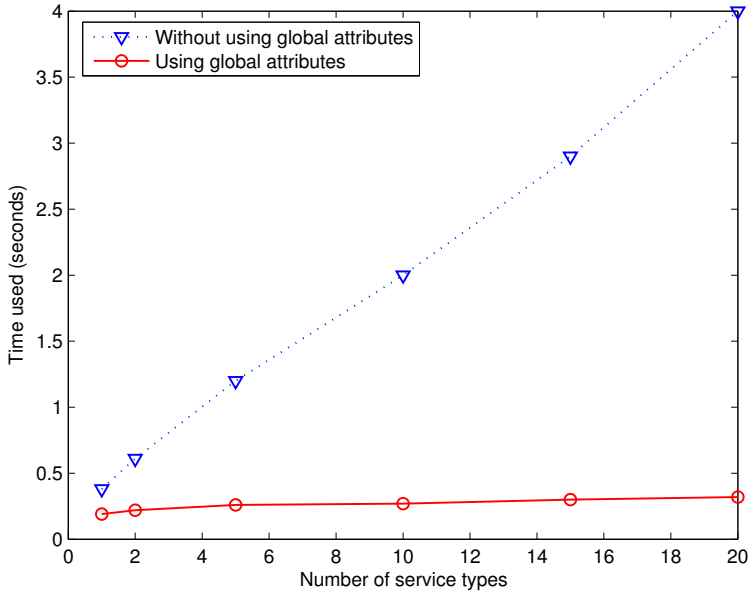


Figure 2.9: Using global attributes versus without using global attributes: the time used for performing location-based queries from a PlanetLab DSL node *gtidsl1*

attributes, the total time used increases as the number of service types increases. In contrast, by using global attributes, the time used is roughly unchanged. We also ran the above experiment using a local UA, and obtained a similar curve except that the time used by the local UA was about two orders of magnitude less than that of the UA at node *gtidsl1*.

2.9 Related Work

Fully-meshed peer relationships are used in IBGP [105]. While an IBGP node is only in one mesh, a multi-scoped mSLP DA may belong to multiple meshes. mSLP supports flexible selective anti-entropy [147] as well as traditional complete anti-entropy [96]. UDDI [129] also uses anti-entropy to replicate registries, but only for full replications, whereas mSLP supports scope-based partial replications.

DNS-based Service Discovery (DNS-SD) [36] proposes a convention for naming and structuring DNS resource records, which allows clients to discover service instances of desired service types at any DNS domain by using only standard DNS queries. Although our proposed remote service discovery and DNS-SD both support DNS-domain-based service discovery, i.e., are not restricted to local service discovery, they differ in terms of design goals and functionality. DNS-SD aims to facilitate service discovery by using DNS only, which is advantageous for easy deployment, but it does not support for service filtering via search filters, which is inefficient for service selection. In contrast, our system supports flexible and powerful service filtering by using SLP and DNS SRV together.

A number of service discovery systems such as Jini [134] and UDDI [129] support selecting and sorting search results, but none of them support generic preference filtering by composing these two basic operations. LDAP sort control [62] and paging control [139] come closest to our proposed preference filters. But LDAP sends all search results back to the client via the paging control mechanism, whereas our system simply sends selected number of search results to the client. Furthermore, our system supports reference-based sorting and supports composing multiple select and sort filters. In the context of service discovery, our preference filters are a simple case for the general issue of preference-based service selection [15]. Beyond service discovery, our preference filters are related to SQL *order by* statement.

Our work on SLP global attributes was motivated by Guttman's work [57] on using service identifiers as SLP service keys. However, Guttman's proposal needs to use hierarchical attributes when a service has multiple URLs with different properties. We believe that it is simpler to keep service URLs as service keys and define service identifiers as a global attribute. Similar to global attributes, Jini [134] defines a set of common entry classes in the *net.jini.lookup.entry* package. When a Jini search

specifies multiple interfaces, it finds services that implement all specified interfaces (logical "and"). In contrast, when an SLP `SrvRqst` message specifies multiple service types, it finds services of any specified type (logical "or").

2.10 Summary

This chapter described four new mechanisms for SLP: mesh enhancement that simplifies SA registrations and improves the consistency of peer DAs, remote service discovery that enables SLP users to discover services at remote DNS domains, preference filters that facilitate processing of search results in SLP servers, and global attributes that allow using a single query to search services across multiple types. These mechanisms can improve SLP efficiency and scalability, and enable SLP to better support new and advanced discovery. Although we discuss these techniques in the context of SLP, we expect that they can also be applied to other service discovery systems. The SLP mesh enhancement (mSLP), remote service discovery, and preference filters are now experimental RFCs (Request for Comments) [157, 156, 158].

Chapter 3

Selective Anti-Entropy

This chapter describes selective anti-entropy, a mechanism we used in SLP mesh enhancement. Since it is a general mechanism for high availability partial replication, we discuss it here separately. We first give some background on replication and anti-entropy, then describe the motivation and design of selective anti-entropy. After presenting the implementation and evaluation for selective anti-entropy, we discuss related work and give a summary.

3.1 Replication

Replication is an important technique for enhancing performance, availability, and scalability of distributed systems.

Based on how data are replicated among replicas, replication systems can be classified into two categories, namely full replication and partial replication. In full replication, all data are replicated to all replicas. We denote a full replication system with r replicas as $\mathcal{F}(r)$. In partial replication, the whole data set is partitioned into subsets referred to as *scopes*, and all data in the same scope are replicated to the same

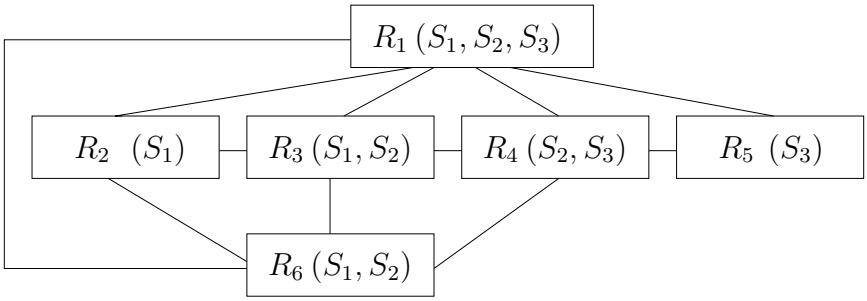


Figure 3.1: An example of $\mathcal{P}(6, 3)$, a partial replication system with 6 replicas and 3 scopes, where replicas are R_1 to R_6, scopes are S_1 to S_3, and an edge between two replicas means that they share scopes.

replica set. We denote a partial replication system with r replicas and s scopes as $\mathcal{P}(r, s)$, where a replica may serve any number of scopes (up to s), and a scope may have any number of replicas (up to r). Clearly, $\mathcal{F}(r)$ is just a special case of $\mathcal{P}(r, s)$ when all replicas serve all scopes. Figure 3.1 shows an example of $\mathcal{P}(6, 3)$, a partial replication system with 6 replicas and 3 scopes, where replicas are R_1 to R_6, scopes are S_1 to S_3, and an edge between two replicas means that they share scopes.

A partial replication system is more flexible and efficient than a full replication system. First, partial replication can help restrict access to data. For example, a company may have service registries for internal services, external services, and all services, which are used by internal users, external users, and administrators, respectively. Secondly, partial replication can reduce the amount of data transferred across wide area networks for geographically distributed replicas, which can lower telecommunications costs. For example, a company may have service registries located at its regional offices and its headquarters for regional services and all services, respectively.

A partial replication system is also more complex than a full replication system. For instance, all replicas in $\mathcal{F}(r)$ are equivalent, but replicas in $\mathcal{P}(r, s)$ can have four different types of relationships based on how they share serving scopes. We use

$S(R_x)$ and $S(R_y)$ to denote the serving scopes of replica R_x and R_y, respectively. The relationship between R_x and R_y can be equivalent, subset, overlap, or non-overlap, explained below.

Equivalent. R_x and R_y are equivalent if they serve the same scopes, i.e., $S(R_x) = S(R_y)$. For example, in Figure 3.1, R_3 and R_6 are equivalent.

Subset. R_x and R_y have a subset relationship if the serving scopes of one replica are a subset of the serving scopes of another replica, i.e., $S(R_x) \subset S(R_y)$ or $S(R_y) \subset S(R_x)$. For example, in Figure 3.1, R_4 and R_1 have a subset relationship.

Overlap. R_x and R_y have an overlap relationship if they have serving scopes in common but they do not have an equivalent or subset relationship, i.e., $S(R_x) \cap S(R_y) \neq \emptyset$, $S(R_x) \neq S(R_y)$, $S(R_x) \not\subset S(R_y)$, and $S(R_y) \not\subset S(R_x)$. For example, in Figure 3.1, R_3 and R_4 have an overlap relationship.

Non-overlap. R_x and R_y have a non-overlap relationship if they do not share any serving scope, i.e., $S(R_x) \cap S(R_y) = \emptyset$. For example, in Figure 3.1, R_2 and R_5 have a non-overlap relationship.

A partial replication system can be converted into several full replication systems. We first consider the case when the scopes in a partial replication system are disjoint, such as free services and paid services. For example, a partial replication system with one registry for free services, one registry for paid services, and one registry for all services can be converted into two full replication systems with two registries for free services and two registries for paid services. Note that the conversion keeps the same replication degree for each scope as before. However, such a conversion involves overhead. First, the conversion needs to use more registries than the original partial replication system. Secondly, a multi-scope query needs to be converted into

multiple single-scope queries, and the corresponding query result sets need to be combined into one result set. When the scopes in a partial replication system overlap, such as internal services and external services, the conversion involves additional overhead. First, a multi-scope service may need to register with more registries. Secondly, when a multi-scope query is converted into multiple single-scope queries, a multi-scope registration may appear in multiple query result sets. Thus, we need to remove duplicate registrations in combining the query result sets. Due to the overhead outlined above, we believe that it is useful to support partial replication.

3.1.1 Anti-Entropy

Anti-entropy [44] is an important mechanism for achieving eventual consistency among a set of replicas, where an update is accepted by one replica first, and then the update is propagated asynchronously to the remaining replicas. An important feature of anti-entropy is that two replicas only exchange new updates to reconcile their inconsistency. To achieve this, all updates need to be labeled correctly, and be propagated in certain order among replicas. Anti-entropy uses the following labeling scheme. Each update is assigned an *accept-id* by its *accept-replica*. The accept-replica for an update is the first replica that accepts the update. An accept-id has two components, an *accept-replica-id* and an *accept-timestamp*. The accept-replica-id is a unique identifier of the accept-replica, and the accept-timestamp is the timestamp when the update is accepted at its accept-replica. All accept-timestamps assigned by the same replica must be monotonically increasing. Two accept-ids are comparable only if they have the same accept-replica-id, and their order is determined by their accept-timestamps.

In anti-entropy, updates are propagated in increasing order of their accept-ids. Thus, each replica can use a *summary vector* to summarize the updates it has received.

This summary vector keeps the largest accept-timestamp for each accept-replica-id. For example, if replica R_i has a summary vector of $((R_1, t_1), (R_2, t_2), ..., (R_r, t_r))$, then R_i has received all updates accepted by replica R_j up to timestamp t_j, where $1 \leq j \leq r$.

Anti-entropy [96, 54] is a mechanism for high availability replication, but it was designed for supporting full replication. We enhanced it to support partial replication by allowing two replicas to selectively reconcile inconsistent data in a session.

3.2 Motivation

We refer to the existing anti-entropy algorithms [96, 54] as *complete anti-entropy* in that two replicas reconcile their inconsistent data completely in one bi-directional session or in two uni-directional sessions. In other words, a replica gets all missing updates from another replica in one session. Complete anti-entropy was designed to support full replication, but it may fail in partial replication. For example, in Figure 3.1, assume that R_2 and R_3 have not received any update yet, and R_6 has accepted three updates u_6^1, u_6^2, and u_6^3 from clients. The scopes and accept-ids for u_6^1, u_6^2, and u_6^3 are shown in Table 3.1. Note that u_6^3 belongs to two scopes, S_1 and S_2. Consider three sessions among R_2, R_3, and R_6 as follows.

- In the session between R_2 and R_6, R_2 gets new updates u_6^1 and u_6^3 in scope S_1 from R_6, and R_2's summary vector changes from () to $((R_6, t_6^3))$.

- In the session between R_3 and R_2, R_3 gets new updates u_6^1 and u_6^3 in scope S_1 from R_2, and R_3's summary vector changes from () to $((R_6, t_6^3))$. Here, the summary for R_6 is wrong since R_3 has not received update u_6^2 which has an accept-id of (R_6, t_6^2).

Update	Scope	Accept-id
u_6^1	S_1	(R_6, t_6^1)
u_6^2	S_2	(R_6, t_6^2)
u_6^3	S_1, S_2	(R_6, t_6^3)

Table 3.1: The scopes and accept-ids for three updates, u_6^1, u_6^2, and u_6^3, at replica R_6 that are accepted by R_6 from clients, and $t_6^1 < t_6^2 < t_6^3$.

- In the session between R_3 and R_6, R_3 wants to get new updates in scope S_1 and S_2 from R_6, but since R_3 and R_6 have the same summary vector of $((R_6, t_6^3))$, R_6 will not send u_6^2 to R_3. Anti-entropy fails here because R_3 has an incorrect summary for R_6.

The reason for the wrong summary is that updates in different scopes are propagated separately, which may not follow the order of their accept-ids. For example, R_3 receives u_6^3 before u_6^2 (which is not in the order of their accept-ids) since u_6^3 is in scope S_1, u_6^2 is in scope S_2, and R_3 receives all updates in scope S_1 first.

To handle scope-based update propagation, it seems straightforward to extend the summary vector mechanism by using a summary matrix to maintain the largest accept-timestamp for each accept-replica-id and scope combination. However, this simple extension can work properly only if the scopes do not overlap, i.e., any update only belongs to a single scope. As we discussed earlier in Section 3.1, unfortunately, an update can belong to multiple scopes. For example, a service may be used by both internal and external users, and consequently it belongs to both internal and external services scopes. Similarly, in Table 3.1, update u_6^3 belongs to both scope S_1 and scope S_2. For such cases, using a summary matrix does not solve the problem.

To avoid the above summary problem (i.e., a replica cannot summarize its received updates using a summary vector or summary matrix) in applying the anti-entropy mechanism to partial replication, we propose to use selective anti-entropy.

3.3 Design

Selective anti-entropy allows two replicas to selectively reconcile inconsistent data in a session. In other words, a replica can choose to reconcile inconsistent data in any number of subsets. When all subsets are chosen, selective anti-entropy is equivalent to complete anti-entropy. Thus, selective anti-entropy is a generalization of complete anti-entropy with added flexibility.

In $\mathcal{P}(r, s)$, all updates can be classified into r categories based on their accept-replicas: $\overline{R_1}$, $\overline{R_2}$, ..., and $\overline{R_r}$, where $\overline{R_i}$ represents the subset of updates accepted by R_i $(1 \leq i \leq r)$. We use $\Theta(R_x, \{\overline{R_{x_1}}, \overline{R_{x_2}}, ..., \overline{R_{x_k}}\}, R_z)$ to denote a selective anti-entropy session, where R_x requests new updates in the following k subsets from R_z: $\overline{R_{x_1}}$, $\overline{R_{x_2}}$, ..., and $\overline{R_{x_k}}$.

We assume that a replica R_x does not request updates accepted by itself (i.e., $\overline{R_x}$) from another replica R_y. But R_x may request updates accepted by R_y (i.e., $\overline{R_y}$) directly from R_y or indirectly from yet another replica R_z. For the former case, the session $\Theta(R_x, \{\overline{R_y}\}, R_y)$ is referred to as a *direct session*. For the latter case, the session $\Theta(R_x, \{\overline{R_y}\}, R_z)$ is referred to as an *indirect session*. If R_x requests updates accepted by R_y and R_z from R_z, then the session $\Theta(R_x, \{\overline{R_y}, \overline{R_z}\}, R_z)$ is referred to as a *mixed session*.

Based on the number of subsets requested and whether the session is direct or indirect, selective anti-entropy sessions can be classified into four types: select-one-direct, select-one-indirect, select-multiple, and select-all. Their corresponding definitions are shown in Table 3.2. A select-multiple or select-all session can be viewed as comprising k $(k \geq 2)$ select-one sessions, in which at most one is select-one-direct session, and the rest are select-one-indirect sessions. A select-all session is equivalent to a complete anti-entropy session.

Type	Number of Subsets Requested	Direct/Indirect Session
select-one-direct	1	direct
select-one-indirect	1	indirect
select-multiple	$[2, r-1)$	indirect/mixed
select-all	$r-1$	mixed

Table 3.2: Four types of selective anti-entropy sessions: select-one-direct, select-one-indirect, select-multiple, and select-all, where the total number of replicas is r.

Figure 3.2 gives an example to show how selective anti-entropy differs from complete anti-entropy. In this example, we have three replicas, R_1, R_2, and R_3, all serving the same scopes. Figure 3.2(a) shows the initial state at each replica: R_1 has no update yet; R_2 has received two updates u_2^1 and u_2^2 from clients, and has propagated u_2^1 to R_3; and R_3 has received two updates u_3^1 and u_3^2 from clients, and has propagated u_3^1 to R_2. Figure 3.2(b) illustrates how R_1 performs complete anti-entropy with R_2 and R_3 sequentially: it first gets three updates (u_2^1, u_2^2, and u_3^1) from R_2, and then it gets one update (u_3^2) from R_3. In contrast, Figure 3.2(c) illustrates how R_1 performs select-one-direct anti-entropy with R_2 and R_3 simultaneously: it gets two updates (u_2^1 and u_2^2) from R_2 and gets two updates (u_3^1 and u_3^2) from R_3 in parallel.

3.3.1 Safe Sessions

An anti-entropy session is *safe* if it will not cause any incorrect summary at the replica initiating the session. To identify which selective anti-entropy session is safe, we present three lemmas below. The basic idea for ensuring a selective anti-entropy session safe is that a replica needs to get all scopes of needed updates in a subset at once; getting partial scopes of needed updates in a subset may cause an incorrect summary of the subset at the replica initiating the session. We use $S(R_x)$, $S(R_y)$, and $S(R_z)$ to denote the serving scopes of replica R_x, R_y, and R_z, respectively.

R_1

$\emptyset$

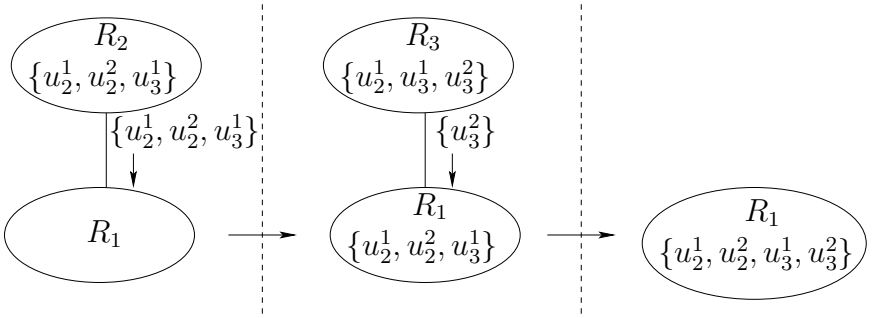
R_2

Update	Accept-id
u_2^1	(R_2, t_2^1)
u_2^2	(R_2, t_2^2)
u_3^1	(R_3, t_3^1)

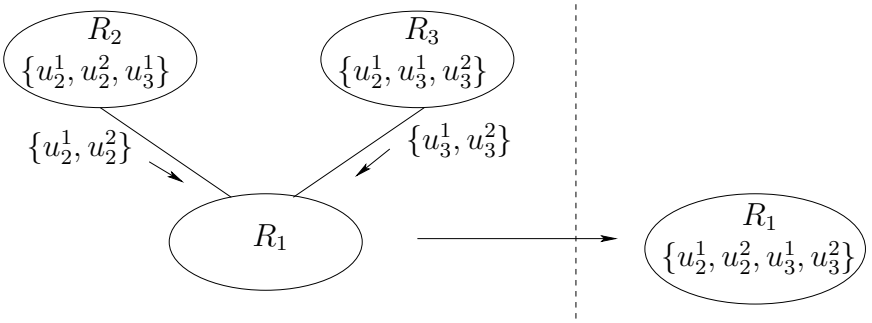
R_3

Update	Accept-id
u_2^1	(R_2, t_2^1)
u_3^1	(R_3, t_3^1)
u_3^2	(R_3, t_3^2)

(a) The initial state at each replica: R_1 has no update yet; R_2 has received two updates u_2^1 and u_2^2 from clients, and has propagated u_2^1 to R_3; and R_3 has received two updates u_3^1 and u_3^2 from clients, and has propagated u_3^1 to R_2, where $t_2^1 < t_2^2$ and $t_3^1 < t_3^2$



(b) R_1 performs complete anti-entropy with R_2 and R_3 sequentially: it first gets three updates (u_2^1, u_2^2, and u_3^1) from R_2, and then it gets one update (u_3^2) from R_3.



(c) R_1 performs select-one-direct anti-entropy with R_2 and R_3 simultaneously: it gets two updates (u_2^1 and u_2^2) from R_2 and gets two updates (u_3^1 and u_3^2) from R_3 in parallel.

Figure 3.2: An example to show how selective anti-entropy (here we use select-one-direct anti-entropy) differs from complete anti-entropy

Lemma 3.1. *Any select-one-direct anti-entropy session, such as* $\Theta(R_x, \{\overline{R_z}\}, R_z)$*, is safe.*

Proof. In a select-one-direct anti-entropy session such as $\Theta(R_x, \{\overline{R_z}\}, R_z)$, R_x requests one subset of updates from R_z, and R_z is the accept-replica for the requested subset $\overline{R_z}$. Note that R_x only requests updates in $S(R_x) \cap S(R_z)$ instead of $S(R_z)$. $S(R_x) \cap S(R_z)$ and $S(R_z)$ can have three different relationships: (1) $S(R_x) \cap S(R_z) \subset S(R_z)$ if R_x only serves partial scopes of R_z; (2) $S(R_x) \cap S(R_z) = S(R_z)$ if R_x serves all scopes of R_z; and (3) $S(R_x) \cap S(R_z) = \emptyset$ if R_x does not serve any scope of R_z. In all cases, we have $S(R_x) \cap S(R_z) \subseteq S(R_z)$, where $S(R_x) \cap S(R_z)$ is the scopes of updates requested by R_x and $S(R_z)$ is the scopes of updates available at R_z. Since the requested scopes are a subset or equal to the available scopes, R_x can get all scopes of needed updates in $\overline{R_z}$ from R_z at once. Note that R_x will have a summary of $\overline{R_z}$ that is correct only for $S(R_x) \cap S(R_z)$. □

Lemma 3.2. *A select-one-indirect anti-entropy session* $\Theta(R_x, \{\overline{R_y}\}, R_z)$ *is safe if* $S(R_x) \cap S(R_y) \subseteq S(R_z)$*. In other words,* $\Theta(R_x, \{\overline{R_y}\}, R_z)$ *may cause an incorrect summary of* $\overline{R_y}$ *at* R_x *if* $S(R_x) \cap S(R_y) \supset S(R_z)$*.*

Proof. In a select-one-indirect anti-entropy session such as $\Theta(R_x, \{\overline{R_y}\}, R_z)$, R_x requests one subset of updates from R_z, but R_z is not the accept-replica for the requested subset $\overline{R_y}$. Based on Lemma 3.1, we can assume that R_z has a summary of $\overline{R_y}$ that is correct for $S(R_z) \cap S(R_y)$. If $S(R_x) \cap S(R_y) \subseteq S(R_z)$, then we have $S(R_x) \cap S(R_y) \subseteq S(R_z) \cap S(R_y)$, where $S(R_x) \cap S(R_y)$ is the scopes of updates requested by R_x and $S(R_z) \cap S(R_y)$ is the scopes of updates available at R_z. Since the requested scopes are a subset or equal to the available scopes, R_x can get all scopes of needed updates in $\overline{R_y}$ from R_z at once. Note that R_x will have a summary of $\overline{R_y}$ that is correct only for $S(R_x) \cap S(R_y)$. □

We show some examples for Lemma 3.2. In Figure 3.1, $\Theta(R_3, \{\overline{R_1}\}, R_6)$ is safe because $S(R_3) \cap S(R_1) = \{S_1, S_2\} = S(R_6)$, but $\Theta(R_3, \{\overline{R_6}\}, R_2)$ may not be safe since $S(R_3) \cap S(R_6) = \{S_1, S_2\} \supset S(R_2) = \{S_1\}$.

Lemma 3.3. *A select-multiple or select-all anti-entropy session is safe if each of its select-one-indirect sessions is safe.*

Proof. Without loss of generality, consider a select-two session $\Theta(R_x, \{\overline{R_y}, \overline{R_z}\}, R_z)$, where R_x requests two subsets of updates, $\overline{R_y}$ and $\overline{R_z}$, from R_z. Since $\overline{R_y}$ and $\overline{R_z}$ are disjoint for the purpose of update propagation, this select-two session can be converted into two select-one sessions, namely a select-one-direct session $\Theta(R_x, \{\overline{R_z}\}, R_z)$ and a select-one-indirect session $\Theta(R_x, \{\overline{R_y}\}, R_z)$. Each select-one session only affects one summary at R_x. If all select-one sessions are safe, then the summary vector at R_x will be connect. Based on Lemma 3.1, any select-one-direct session is safe. Thus, a select-multiple or select-all anti-entropy session is safe if each of its select-one-indirect sessions is safe. □

Based on Lemma 3.1, 3.2, and 3.3, we have Theorem 3.4 on safe sessions for selective anti-entropy.

Theorem 3.4. *Using selective anti-entropy, a replica can avoid using unsafe sessions and always use safe sessions.*

Proof. Based on Lemma 3.2 and 3.3, a replica can avoid using unsafe select-one-indirect, unsafe select-multiple, and unsafe select-all sessions.

Based on Lemma 3.1, a replica can always use safe select-one-direct sessions, assuming that a replica does not fail permanently. Moreover, a replica can use safe select-one-indirect, safe select-multiple, and safe select-all sessions. □

Based on Theorem 3.4, the main advantage of using selective anti-entropy is that it enables a replica to avoid using unsafe sessions and always use safe sessions. In contrast, a replica cannot guarantee a complete anti-entropy session to be safe in some cases. For example, consider two complete anti-entropy sessions, Θ_1 and Θ_2, in Figure 3.1: Θ_1 is between R_3 and R_6, Θ_2 is between R_3 and R_2, and Θ_1 is performed before Θ_2. R_3 cannot guarantee Θ_2 to be safe since between Θ_1 and Θ_2, R_2 may perform another session with R_6 and get new updates in $\overline{R_6}$.

3.3.2 Parallel Sessions

While a replica needs to perform complete anti-entropy sessions with other replicas sequentially, it can perform some selective anti-entropy sessions in parallel. For example, a replica can perform all select-one-direct sessions in parallel with other replicas. In general, we have Theorem 3.5 on parallel sessions for selective anti-entropy.

Theorem 3.5. *Replica R_x can perform k ($k \geq 2$) selective anti-entropy sessions $(\Theta_1, \Theta_2, ..., \Theta_k)$ in parallel with other replicas if and only if the requested subsets of updates $(U_1, U_2, ..., U_k)$ in these sessions do not overlap, i.e., $U_1 \cap U_2 \cap ... \cap U_k = \emptyset$.*

Proof. In a replication system with r replicas, updates are classified into r subsets: $\overline{R_1}$, $\overline{R_2}$, ..., and $\overline{R_r}$, where $\overline{R_i}$ represents the subset of updates accepted by replica R_i ($1 \leq i \leq r$). For the purpose of update propagation, these r subsets of updates are disjoint. We assume that replica R_x doe not need to get updates in $\overline{R_x}$ from another replicas. Thus, without getting duplicates of the same updates, R_x can requests updates in up to $r-1$ subsets at once, which may be done via a single select-multiple or select-all session, or via multiple parallel sessions where each subset is requested at most once. □

We show an example for Theorem 3.5. Assume $k = 2$, $U_1 = \{\overline{R_2}, \overline{R_4}\}$, and $U_2 = \{\overline{R_3}, \overline{R_5}\}$, then session Θ_1 and Θ_2 can be performed in parallel because $U_1 \cap U_2 = \emptyset$.

Running anti-entropy sessions in parallel at a replica can improve performance since the replica does not need to wait until a session is finished before it can start another session.

3.4 Implementation

We have implemented selective anti-entropy in SLP mesh enhancement (mSLP) [156, 145]. For full replication, complete anti-entropy is sufficient, but for partial replication, selective anti-entropy is needed. Thus, mSLP supports both selective and complete anti-entropy sessions. In mSLP, the anti-entropy request message carries an anti-entropy type ID to indicate the session type. There are two session control functions specific to selective anti-entropy, namely choosing safe sessions and performing multiple sessions in parallel. A simple strategy is to use select-one-direct sessions as much as possible for selective anti-entropy because all select-one-direct sessions are safe, and they can be performed in parallel.

3.5 Evaluation

While the main goal of selective anti-entropy is to solve the summary problem in applying the anti-entropy mechanism to partial replication, we give a brief evaluation of its performance in this section. We consider a replication system with r replicas, and we measure the time that a replica needs to finish all $r - 1$ anti-entropy sessions with the rest replicas. For simplicity, we assume that network conditions (bandwidth, delay, and loss rate) are roughly equivalent among all replicas, which is typical in a local

area network environment. In general, if only sequential sessions are used, selective anti-entropy and complete anti-entropy should deliver equivalent performance. Thus, we focus on selective anti-entropy with parallel sessions. Specifically, we compare the performance of parallel select-one-direct sessions with that of sequential complete sessions.

3.5.1 Performance Analysis

Using parallel select-one-direct sessions has the potential to deliver better performance than using sequential complete sessions for the following reasons. First, the setup time of these parallel sessions can overlap. Secondly, to process a select-one-direct anti-entropy request, a replica only needs to send new updates accepted by itself, which is much simpler than sending new updates accepted by $r - 1$ replicas. Remember that all new updates need to be propagated in increasing order of their accept-ids. Finally, if a replica has sufficient network bandwidth and processing power, it can receive updates at a higher rate by using parallel sessions.

3.5.2 Experimental Results

We used our mSLP Java implementation [145], and carried out experiments on a cluster of Sun Ultra Sparc (Ultra-1, Ultra-2, or Ultra-10) workstations. All machines ran Solaris 5.7, and were connected via 10 Mb/s Ethernet. We used a varying number of n machines as existing replicas, one machine as the new replica, and one machine as the client. The initial state was as follows. The client had sent 5005 different updates to each of the n replicas; and each replica had propagated 5000 updates accepted by itself to the rest replicas. Thus, each replica had $5000n + 5$ updates. When the new replica joined the replication system, it got updates from the existing n replicas.

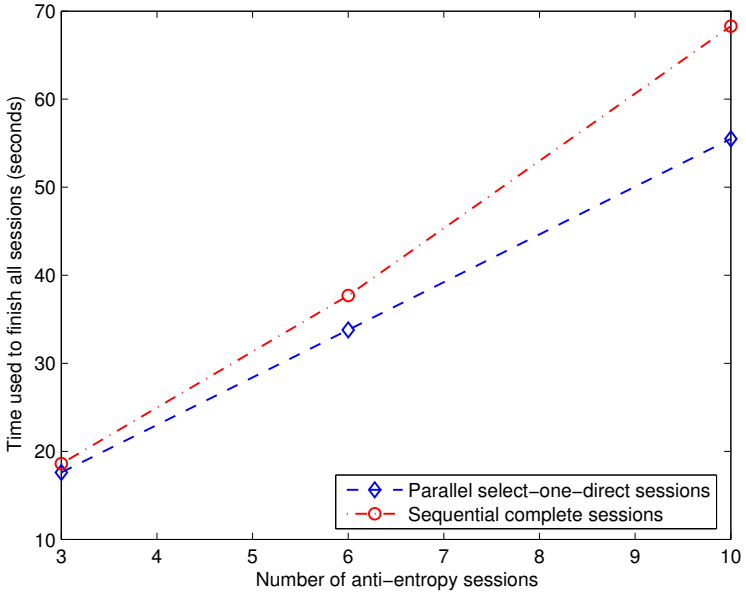


Figure 3.3: The performance of parallel select-one-direct sessions compared to that of sequential complete sessions

Figure 3.3 shows the time that the new replica took to finish anti-entropy with the existing n replicas. We observe that using parallel select-one-direct sessions always performs better than using sequential complete sessions. Also, the difference of the time used in the above two cases becomes larger as n increases.

3.6 Related Work

Using anti-entropy for high availability replication was first described in [44]. The time-stamped anti-entropy (TSAE) [54] proposed the summary vector technique, and suggested using multicast to propagate updates quickly. Bayou [96] enhanced anti-entropy flexibility by using uni-directional pair-wise sessions. Our work on selective anti-entropy [147] generalizes the anti-entropy mechanism, making it applicable to generic partial replication as well. We use selective anti-entropy in designing the SLP

mesh enhancement (mSLP) [156], an automated partial replication system for scope-based directory services. A recent replication system that employs anti-entropy is UDDI [129], a registry framework for universal description, discovery, and integration of web services, but UDDI only supports full replication.

3.7 Summary

This chapter described selective anti-entropy, a generic mechanism for high availability partial replication. By using safe sessions, selective anti-entropy can avoid the summary problem in applying the anti-entropy mechanism to partial replication. By using parallel sessions, selective anti-entropy can improve performance. As a generalization of complete anti-entropy, selective anti-entropy is flexible and applicable to both full replication and partial replication.

Chapter 4

DotSlash: An Automated Web Hotspot Rescue System

This chapter describes the DotSlash framework, an automated web hotspot rescue system that enables a web site to build an adaptive distributed web server system on the fly [150, 149]. We focus on load migration for static content in this chapter, and will address load migration for dynamic content [151, 154] in the next two chapters. In this chapter, we first introduce the web hotspot problem and discuss related work. Then, we provide an overview of DotSlash, including its usage models and some rescue examples. After presenting the DotSlash design, implementation, and evaluation, we give a summary.

4.1 Introduction

As more web sites are experiencing a request load that can no longer be handled by a single server, using multiple servers to serve a single site has become a widespread approach. Traditionally, a distributed web server system has used a fixed number of

dedicated servers based on capacity planning, which works well if the request load is relatively consistent and matches the planned capacity. However, web requests could be very bursty. A well-identified problem web hotspots, also known as flash crowds or the Slashdot effect [4], may trigger a large load increase but only last for a short period [65, 115]. For such situations, over-provisioning a web site is not only uneconomical but also difficult since the peak load is hard to predict [74].

To handle web hotspots effectively, we advocate dynamic allocation of server capacity from a server pool distributed globally. It is important to distribute load across wide area networks because the access link of a local network could become a bottleneck. As an example of global server pools, content delivery networks (CDNs) [131, 5, 45] have been used by large web sites, but small web sites often cannot afford the cost particularly since they may need these services very rarely. We seek a more cost-effective mechanism. As different web sites, e.g., different types or in different geographic locations, are less likely to experience their peak request loads at the same time, they could form a mutual-aid community, and use spare capacity in the community to relieve web hotspots experienced by any individual site [38]. Based on this observation, we designed *DotSlash* which allows a web site to build an adaptive distributed web server system on the fly to expand its capacity by utilizing spare capacity at other sites. Using DotSlash, a web site not only has a fixed set of *origin servers*, but also has a changing set of *rescue servers* drafted from other sites. A web server allocates and releases rescue servers based on its load conditions. The rescue process is completely self-managing and transparent to clients.

DotSlash does not aim to support a request load that is persistently higher than a web site's planned capacity, but rather to complement the existing web server infrastructure to handle short-term load spikes effectively. We envision a spectrum of mechanisms for web sites to handle load spikes. Infrastructure-based approaches

should handle the request load sufficiently in most cases (e.g., 99.9% of the time), but they might be too expensive for short-term enormous load spikes and insufficient for unexpected load increases. For these cases, DotSlash intervenes so that a web site can support its request load in more cases (e.g., 99.999% of the time). In parallel, a web site can use service degradation [1] such as turning off dynamic content and serving a trimmed version of static content under heavily-loaded conditions. As the last resort, a web site can use admission control [140] to reject a fraction of requests and only admit preferred clients.

DotSlash has the following advantages. First, it is self-configuring in that service discovery [59] is used to allow servers of different web sites to learn about each other dynamically, rescue actions are triggered automatically based on load conditions, and a rescue server can serve the content of its origin servers on the fly without the need of any advance configuration. Second, it is scalable because a web server can expand its capacity as needed by using more rescue servers. Third, it is very cost-effective since it utilizes spare capacity across a web server community to benefit any participating server, and it is built on top of the existing web server infrastructure, without incurring any additional hardware cost. Fourth, it is easy to use because standard DNS mechanisms and HTTP redirect are used to offload client requests from an origin server to its rescue servers, without the need of changing operating system or DNS server software. An add-on module to the web server software is sufficient to support all needed functions. Fifth, it is transparent to clients since it only uses server-side mechanisms. Client browsers remain unchanged, and client bookmarks continue to work. Finally, an origin server has full control of its own rescue procedure, such as how to choose rescue servers and when to offload client requests to rescue servers.

DotSlash targets small web sites, although large web site can also benefit from it.

Parts of this work may be applicable to other services such as Grid computational services [50] and VoIP services [108].

4.2 Related Work

Caching. Caching [135, 42, 34] provides many benefits for web content retrieval and distribution, such as reducing bandwidth consumption and client-perceived latency. Caching may appear at several different places, such as client-side proxy caching, intermediate network caching, and server-side reverse caching, many of which are not controlled by origin web servers. DotSlash uses caching at rescue servers to relieve load spikes at an origin server, where caching is set up on demand and fully controlled by the origin server.

Content delivery networks (CDNs). Commercial CDN services [131, 5, 45] deliver part or all of the content for a web site to improve the performance of content delivery. As an infrastructure-based approach, CDN services are good for reinforcing a web site in a long run, but less efficient for handling short-term load spikes. Also, using CDN services needs advance configuration such as contracting with a CDN provider and changing the URIs of offloading objects (e.g., Akamaizing [5]). As an alternative mechanism to CDN services, DotSlash offers cost-effective and automated rescue services for better handling short-term load spikes. Recently, free CDNs services such as CoralCDN [52] and CoDeeN [137] have been developed. CoralCDN [52] allows volunteer sites to form a peer-to-peer (P2P) CDN, which is transparent to clients. However, CoralCDN requires that web site publishers change selected URLs to Coralized URLs or Coral-aware users manually construct Coralized URLs because only Coralized URLs can use CoralCDN. CoDeeN [137] uses a network of open web proxy servers on PlanetLab to distribute and cache client requests, which can be ben-

eficial to both clients and web site publishers. However, CoDeeN is not transparent to clients because clients need to specify a CoDeeN proxy in their browser settings.

Distributed web servers. Using pre-arranged distributed web server systems is a widespread approach to support high request loads and reduce client-perceived delays. These systems often use replicated web servers (e.g., ScalaServer [13] and GeoWeb [29]), with a focus on load balancing and serving a client request from the closest server. In contrast, DotSlash allows an origin server to build a distributed system of heterogeneous rescue servers on demand so as to relieve the heavily-loaded origin server.

Server-side collaboration. DotSlash is a system that facilitates dynamic collaboration among different web sites. By using service discovery technology, DotSlash enables an origin server to collaborate with a dynamically changing set of rescue servers for handling unexpected load spikes. The distributed cooperative Apache web server system [77] and the proactive hot spot avoidance system [48] also support collaboration among different web servers. However, they do not address the issue of how to form a collaborative server group dynamically, which limits their scalability and adaptivity to changing environments. Backslash [121] uses P2P overlay networks to form a collaborative web mirroring system and uses distributed hash tables (DHTs) to locate resources. However, Backslash relies on the uniformity of URL hashing for load distribution, which may not achieve good load balancing due to skewed demand for web objects and diverse capacity of surrogate nodes. In contrast, DotSlash allows an origin server to fully control its load distribution based on the available capacity of each rescue server.

Content internetworking. The Internet Engineering Task Force (IETF) has developed a model for content (distribution) internetworking (CDI) [43, 113]. The DotSlash architecture appears to be a special case of the CDI architecture, where each

web server itself is a content network. However, the CDI framework does not address how to dynamically allocate servers and adjust redirect rate based on feedback to handle short-term load spikes, which is the main focus of DotSlash.

Client-side cooperation. Client-side mechanisms allow clients to help each other so as to alleviate server-side congestion and reduce client-perceived delays. An origin web server can mediate client cooperation by redirecting a client to another client that has recently downloaded the URI, e.g., Pseudoserving [70] and CoopNet [92]. Clients can also form P2P overlay networks and use search mechanisms to locate resources, e.g., PROOFS [122] and BitTorrent [23]. Client-side P2P overlay networks have advantages in sharing large and popular files, which can reduce request load at origin web servers. In general, client-side mechanisms scale well as the number of clients increases, but they are not transparent to clients, which is likely to prevent widespread deployment.

Grid. Grid technologies allow "coordinated resource sharing and problem solving in dynamic, multi-institutional organizations" [50], with a focus on large-scale computational problems and complex applications. The sharing in Grid is broader than simply file exchange; it can involve direct access to computers, software, data, and other resources. In contrast, DotSlash employs inter-web-site collaborations to handle web hotspots effectively, with an emphasis on overload control at web servers and disseminating popular files to a large number of clients.

Océano. Océano [10] allows multiple customers to be hosted on a shared computing utility. Although Océano and DotSlash both support dynamic allocation and de-allocation of servers based on load conditions, Océano targets a server farm managed by the same administrator, whereas DotSlash targets different web sites within a mutual-aid community. Consequently, Océano uses a centralized control for server allocation, whereas DotSlash uses a distributed negotiation for rescue server allocation.

In terms of server sharing, Océano employs sequential sharing at the granularity of whole servers, whereas DotSlash allows a rescue server to serve its own content and the origin server's content in parallel. Moreover, Océano dynamically changes a server's computing environment, such as operating system and application software, so as to support different customers. In contrast, DotSlash employs service discovery to allow an origin server to find suitable rescue servers without the need to change the computing environment of any server.

4.3 DotSlash Overview

In DotSlash, a web server is in one of the following states at any time: in *SOS state* it gets rescue services from others, in *rescue state* it provides rescue services to others, and in *normal state* otherwise. These three states are mutually exclusive: a server is not allowed to get rescue services and provide rescue services at the same time. Using this rule can avoid complex rescue scenarios (e.g., a rescue loop where S_1 requests a rescue service from S_2, S_2 requests a rescue service from S_3, and S_3 requests a rescue service from S_1), and keeps DotSlash simple and robust without compromising scalability.

Throughout this thesis, we use the notation origin server and rescue server in the following way. When two servers set up a rescue relationship, the one that benefits from rescue services is the *origin server*, and the one that provides rescue services is the *rescue server*. Figure 4.1 shows an example of rescue relationships for eight web servers, where an arrow from S_y to S_x denotes that S_y provides rescue services to S_x. In this figure, S_1 and S_2 are origin servers; S_3, S_4, S_5, and S_6 are rescue servers; and S_7 and S_8 are not involved with rescue services yet.

Next, we describe DotSlash usage models and gives some rescue examples.

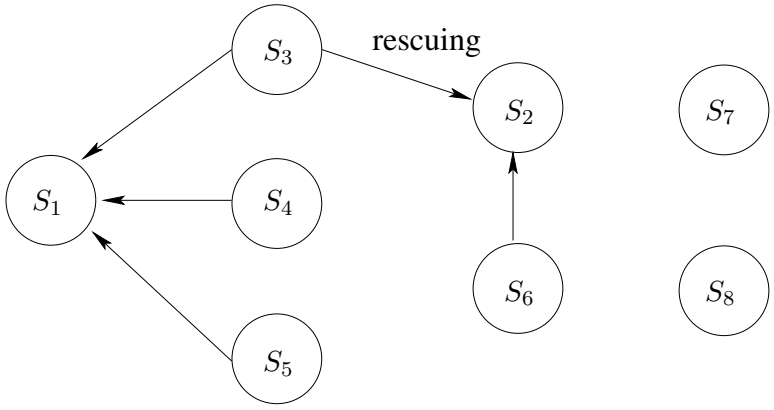


Figure 4.1: An example of DotSlash rescue relationships for eight web servers S_1 to S_8, where an arrow from S_y to S_x denotes that S_y provides rescue services to S_x. In this figure, S_1 and S_2 are origin servers; S_3, S_4, S_5, and S_6 are rescue servers; and S_7 and S_8 are not involved with rescue services yet.

4.3.1 Usage Models

DotSlash allows different web sites to form a mutual-aid community, and use spare capacity in the community to relieve web hotspots experienced by any individual site. We consider three types of mutual-aid communities, namely open communities, closed communities, and flood-insurance closed communities.

An open mutual-aid community needs to deploy at least one DotSlash service registry accessible to public. For reliability, multiple registries need to be deployed, which replicate service registration information from each other automatically (see Section 4.4.5 for details). It is beneficial to set up a DNS domain for a mutual-aid community, which allows DotSlash service registries to be discovered via DNS SRV, eliminating manual configuration for registry discovery (see Section 4.4.5 for details). A web server joins an open mutual-aid community by registering itself with any DotSlash service registry in the community, and contributing its spare capacity to the community. Under heavy load, a participating server discovers and uses spare capacities at other servers in the community via DotSlash rescue services. An open

mutual-aid community is intended for a cooperative environment; it is simple, but it does not provide security measures against attacks or abuse.

Since DotSlash uses the Service Location Protocol (SLP) [59] for service discovery, we can employ the SLP authentication features to eliminate malicious service registrations. Thus, only authorized web sites can register with DotSlash service registries, and each registration can be verified. As SLP authentication is based on public key cryptography, a closed mutual-aid community needs to have an authority for key management and distribution. When a web site is authorized to join a closed community, it needs to create a pair of public/private keys, and register its public key with the community authority. As a result, the community authority maintains a list of public keys for all authorized participating servers. When a web server performs a service registration, it uses its private key to sign the registration message. When a DotSlash service registry receives a registration, it checks whether the public key specified in the SPI (Security Parameters Index) of the registration is a valid key in the community, and uses the key, if valid, to verify the registration. Similarly, when a web server receives a list of matching URLs for its rescue server discovery, it checks whether the public key specified in an SPI is a valid key in the community, and uses the key, if valid, to verify the corresponding URL entry. By doing so, only authorized web sites can serve as rescue servers. The same mechanism can be used for origin server authentication as follows. When an origin server initiates a rescue relationship with a rescue server, it signs its *SOS* request using its private key (see Section 4.4.4 for details). The rescue server accepts an *SOS* request only if it is from a verified member of the community.

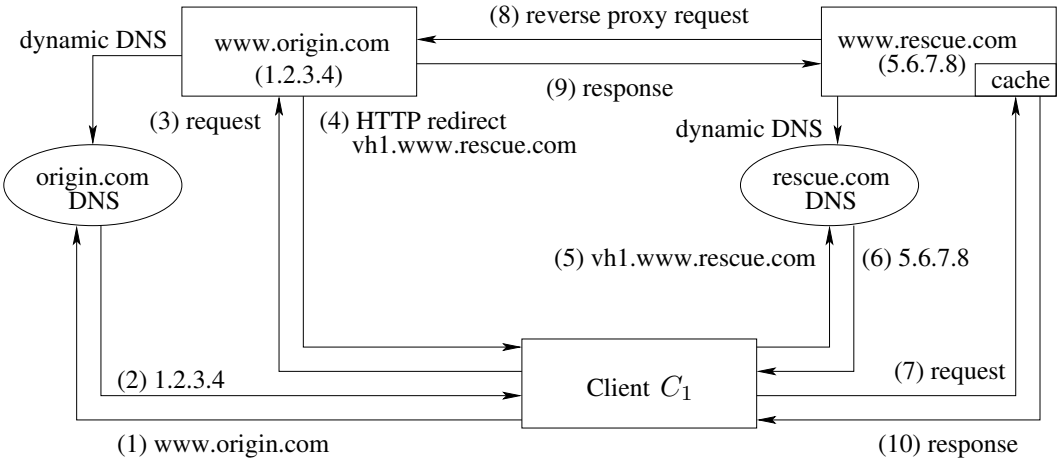
To increase the incentive for providing DotSlash rescue services and reduce abuse, a flood-insurance closed mutual-aid community can be used. In such a community, each participating web server needs to have its own public key and private key, and

it needs to pay the community authority a small premium to obtain tokens for using rescue services. For example, $3 might be needed for 10 tokens. The community authority maintains a list of all valid tokens in the community, and associates each token with a participating web server. Each time when an origin server sets up a rescue relationship with a rescue server, it needs to transfer one token to the rescue server, which is performed via a *TOKEN* message that carries the token signed by the origin server using its private key. When a rescue server receives a *TOKEN* message, it signs the token using its private key, and then sends this double-signed message to the community authority for verification. When the community authority receives a *TOKEN* message, if it can verify the token, it changes the token's association from the origin server to the rescue server, and returns "200 OK"; otherwise, it returns "406 Token_Invalid". When an origin server has used up all its tokens, it needs to buy new tokens for using rescue services. On the other hand, a rescue server can accumulate tokens for its own rescue needs or sell its tokens.

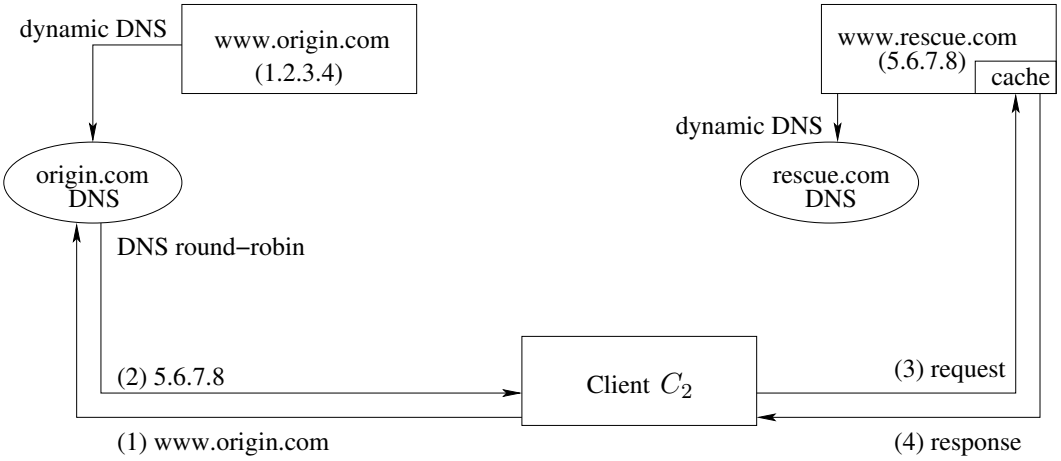
4.3.2 Rescue Examples

In DotSlash, an origin server uses HTTP redirect and DNS round robin to offload client requests to its rescue servers, and a rescue server serves as a reverse caching proxy for its origin servers (see Section 4.4.1 for details). There are four rescue cases: (1) HTTP redirect at the origin server and cache miss at the rescue server, (2) HTTP redirect at the origin server and cache hit at the rescue server, (3) DNS round robin at the origin server and cache miss at the rescue serever, and (4) DNS round robin at the origin server and cache hit at the rescue server. We show examples for case 1 and 4 next; case 2 and 3 can be derived similarly.

In Figure 4.2, the origin server S_o is *www.origin.com* with IP address *1.2.3.4*, and



(a) A rescue example for using HTTP redirect at the origin server and having a cache miss at the rescue server, where client C_1 follows a ten-step procedure to retrieve *http://www.origin.com/index.html*.



(b) A rescue example for using DNS round robin at the origin server and having a cache hit at the rescue server, where client C_2 follows a four-step procedure to retrieve *http://www.origin.com/index.html*.

Figure 4.2: Two examples for DotSlash rescue services

the rescue server S_r is *www.rescue.com* with IP address *5.6.7.8*. We assume that S_o and S_r has set up a rescue relationship, S_r has assigned an alias *vh1.www.rescue.com* to S_o, and S_o has added S_r's IP address to its round robin local DNS. Figure 4.2(a) gives an example for case 1, where client C_1 follows a ten-step procedure to retrieve *http://www.origin.com/index.html*.

1. C_1 resolves S_o's domain name *www.origin.com*;
2. C_1 gets S_o's IP address *1.2.3.4*;
3. C_1 makes an HTTP request to S_o using *http://www.origin.com/index.html*;
4. C_1 gets an HTTP redirect from S_o to *http://vh1.www.rescue.com/index.html*;
5. C_1 resolves S_r's alias *vh1.www.rescue.com*;
6. C_1 gets S_r's IP address *5.6.7.8*;
7. C_1 makes an HTTP request to S_r using *http://vh1.www.rescue.com/index.html*;
8. S_r makes a reverse proxy request to S_o using *http://www.origin.com/index.html* because of a cache miss for *http://vh1.www.rescue.com/index.html*;
9. S_o sends the requested file to S_r;
10. S_r caches the requested file, and returns the file to C_1.

Figure 4.2(b) gives an example for case 4, where client C_2 follows a four-step procedure to retrieve *http://www.origin.com/index.html* using dynamic DNS.

1. C_2 resolves S_o's domain name *www.origin.com*;
2. C_2 gets S_r's IP address *5.6.7.8* due to DNS round robin at S_o's local DNS;

3. C_2 makes an HTTP request to S_r using *http://www.origin.com/index.html*;

4. C_2 gets the requested file from S_r because here is a cache hit.

4.4 DotSlash Design

The main focus of DotSlash is to allow a web site to build an adaptive distributed web server system in a fully automated way. DotSlash consists of dynamic virtual hosting, request redirection, workload monitoring, rescue control, and service discovery.

4.4.1 Dynamic Virtual Hosting

Dynamic virtual hosting allows a rescue server to serve the content of its origin servers on the fly. Existing virtual hosting (e.g., Apache [9]) needs advance configuration: registering virtual host names in DNS, creating *DocumentRoot* directories, and adding directives to the configuration file to map virtual host names to *DocumentRoot* directories. DotSlash handles all these configuration actions dynamically.

When a rescue relationship is set up between an origin server and a rescue server, the rescue server assigns a unique virtual host name to the origin server, which is used in the HTTP redirects issued from the origin server. A rescue server generates needed virtual host names dynamically by adding a sequence number component to its configured name, e.g., *vh<seqnum>.host.domain* for *host.domain*, where *<seqnum>* is monotonically increasing. Virtual host names are registered using A records via dynamic DNS updates [132]. We have set up a domain *dot-slash.net* that accepts virtual host name registrations. For example, *www.rescue.com* can obtain a unique host name *foo* in *dot-slash.net*, and register its virtual host names as *vh<seqnum>.foo.dot-slash.net*.

As a rescue server, *www.rescue.com* may receive client requests that use three different kinds of host names: its configured name *www.rescue.com*, an assigned virtual host name such as *vh1.www.rescue.com*, or an origin server name such as *www.origin.com*. Its own content is requested in the first case, whereas the content of its origin servers is requested in the last two cases. Moreover, the second case is due to HTTP redirects from the origin servers, and the third case is due to DNS round robin at the origin servers.

In order to map each assigned virtual host name to its corresponding origin server, a rescue server maintains a host name mapping table. When a rescue server receives a client request, it first checks whether the requested host name is its own configured name. If that is not the case, it lookups the requested host name in its host name mapping table. If the requested host name matches a mapping entry's virtual host name or origin server name, the origin server name is returned. Due to client-side caching, web clients may continue to request an origin server's content from its old rescue servers. To handle this situation properly, a rescue server does not remove a mapping entry immediately after the rescue service has been terminated, but rather keeps the mapping entry for a configured time such as 24 hours, and redirects such a request back to the corresponding origin server via an HTTP redirect.

A rescue server works as a reverse caching proxy for its origin servers. For example, when *www.rescue.com* has a cache miss for *http://vh1.www.rescue.com/index.html*, it maps *vh1.www.rescue.com* to *www.origin.com*, and issues a reverse proxy request for *http://www.origin.com/index.html*. Using reverse caching proxy offers a few advantages. First, as files are replicated on demand, the origin server incurs low cost since it does not need to maintain states for replicated files and can avoid transferring files that are not requested by rescue servers. Second, as proxy and caching are functions supported by most web server software, it is simple to use reverse proxying to get

needed files, and use the same caching mechanisms to cache proxied files and local files.

By using reverse caching proxy, DotSlash employs an on-demand replication policy, that is a document is transferred from an origin server to a rescue server only when the document is requested and is a cache miss at the rescue server. This policy can reduce the bandwidth requirement at the origin server as much as possible. Note that typically only a small set of documents are requested during web hotspots [65], thus there is no need to replicate the whole site from the origin server for the purpose of web hotspot rescue. In general, replication policies control how content is replicated from an origin server to its rescue servers, such as pushed by the origin server or pulled by its rescue servers. Previous work [98, 97] indicates that no single replication policy can efficiently manage all documents. The optimal replication policy not only depends on how documents are accessed, but also depends on how servers are organized, e.g., replication strategies for unstructured P2P networks have been studied in [39].

4.4.2 Request Redirection

Having described how a rescue server serves the content of its origin servers on the fly in the last section, we discuss how client requests are redirected from an origin server to its rescue servers in this section. Request redirection [17, 28, 136] involves two aspects: the mechanisms to route client requests from an origin server to its rescue servers and the policies to choose a rescue server among multiple choices, namely server selection [47, 30]. A client request can be redirected by the origin server's authoritative DNS, the origin server itself, or a redirector at the transport layer (content-blind) or application layer (content-aware). Redirection policies can

be based on load at rescue servers, locality of requested files at rescue servers, and proximity between the client and rescue servers.

We first provide a brief overview of existing request redirection mechanisms. DNS-based redirection routes client requests during the server name resolution phase. It is widely used [5, 45] because of its generality and simplicity: it is applicable to any IP-based applications and no changes are required to clients and servers. However, DNS-based redirection has certain limitations. For example, it requires reducing or eliminating client-side caching of domain name binding, which has negative effects on DNS performance [116]. Also, DNS-based redirection can only provide crude load balancing. Network routers can be extended, such as anycast [93, 64] and Cisco DistributedDirector [37], to distribute client requests transparently based on routing metrics, but this approach is less flexible in that it is hard to incorporate server load conditions. A cluster load balancer can dispatch client requests among servers within a cluster, but it may become a single point of failure and a bottleneck. ScalaServer [13] describes a scalable request distribution scheme for cluster-based network servers by using a centralized dispatcher as the front-end and replicating the distributor component onto each cluster node. Using mobility support in IPv6 [2] and the Internet Indirection Infrastructure [124] have also been proposed to route client requests. HTTP redirect [49] is easy to use, but clients incur longer delays.

DotSlash uses two mechanisms for request redirections: DNS round robin as the first level crude load distribution, and HTTP redirect as the second level fine-grained load balancing. DNS round robin can reduce the request arrival rate at the origin server, and HTTP redirect can increase the service rate of the origin server because an HTTP redirect is much cheaper to serve than the original content. Both mechanisms can increase the origin server's throughput for request handling.

We have investigated three options for constructing redirect URIs: IP address,

virtual directory, and virtual host name. Using the rescue server's IP address can save the client's DNS lookup time for the rescue server's name, but the rescue server is unable to tell whether a request is for itself or for one of its origin servers. Using a virtual directory such as */dotslash-vh*, *http://www.origin.com/index.html* can be redirected as *http://www.rescue.com/dotslash-vh/www.origin.com/index.html*. The problem is that it does not work for embedded relative URIs. DotSlash uses virtual host names in HTTP redirects, which allows proper virtual hosting at the rescue server, and works for embedded relative URIs.

In terms of redirection policies, DotSlash uses standard DNS round robin without modifying the DNS server software, and uses weighted round robin (WRR) for HTTP redirects, where the weight is the allowed rate of HTTP redirects assigned by each rescue server. The allowed rate of HTTP redirects is specified via two parameters, allowed redirect rate in requests per second and allowed redirect data rate in kilobytes per second, which are used to control the rate of load migration for dynamic content and static content, respectively. Due to factors such as caching and embedded relative URIs, the rate of HTTP redirects seen by the origin server may be different from that served by the rescue server. Thus, in DotSlash an origin server uses rate feedback from the rescue server to adjust its rate of HTTP redirects (see Section 4.4.4 for details).

DotSlash performs request redirection using DNS round robin and HTTP redirect. However, DNS round robin and HTTP redirect need to be avoided in certain cases. On one hand, a request sender needs to bypass DNS round robin at a web server by using the web server's IP address directly in the following cases: when an origin server initiates a rescue connection to a rescue server, when a rescue server makes a reverse proxy request to an origin server, and when a web client retrieves a web server's status information. On the other hand, a request receiver needs to avoid

Table 4.1: Major DotSlash parameters, where type C, O, I, and D denote configurable parameters, measured outputs, control inputs, and derived parameters, respectively, and 1 kB = 1000 bytes.

Parameter	Description	Type
ρ_n^l and ρ_n^u	lower and upper threshold for network utilization, default 50% and 75%	C
λ_d^m	maximum data rate (kB/s) for outbound HTTP traffic	C
τ	control interval, default 1 second	C
α	used in exponentially weighted moving average filter, default 0.5	C
λ_d	real data rate (kB/s) of outbound HTTP traffic	O
λ_{rd}	real redirect data rate (kB/s)	O
λ_{rd}^a	allowed redirect data rate (kB/s)	I
P_r	redirect probability	I
ρ_n	network utilization, $\rho_n = \lambda_d/\lambda_d^m$	D
$\hat{\rho_n}$	reference network utilization, $\hat{\rho_n} = (\rho_n^u + \rho_n^l)/2$	D
$\hat{\lambda_d}$	reference data rate (kB/s), $\hat{\lambda_d} = \hat{\rho_n}\lambda_d^m$	D
β	adjustment factor for control inputs, $\beta = \rho_n/\hat{\rho_n}$	D

performing an HTTP redirect if the request is from a rescue server, or if the request is for retrieving the server's status information.

4.4.3 Workload Monitoring

Having described how DotSlash performs request redirection in last section, we discuss workload monitoring in this section, which allows a web server to react quickly to load changes. Major DotSlash parameters are summarized in Table 4.1. We measure the utilization of each resource at a web server separately. We use ρ_n and ρ_c to denote network utilization and CPU utilization, respectively. According to a recent study [92], network bandwidth is the most constrained resource for most static web sites during hotspots. Thus, we focus on monitoring network utilization ρ_n in this section. We use two configurable parameters, lower threshold ρ_n^l and upper threshold ρ_n^u, to

define three regions for ρ_n: lightly loaded region $[0, \rho_n^l)$, desired load region $[\rho_n^l, \rho_n^u]$, and heavily loaded region $(\rho_n^u, 100\%]$. Furthermore, we define a reference utilization $\hat{\rho_n}$ as $(\rho_n^l + \rho_n^u)/2$.

In DotSlash, we monitor outbound HTTP traffic within a web server, without relying on an external module to monitor traffic on the link. We assume there is no significant other traffic besides HTTP at a web server, and assume a web server has a symmetric link or its inbound bandwidth is greater than its outbound bandwidth, which is true, for example, for a web server behind DSL. Since a web server's outbound data rate is normally greater than its inbound data rate, it should be sufficient to only monitor outbound HTTP traffic.

Due to header overhead (such as TCP and IP headers) and retransmissions, the HTTP traffic rate monitored by DotSlash is less than the real traffic rate on the link. Since the header overhead is relatively constant and other overheads are usually small, to simplify calculation, we use a configurable parameter λ_d^m to denote the maximum data rate for outbound HTTP traffic, where $\lambda_d^m = BU$, B is the network bandwidth, and U is the percentage of bandwidth that is usable for HTTP traffic. We perform a special accounting for HTTP redirects because they may account for a large percentage of HTTP responses and their header overhead is large compared to their small payload sizes. For an HTTP redirect response of n bytes, its accounting size is $A_r = (n + O)U$ bytes, where O is the header overhead in bytes. A web server sends five TCP packets for each HTTP redirect: one for accepting the client TCP connection, one for acknowledging the client HTTP request, one for sending the HTTP redirect to the client, and two for terminating the client TCP connection. The first TCP header (SYN ACK) is 40 bytes, and the remaining four TCP headers are 32 bytes each. Thus, $O = (40 + 32 * 4) + 20 * 5 + (14 + 4) * 5 = 358$ bytes, which includes the TCP and IP headers, and the Ethernet headers and trailers.

4.4.4 Rescue Control

Rescue control allows a web server to tune its resource utilization by using rescue actions that are triggered automatically based on load conditions. For example, to control network utilization ρ_n within the desired load region $[\rho_n^l, \rho_n^u]$, overload control actions are triggered if $\rho_n > \rho_n^u$, and under-load control actions are triggered if $\rho_n < \rho_n^l$. In general, to control the utilization of multiple resources such as network and CPU, overload control actions are triggered if *any* resource is heavily loaded, and under-load control actions are triggered if *all* resources are lightly loaded. For simplicity, we discuss rescue control based only on network utilization ρ_n in this section. Next, we first introduce DotSlash rescue protocol and give an overview of DotSlash rescue control. Then, we describe DotSlash rescue actions and state transitions in details.

4.4.4.1 Rescue Protocol

DotSlash rescue protocol (DSRP) allows servers of different web sites to collaborate with each other. DSRP is an application-level request-response protocol using single-line plain text messages. A request has a command string (starting with a letter) followed by optional parameters, whereas a response has a response code (three digits) followed by the response string and optional parameters. DSRP defines five requests: *SOS* for initiating a rescue relationship, *TOKEN* for transferring one token from an origin server to a rescue server, *RATE* for adjusting the allowed rate of HTTP redirects, *KEEPALIVE* for indicating a rescue server alive, and *SHUTDOWN* for terminating a rescue relationship. As shown in Figure 4.3, *SOS* and *TOKEN* requests are always sent by origin servers, and *RATE* and *KEEPALIVE* requests are always sent by rescue servers, but *SHUTDOWN* requests may be sent by either origin servers

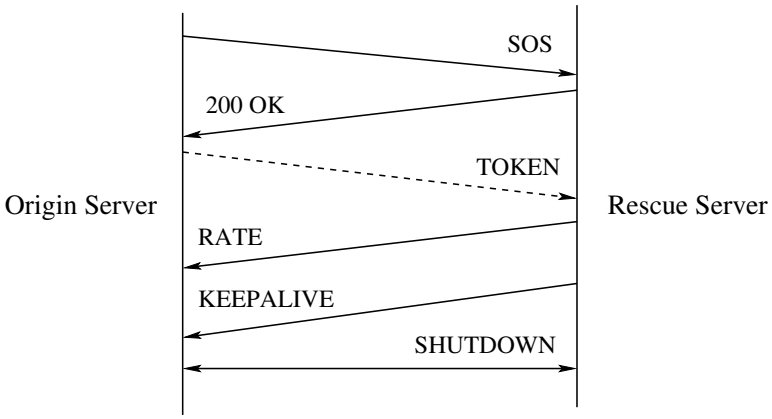


Figure 4.3: DotSlash rescue protocol (DSRP)

or rescue servers. Note that *TOKEN* requests are used only in the flood-insurance model.

To initiate a rescue relationship, an origin server sends an *SOS* request to a chosen rescue server candidate. The request has the following parameters: the origin server's fully qualified domain name, its IP address, and its port number for web requests. When a web server receives an *SOS* request, it can accept the request by sending a "200 OK" response or reject the request by sending a "403 Reject" response. A "200 OK" response has the following parameters: a unique alias of the rescue server assigned to the origin server, the rescue server's IP address, the rescue server's port number for web requests, and the allowed redirect rate and redirect data rate that the origin server can offload to the rescue server. When authentication is enable, an origin server needs to sign its *SOS* request using its private key, and a rescue server needs to authenticate the origin server before accepting the *SOS* request. When the flood-insurance model is used, an origin server needs to transfer one token to a rescue server after its *SOS* request has been accepted by the rescue server, which is done via a *TOKEN* request that carries one token signed by the origin server using its private key. When a rescue server receives a *TOKEN* request, it signs the token using its

private key, and then sends this double-signed message to the community authority for verification. When the community authority receives a *TOKEN* message, it returns "200 OK" if it can verify the token, and returns "406 Token_Invalid" otherwise.

4.4.4.2 Rescue Control Overview

We use the following control strategies. First, we use a configurable parameter τ to denote the control interval, which is the smallest time unit for performing workload monitoring and rescue control. Other time intervals are specified as a multiple of the control interval. To react quickly to load changes, we use a small control interval, default to 1 second. Secondly, to handle stochastics and avoid over-reacting to load changes, we apply an exponentially weighted moving average filter to all control inputs and measured outputs. Using network utilization ρ_n as an example, for interval k we compute $\overline{\rho_n(k)} = \alpha\overline{\rho_n(k-1)} + (1-\alpha)\rho_n(k)$, where $\rho_n(k)$ is the current raw measurement, $\overline{\rho_n(k)}$ is the filtered value of $\rho_n(k)$, $\overline{\rho_n(k-1)}$ is the previous filtered value, and α is a configurable parameter with a default value of 0.5. Third, if multiple rescue server candidates are available, the one with the largest rescue capacity is used first. This policy helps an origin server to keep the number of its rescue servers as small as possible. Minimizing the number of rescue servers can reduce their cache misses, and thus reduce the data transfer volume at the origin server.

DotSlash employs a closed-loop rescue control system by adjusting control inputs adaptively based on measured outputs and reference inputs, where control inputs are various rescue actions, measured outputs are the utilization measurements of different resources, and reference inputs are the desired utilization levels of controlled resources. Figure 4.4 illustrates the closed-loop rescue control system in DotSlash. Note that origin servers and rescue servers use different control inputs. For example, an origin server controls the *redirect probability* P_r, whereas a rescue server controls the *allowed*

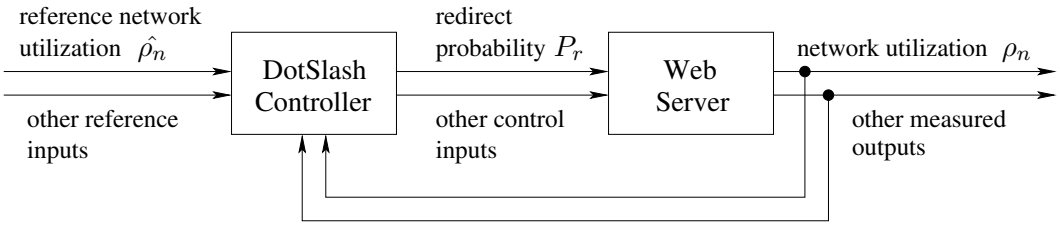


Figure 4.4: DotSlash closed-loop rescue control system

redirect data rate λ_{rd}^a for each of its origin servers. Here, P_r is the probability that an origin server redirects client requests to its rescue servers, and λ_{rd}^a is the network bandwidth that a rescue server allocates to an origin server. Note that an origin server should ensure that the real redirect data rate $\lambda_{rd} \leq \lambda_{rd}^a$, but a rescue server may experience $\lambda_{rd} > \lambda_{rd}^a$.

DotSlash uses a rule-based controller [94], which performs rescue control based on a set of heuristic rules. These rules are in the form of "**If** certain conditions are true **Then** take certain rescue actions", where conditions are specified based on the current DotSlash state and measured control outputs. For example, we have two rules for origin servers as follows: (1) if $\rho_n > \rho_n^u$, then increase P_r; and (2) if $\rho_n < \rho_n^l$, then decrease P_r. Similarly, we have two rules for rescue servers as follows: (1) if $\rho_n > \rho_n^u$, then decrease λ_{rd}^a; and (2) if $\rho_n < \rho_n^l$, then increase λ_{rd}^a. These rules are straightforward and generally applicable to all web servers. This approach allows us to build an autonomic system to automate the whole rescue process. In contrast, a conventional controller such as a proportional-integral-derivative (PID) controller [51] cannot be applied to all web servers. The reason is that in classical control theory, different web servers are different target systems, and thus they should be modeled separately. As a result, different conventional controllers are needed for different web servers.

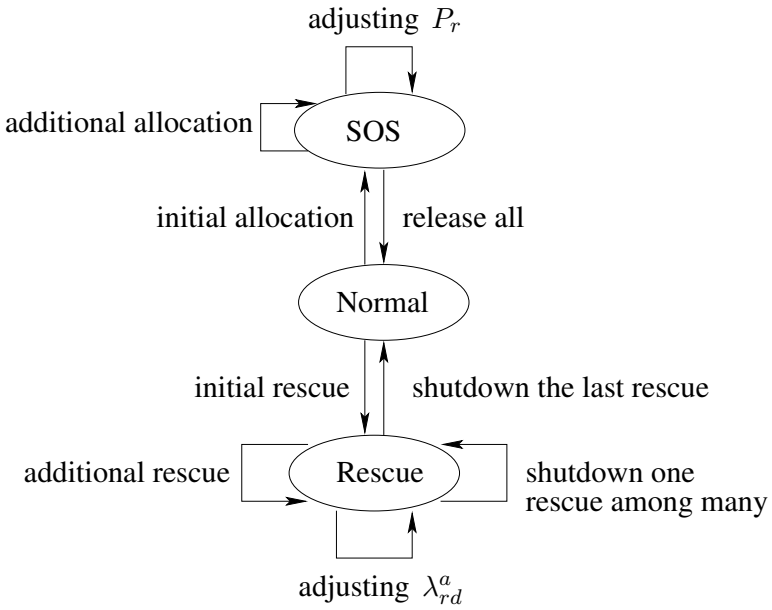


Figure 4.5: DotSlash rescue actions and state transitions

4.4.4.3 Rescue Actions and State Transitions

We summarize DotSlash rescue actions and state transitions in Figure 4.5, and describe rescue actions in each state next.

The normal state has two rescue actions: initial allocation and initial rescue. For the first case, if a web server is heavily loaded (i.e., $\rho_n > \rho^u_n$), then it needs to allocate its first rescue server, set P_r to 0.5, and switch to the SOS state. For the second case, if a web server receives a rescue request and it is lightly loaded (i.e., $\rho_n < \rho^l_n$), then it can accept the rescue request, set λ^a_{rd} to $(\hat{\rho_n} - \rho_n)\lambda^m_d$ or a smaller value determined by a rate allocation policy, and switch to the rescue state.

The SOS state has four rescue actions: increase P_r, additional allocation, decrease P_r, and release. For the first case, if an origin server is heavily loaded and it has unused redirect capacity (i.e., $\lambda_{rd} < \lambda^a_{rd}$), then it needs to increase P_r until P_r reaches 1. For the second case, if an origin server is heavily loaded and it has run out of redirect

```
// Compute β
β = ρ_n/ρ̂_n;

// Increase P_r if ρ_n > ρ_n^u
if (P_r < 1) {
  if (P_r < 0.5) {
    P_r = 0.5;
  } else if (P_r > 0.99) {
    P_r = 1;
  } else {
    t = min(βP_r, 1);
    P_r = αP_r + (1 − α)t;
  }
}

// Decrease P_r if ρ_n < ρ_n^l
if (P_r > 0) {
  if (P_r < 0.1) {
    P_r = 0;
  } else {
    t = βP_r;
    P_r = αP_r + (1 − α)t;
  }
}
```

Figure 4.6: Algorithm for adjusting P_r at an origin server

capacity (i.e., λ_{rd} equals λ^a_{rd}), then it needs to allocate an additional rescue server so as to increase its redirect capacity. For the third case, if an origin server is lightly loaded and it still redirects requests to rescue servers (i.e., $P_r > 0$), then it needs to decrease P_r until P_r reaches 0. For the last case, if an origin server has been lightly loaded and has not redirected requests to rescue servers (i.e., P_r is 0) for a configured number of consecutive control intervals, then it needs to release all rescue servers.

Figure 4.6 gives the algorithm for adjusting P_r at an origin server, which increases P_r if $\rho_n > \rho^u_n$, and decreases P_r if $\rho_n < \rho^l_n$. The adjustment is controlled by parameter $\beta = \rho_n/\hat{\rho_n}$, where $\beta > 1$ for increase since $\rho_n > \rho^u_n > \hat{\rho_n}$, and $\beta < 1$ for decrease since $\rho_n < \rho^l_n < \hat{\rho_n}$. Further, the adjustment is smoothed by using an exponentially weighted moving average filter with $\alpha = 0.5$. To allow P_r to converge quickly to 1 or 0 as needed, an increase from above 0.99 is set to 1, and a decrease from below 0.1 is set to 0. To react quickly to load spikes, an increase from below 0.5 is set to 0.5.

Figure 4.7 illustrates how P_r is adjusted at an origin server for two different workloads by using the algorithm shown in Figure 4.6. Note that ρ_n in this figure is the filtered value of the raw measurement of network utilization. For simplicity, we assume that the raw measurement of network utilization only changes at control interval 21: from 0.8 to 0.2 for workload1, and from 0.9 to 0.1 for workload2. We

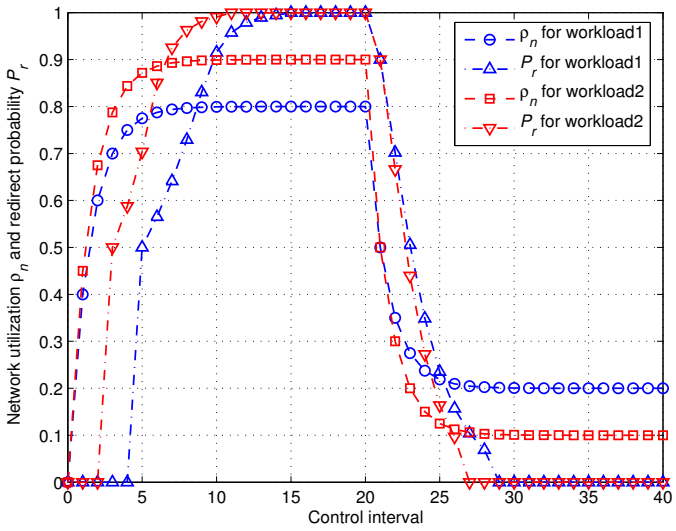


Figure 4.7: Adjusting P_r at an origin server for two different workloads by using the algorithm shown in Figure 4.6. In this figure, ρ_n is the filtered value of the raw measurement of network utilization which changes at control interval 21: from 0.8 to 0.2 for workload1, and from 0.9 to 0.1 for workload2.

use $\alpha = 0.5$, $\rho_n^l = 0.5$, $\rho_n^u = 0.75$, and $\hat{\rho}_n = (\rho_n^l + \rho_n^u)/2 = 0.625$. For initial values, $\rho_n = 0$ and $P_r = 0$. We can observe the following results from Figure 4.7. First, when the workload changes, it takes 7 control intervals for the filtered value to converge to the raw measurement. Secondly, when the workload is consistently above the upper threshold, it takes 11 to 15 control intervals to increase P_r up to 1, and the higher the workload, the quicker the P_r increase. In contrast, when the workload is consistently below the lower threshold, it takes 7 to 9 control intervals to decrease P_r down to 0, and the lower the workload, the quicker the P_r decrease. Since DotSlash uses a small control interval with a default value of 1 second, it only takes a short time (from a few seconds to tens of seconds) to adjust P_r to the desired value.

The rescue state has five rescue actions: decrease λ_{rd}^a, heavy-load shutdown, increase λ_{rd}^a, additional rescue, and idle shutdown. For the first case, if a rescue server

```
// Compute β
β = ρ_n/ρ̂_n;
if (β < 0.5) {
   β = 0.5;
} else if (β > 2) {
   β = 2;
}
```

```
// Decrease λ^a_rd if ρ_n > ρ^u_n
if (λ^a_rd > 0) {
   if (λ^a_rd < 0.1λ^m_d) {
      λ^a_rd = 0;
   } else {
      t = λ^a_rd/β;
      λ^a_rd = αλ^a_rd + (1 − α)t;
   }
}
```

```
// Increase λ^a_rd if ρ_n < ρ^l_n
if (λ^a_rd < λ̂_d && λ_rd > γλ^a_rd) {
   if (λ^a_rd > 0.99λ̂_d) {
      λ^a_rd = λ̂_d;
   } else {
      t = min(λ^a_rd/β, λ̂_d);
      λ^a_rd = αλ^a_rd + (1 − α)t;
   }
}
```

Figure 4.8: Algorithm for adjusting λ^a_{rd} at a rescue server

is heavily loaded and its $\lambda^a_{rd} > 0$, then it needs to decrease λ^a_{rd} until λ^a_{rd} reaches 0. For the second case, if a rescue server is heavily loaded and its λ^a_{rd} is 0, then it needs to shutdown the rescue relationship. When a rescue server has shutdown all rescue relationships, it switches to the normal state. For the third case, when a rescue server is lightly loaded and $\lambda^a_{rd} < \hat{\lambda}_d$, then it can increase λ^a_{rd}. Note that a rescue server should not increase λ^a_{rd} if λ_{rd} is far below λ^a_{rd}. For the fourth case, if a rescue server is lightly loaded, and it receives a new rescue request, then it can accept the rescue request, and assign a λ^a_{rd} to the new origin server. By doing so, the rescue server will have multiple origin servers, and a separate λ^a_{rd} is assigned to each origin server. For the last case, if a rescue server has an origin server whose λ_{rd} has been 0 for a configured number of consecutive control intervals, then the rescue server should shutdown the rescue relationship so as to release rescue resources in case of the origin server failure or network separation.

Figure 4.8 gives the algorithm for adjusting λ^a_{rd} at a rescue server, which decreases λ^a_{rd} if $\rho_n > \rho^u_n$, and increases λ^a_{rd} if $\rho_n < \rho^l_n$. Note that λ^a_{rd} is increased only if $\lambda_{rd} > \gamma\lambda^a_{rd}$, where γ is a configurable parameter with a default value of 0.9. This algorithm is very similar to the algorithm shown in Figure 4.6. However, these two algorithms make adjustments in opposite directions because their adjusting factors are β and $1/\beta$, respectively. We keep the adjusting factor for λ^a_{rd} within the range of

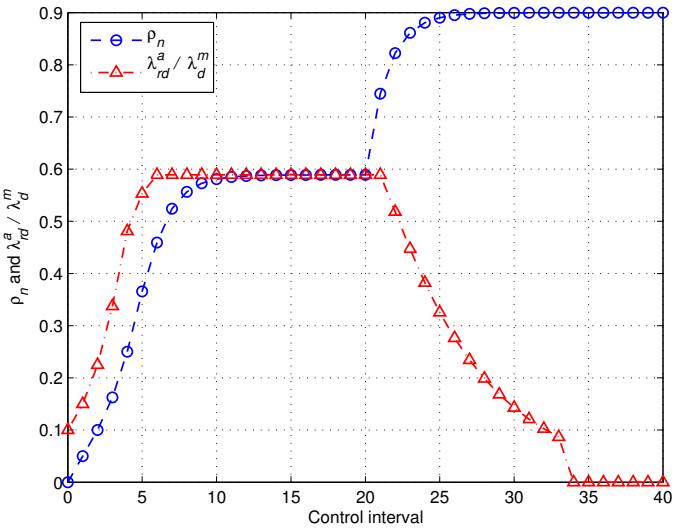


Figure 4.9: Adjusting λ_{rd}^a at a rescue server by using the algorithm shown in Figure 4.8. In this figure, ρ_n is the filtered value of the raw measurement of network utilization which changes from $\lambda_{rd}^a/\lambda_d^m$ to 0.9 at control interval 21.

$[0.5, 2]$ to avoid over-reacting adjustments. To allow λ_{rd}^a to converge quickly to $\hat{\lambda_d}$ or 0 as needed, an increase from above $0.99\hat{\lambda_d}$ is set to $\hat{\lambda_d}$, and a decrease from below $0.1\lambda_d^m$ is set to 0.

Figure 4.9 illustrates how λ_{rd}^a is adjusted at a rescue server by using the algorithm shown in Figure 4.8. Note that ρ_n in this figure is the filtered value of the raw measurement of network utilization. For simplicity, we assume that the raw measurement of network utilization changes at control interval 21 from $\lambda_{rd}^a/\lambda_d^m$ to 0.9. We use $\alpha = 0.5$, $\gamma = 0.9$, $\rho_n^l = 0.5$, $\rho_n^u = 0.75$, and $\hat{\rho}_n = (\rho_n^l + \rho_n^u)/2 = 0.625$. For initial values, $\rho_n = 0$ and $\lambda_{rd}^a = 0.1\lambda_d^m$. Similar to Figure 4.7, we can observe the following results from Figure 4.9: (1) it takes 6 to 10 control intervals for the filtered value to converge to the raw measurement, and (2) it takes 6 to 14 control intervals to adjust λ_{rd}^a to the desired value.

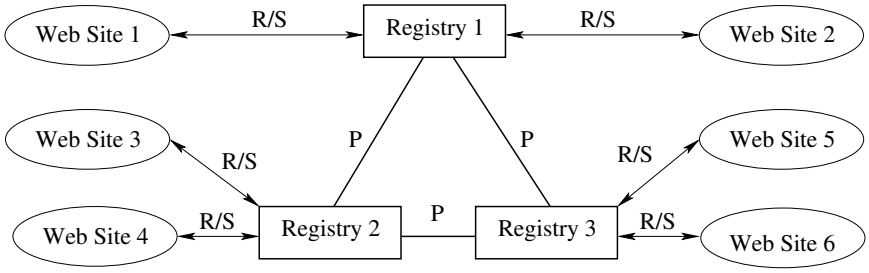


Figure 4.10: Rescue server discovery via a set of fully-meshed DotSlash registries (based on mSLP DAs), where R/S denotes registration and search operations, and P denotes a peer relationship.

4.4.5 Service Discovery

Service discovery allows servers of different web sites to learn about each other dynamically and collaborate without any administrator intervention. DotSlash uses the Service Location Protocol (SLP) [59] since it is an IETF proposed standard for service discovery in IP networks, and it is flexible, lightweight and powerful. Based on the SLP mesh enhancement (mSLP) described in Chapter 2, DotSlash uses a distributed service registries that maintain a fully-meshed peer relationship, as shown in Figure 4.10. A web server can discover the available DotSlash registries via DNS SRV [158]. For example, DotSlash registries deployed in *dot-slash.net* can be discovered via a DNS query by using *query_name=_slpda._tcp.dot-slash.net* and *query_type=srv*. After obtaining the available DotSlash registries, a web server can use any service registry to register its information and to search information about other web servers. Service registrations received by one registry will be propagated automatically to all of its peer registries, and anti-entropy (described in Chapter 3) is used to ensure consistency among all service registries. When a registry reboots after failures, it can obtain the up-to-date registration information from its peer registries. Only a small number of such service registries are needed for reliability and scalability. All of them serve the

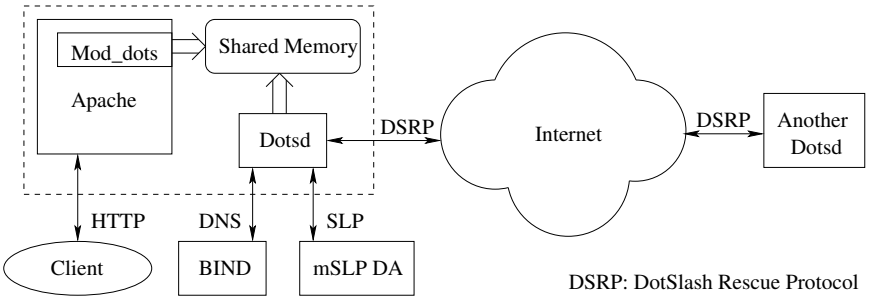


Figure 4.11: DotSlash software architecture

scope "DotSlash" (reserved for DotSlash rescue services) so that they will not affect local service discovery.

The template for DotSlash rescue services has the following attributes: the domain name for the web server, its IP address which is used to bypass DNS round robin, its port number for web requests, its port number for DotSlash rescue services, and the current allowed redirect data rate λ_{rd}^a computed as $\mathsf{max}((\hat{\rho_n} - \rho_n)\lambda_d^m, 0)$. A web server performs service registrations and rescue server searches periodically with a configurable interval τ_r and τ_s, respectively. To get ready for load spikes, a web server maintains a list of rescue server candidates. A DotSlash service search request uses preference filters [157] (see Section 2.5) that allow the registry to sort the search result based on λ_{rd}^a and to only return the desired number of matching entries, which is useful if many entries match a search request.

4.5 Implementation

We use Apache [9] as our base system since it is open source and is the most popular web server [88]. Figure 4.11 shows the DotSlash software architecture. DotSlash is implemented as two parts: *Mod_dots* and *Dotsd*. Mod_dots is an Apache module that supports DotSlash functions related to client request processing, including

accounting for each response, HTTP redirect, and dynamic virtual hosting. Dotsd is a daemon that accomplishes other DotSlash functions, including service discovery, dynamic DNS updates, and rescue control and management. For convenience, Dotsd is started within the Apache server, and is shutdown when the Apache server is shutdown. Dotsd and Mod_dots share control data structures via shared memory. DNS servers and DotSlash service registries are DotSlash components external to the Apache server. We use BIND as DNS servers, and use mSLP Directory Agents (DAs) as DotSlash service registries. A web server interacts with other web servers via its Dotsd using DSRP (see Section 4.4.4).

DotSlash control data are divided into two parts: a workload meter for the web server itself, and a peer table for collaborating web servers. The peer table maintains accounting information of redirected traffic for peers. Traffic accounting is performed at two time scales: the current control interval and the server's lifetime which is from the server's starting time to now. The former accounting is used to trigger rescue actions, and the corresponding counters are reset to zero at the end of the current control interval. The latter accounting allows computing various average traffic rates by sampling the corresponding counters at desired time intervals.

Dotsd is implemented using the pthread threading library. It has two main threads: a control thread and a DSRP server. The control thread runs at the end of each control interval for processing tasks that need to be done periodically such as computing the current workload level, triggering rescue actions if needed, and checking whether it needs to perform service discovery. The DSRP server accepts connections from other Dotsds and creates a new thread for processing each accepted connection. Dotsd also includes three clients: a DNS client for dynamic DNS updates, an SLP Service Agent for service registrations, and an SLP User Agent for service searches.

Mod_dots handles traffic accounting, performs HTTP redirects for an origin server, supports dynamic virtual hosting for a rescue server, and implements a content handler for */dotslash-status* so that a request for *http://host.domain/dotslash-status* can retrieve the current DotSlash status for the web server *host.domain*.

4.6 Evaluation

For a web server, we use two performance metrics D and R, where D is the maximum data rate (in kilobytes per second) of HTTP responses delivered to clients, and R is the maximum request rate (in requests per second) supported. Our goal is to improve a web server's D and R by using DotSlash rescue services. For a web server without using DotSlash, its D and R can be estimated as λ_d^m and $\lambda_d^m/(F+H)$, respectively, where F is the average size of requested files, and H is the average HTTP header size of responses, assuming the CPU is not a bottleneck. For any web server, the maximum rate of HTTP redirects it can support can be estimated as λ_d^m/A_r, where A_r is the accounting size for an HTTP redirect (see Section 4.4.3). Thus, a web server can improve its R and D by using DotSlash as follows. If it only uses HTTP redirect to offload client requests, its R is bounded by λ_d^m/A_r, and its D is bounded by $R(F+H)$. However, a web server can use DNS round robin to overcome this scaling limitation so as to further improve its R and D.

We performed experiments in our local area network and on PlanetLab [99] to achieve two goals. First, given a web server with a constraint on its outbound bandwidth, we want to improve its R and D by using DotSlash rescue services, and aim to achieve an improvement close to the analytical bound, i.e., the web server can handle a request rate close to λ_d^m/A_r when only HTTP redirect is used. Second, we want to confirm that our workload control algorithm works as expected.

4.6.1 Workload Generation

We use *httperf* [84] to generate workloads, which provides a flexible facility for generating various HTTP workloads and for measuring server performance. If the request rate to be generated is high, multiple httperf clients are used, each running on a separate machine. To simulate web hotspots, a small number of files are requested repeatedly from a web server. Each request uses a separate TCP connection. Thus, the request rate equals the connection rate.

We made two enhancements to httperf to facilitate experiments on DotSlash. First, we extended httperf to handle HTTP redirects automatically since an httperf client needs to follow HTTP redirects in order to complete workload migrations from an origin server to its rescue servers. Second, we wrote a shell script to support workload profiles. A workload profile specifies a sequence of request rates and their testing durations, which is convenient for describing workload changes.

For a web server, its R and D are determined as follows. We use httperf clients to issue requests to the web server, starting at a low request rate, and increasing the request rate gradually until the web server gets overloaded. A client uses 7 seconds [29] as the timeout value for getting each response. If more than 10% [29] of issued requests time out, a client declares the web server as being overloaded. For a sequence of testing request rates that are monotonically increasing, $r_1 < r_2 < \cdots$, if the web server gets overloaded at r_i, then $R = r_{i-1}$. For all testing request rates, up to R, the maximum data rate delivered to clients is D.

4.6.2 Experimental Setup

We performed experiments in our local area network (LAN) and on PlanetLab [99]. In our LAN experiments, we used a cluster of 30 Linux machines, which were connected

using 100 Mb/s fast Ethernet. These machines had two different configurations, *CLIC* and *iDot*. The former had a 1 GHz Intel Pentium III CPU and 512 MB of memory, whereas the latter had a 2 GHz AMD Athlon XP CPU and 1 GB of memory. They all ran Redhat 9.0 with Linux kernel 2.4.20-20.9. At the time of our experiments, PlanetLab consisted of more than 300 nodes distributed all over the world. Each node had a CPU of at least 1 GHz clock rate and had at least 1 GB of memory. PlanetLab nodes had four types of network connections: DSL lines, Internet2, North America commodity Internet, and outside North America. They all ran Redhat 9.0 with Linux kernel 2.4.22-r3_planetlab and PlanetLab software 2.0.

We set up the DotSlash software in three steps. First, we compiled Apache 2.0.48 with the *worker* multi-processing module, the proxy modules, the cache modules, and our DotSlash module. We configured Apache as follows. Since reverse proxying was taken care of by DotSlash automatically, no proxy configuration was needed. Web caching was configured with 256 KB of memory cache, and 10 MB of disk cache, and the maximum file size allowed in memory cache was 20 kB. For the DotSlash module, we only configured λ_d^m. Second, we used BIND 9.2.2 as the DNS server software, and set up a DNS domain *dot-slash.net*. All rescue servers registered their virtual host names in this domain via dynamic DNS updates. We tested DotSlash workload migration via HTTP redirect and DNS round robin. In this section, we give experimental results for workload migration via HTTP redirect only because the results for DNS round robin may vary from time to time due to DNS caching. Third, we set up a DotSlash service registry using an mSLP DA. Each web server registered itself with this service registry, and discovered other web servers by looking up this registry.

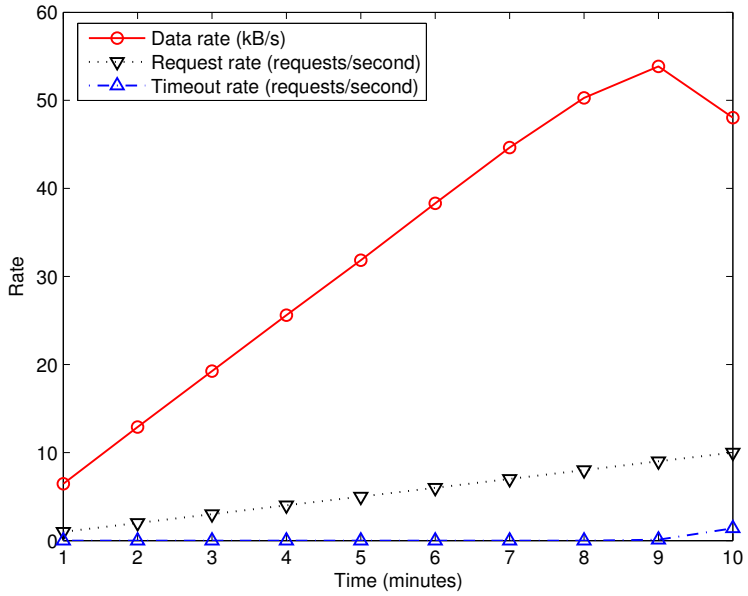
4.6.3 Experimental Results on PlanetLab

We ran a web server on a PlanetLab DSL node, *planetlab1.gti-dsl.nodes.planet-lab.org* (referred to as *gtidsl1*), for which the outbound bandwidth was the bottleneck. We ran httperf on a local CLIC machine. Ten files were requested repeatedly from *gtidsl1*, with an average size of 6 KB [136]. Our goal is to measure, from the client side, *gtidsl1*'s R and D in two cases, namely without using DotSlash versus using DotSlash.

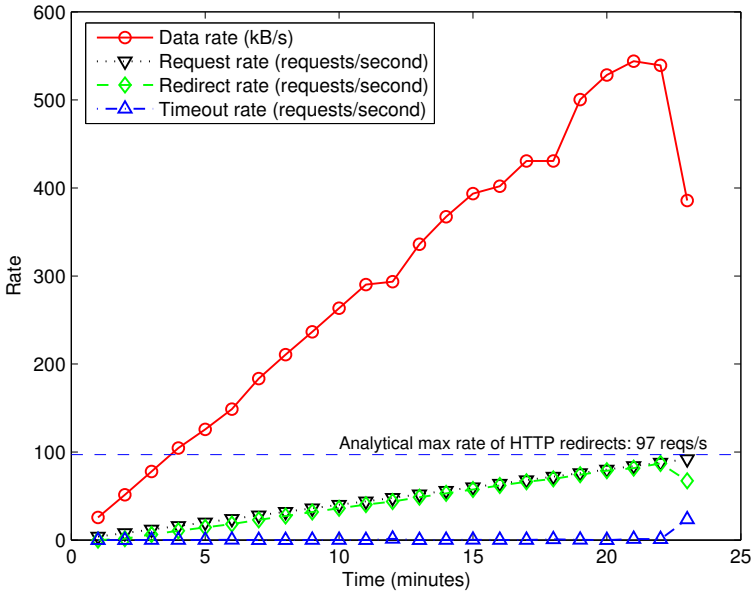
For the first case, DotSlash was disabled. The request rate started at 1 request/second, increased to 20 requests/second, with a step size of 1, and each request rate lasted for 60 seconds. Figure 4.12(a) shows the experimental results. In this figure, *gtidsl1* got overloaded at 10 requests/second, where 14% of requests, 84 out of 600, timed out. Thus, R was 9 requests/second. The measured D was 53.9 kB/s (1 kB = 1000 bytes), attained when the request rate was R.

For the second case, DotSlash was enabled. We set *gtidsl1*'s λ_d^m to 53.9 kB/s. To provide needed rescue capacity for *gtidsl1*, we ran another web server on a local *iDot* machine named as *maglev*, and its λ_d^m was set to 2000 kB/s. The request rate started at 4 requests/second, increased to 200 requests/second, with a step of 4, and each request rate lasted for 60 seconds. Figure 4.12(b) shows the experimental results. In this figure, when the request rate reached 8 requests/second, the origin server *gtidsl1* started to redirect client requests via HTTP redirects to the rescue server *maglev*. As the request rate increased, the redirect rate increased accordingly. Eventually, *gtidsl1* redirected almost all clients requests to *maglev*. In this experiment, *gtidsl1* got overloaded at 92 requests/second, where 25% of requests, 1404 out of 5520, timed out. Thus, R was 88 requests/second. The measured D was 544.1 kB/s, attained when the request rate was 84 requests/second.

Comparing the results obtained from the above two cases, we have $88/9 = 9.78$,



(a) Without using DotSlash rescue services



(b) Using DotSlash rescue services

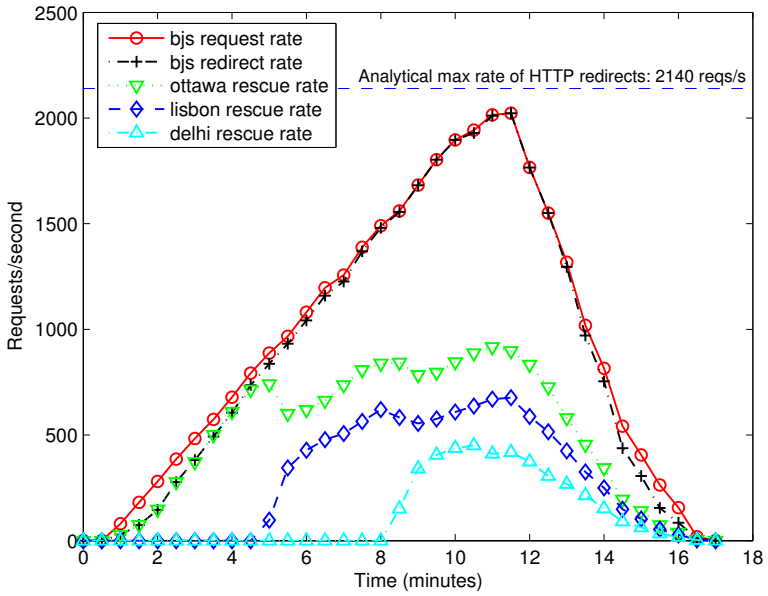
Figure 4.12: The data rate and request rate for a PlanetLab DSL node *gtidsl1* in two cases, namely without using DotSlash verses using DotSlash. Note that figure (a) and (b) have different scales of ordinates, and 1 kB = 1000 bytes.

and $544.1/53.9 = 10.1$, meaning that by using DotSlash rescue services, we got about an order of magnitude improvement for *gtidsl1* on its R and D, even if only HTTP redirect is used. To show the effectiveness of DotSlash, we also compare R with its analytical bound λ_d^m/A_r below. In this experiment, we only measured λ_d^m at *gtidsl1*, without knowing its outbound bandwidth B. To be conservative, we use $U = (F + H)/(F + H + O) = 95\%$, where $F = 6$ KB, $H = 250$ bytes, and $O = 358$ bytes. Here the header overhead O for a single-request HTTP transaction is the same as that for an HTTP redirect (calculated in Section 4.4.3). Since the size of an HTTP redirect response is $n = 227$ bytes in the experiment, we have $A_r = (n + O)U = 556$ bytes. As a result, R is bounded by $\lambda_d^m/A_r = 53.9 * 1000/556 = 97$ requests/second, and we achieved $88/97 = 91\%$ of its analytical bound.

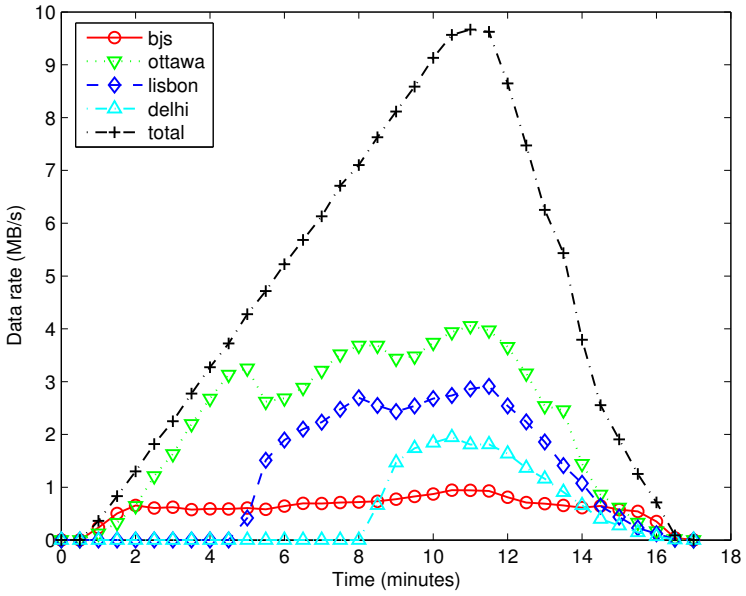
4.6.4 Experimental Results in Local Area Networks

In the previous section we have shown the performance improvement, measured from the client side, for a web server by using DotSlash rescue services in a wide area network setting. In this section we will show, via an inside look from the server side, how workload migrates from an origin server to its rescue servers. The workload monitoring component in DotSlash maintains a number of counters for outbound HTTP traffic, including total bytes served, the number of client requests served, the number of client requests redirected, and the number of requests served for rescuing others. The values of these counters for a web server *host.domain* can be obtained from *http://host.domain/dotslash-status?auto*. By sampling these counters at a desired interval, we can calculate the needed average values of outbound data rate, request rate, redirect rate, and rescue rate.

In this experiment, we ran four machines, *bjs*, *ottawa*, *lisbon*, and *delhi*, as web



(a) The request rate and redirect rate at *bjs*, and the rescue rates at its rescue servers



(b) The data rate at each web server, and the total data rate of all web servers

Figure 4.13: The request rates and data rates at the origin server *bjs* and its rescue servers

servers, where *bjs* is an *iDot* machine, and the other three are *CLIC* machines. To emulate a scenario where *bjs* works as an origin server with a bottleneck on its outbound bandwidth, and the remaining web servers work as rescue servers, we configured their λ_d^m as 1000, 7000, 5000, and 3000 kB/s, respectively. We ran httperf on five *CLIC* machines, which issued requests to *bjs* using the same workload profile. The maximum request rate was $400 * 5 = 2000$ requests/second, and the duration of the experiment was 15 minutes. Ten files were requested repeatedly, with an average size of 4 KB. We ran a shell script to get the DotSlash status from the four web servers at an interval of 30 seconds. The retrieved status data were stored in round-robin databases using RRDtool [110], with one database for each web server. Figure 4.13 shows the data rates and request rates for the four web servers over a duration of 17 minutes.

We observe the following results from Figure 4.13(a). First, *bjs* can support a request rate of 2000 requests/second, which is close to λ_d^m/A_r, the analytical maximum rate of HTTP redirects at *bjs*. Since $A_r = (n + O)U = 468$ bytes in this experiment, where $n = 227$ bytes, $O = 358$ bytes, and U takes its default value 80%, we have $\lambda_d^m/A_r = 2140$ requests/second. Second, the redirect rate at *bjs* increases as the request rate increases, and it is roughly the same as the request rate once it is above 1500 requests/second. The reason is that *bjs* increases its redirect probability P_r as its load increases. When the rate of HTTP redirects is greater than $\lambda_d^m \rho_n^u/A_r = 1603$ requests/second, P_r will stay at 1, that is all client requests are redirected from *bjs* to its rescue servers. Third, *bjs* allocates one rescue server at a time, and uses the one with the largest rescue capacity first. When a new rescue server is added in, the rescue rates at the existing rescue servers decrease. Also, the rescue rates at rescue servers are proportional to their rescue capacities because of the WRR load distribution at *bjs*.

Comparing Figure 4.13(b) and 4.13(a), we observe that rescue servers have simi-

larly shaped curves for their data rates and rescue rates. In contrast, as we expected, the origin server *bjs* has quite a different shape for its request rate and data rate curves: its request rate increases significantly from 200 requests/second at 1.5 minutes to 2000 requests/second at 11 minutes, but its data rate is roughly unchanged, staying at $\lambda_d^m \rho_n^u = 750$ kB/s for the most part. This indicates that *bjs* has successfully migrated its workload to its rescue servers under the constraint of its outbound bandwidth. Also, we observe that when the request rate is between 1600 and 2000 requests/second, the data rate at *bjs* rises above 750 kB/s, but still stays below $\lambda_d^m = 1000$ kB/s. This is because *bjs* can only support a rate of 1600 requests/second for HTTP redirects with a data rate of 750 kB/s. Furthermore, we observe that the total data rate of all web servers has a maximum value of 9.7 MB/s, which is higher than 9.2 MB/s, the maximum data rate measured from the httperf clients. The difference is due to our special accounting for HTTP redirects. As described in Section 4.4.3, an HTTP redirect is 227 bytes, but is counted as 468 bytes, which results in a rate increase of $241 * 2000 = 0.482$ MB/s for 2000 HTTP redirects.

4.7 Summary

This chapter described the DotSlash framework. As a rescue system, DotSlash complements the existing web server infrastructure to handle web hotspots effectively. It is self-configuring, scalable, cost-effective, easy to use, and transparent to clients. Through our experimental results, we have demonstrated that by using DotSlash a web server can increase the request rate it supports and the data rate it delivers to clients by an order of magnitude, even if only HTTP redirect is used. Using DNS round robin and HTTP redirect together would further improve the performance.

Chapter 5

Hotspot Rescue for Dynamic Content by Replicating Application Programs Dynamically

The previous chapter described the DotSlash framework that enables a web site to build an adaptive distributed web server system across wide area networks on the fly [150]. By effectively removing the bottlenecks at access network bandwidth and web servers, the DotSlash base system is sufficient for handling web hotspots at *static content* web sites. To perform hotspot rescue for *dynamic content*, DotSlash needs to address the bottlenecks at application servers and database servers. This chapter describes replicating application programs dynamically from an origin server to its rescue servers, eliminating the application server bottleneck [151]. The next chapter will describe using on-demand distributed query result caching to relieve the database server bottleneck [154]. In this chapter, we first introduce the problem of web hotspots for dynamic content and discuss related work. Then, we describe dynamic script replication. After presenting the experimental results and evaluation,

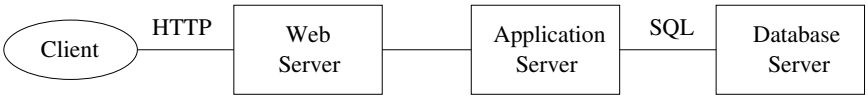


Figure 5.1: The three-tier architecture for dynamic content web sites

we give a summary.

5.1 Introduction

Handling web hotspots at dynamic content web sites is a challenging problem. First, a dynamic content web site is more likely to be overwhelmed by flash crowds because the request rate it supports is often much lower than that of a static content web site [34] since generating dynamic web pages consumes more CPU cycles than serving static web pages. Secondly, many existing caching mechanisms are designed for static content, and cannot be applied to dynamic content directly [34, 42, 131]. Furthermore, a dynamic content web site typically has a three-tier architecture as illustrated in Figure 5.1: a front-end web server handles the HTTP requests from clients, an application server implements the business logic, and a back-end database server stores the content. Depending on different applications and system configurations, different servers in the infrastructure may become the bottleneck [7, 33].

Dynamic content can be generated using different technologies, such as PHP, Active Server Pages (ASP), Java Server Pages (JSP), Java Servlets, and Enterprise Java Beans (EJB). PHP is the most popular dynamic web technology used with Apache, and Apache is the most popular web server. Thus, we discuss web hotspot rescue in the context of the common LAMP (Linux, Apache, MySQL, and PHP) configuration, and expect that similar techniques can be applied to other configurations of dynamic content web sites [33, 127]. In the LAMP configuration as shown in Figure 5.2, PHP

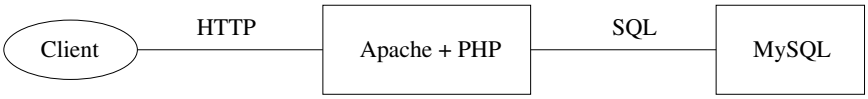


Figure 5.2: The LAMP configuration for dynamic content web sites

is a module of the Apache web server. Note that there is no separate application server; instead, the Apache server performs the tasks of an application server as well. In LAMP implementation, the Apache server and database server are usually running on separate machines.

Previous work [7] has shown that different applications may have different bottlenecks in the LAMP configuration. The database server is the bottleneck for the TPC-W benchmark [128] that models online bookstores such as amazon.com. But the web/application server is the bottleneck for the RUBiS benchmark [112] that models auction sites such as eBay, and for the RUBBoS benchmark [111] that models bulletin board sites such as Slashdot. We focus on the web/application server bottleneck in this chapter, and will address the database server bottleneck in the next chapter. Our approach is as follows. When a web server is heavily loaded, it drafts a number of rescue servers from other web sites on the fly, and redirects a fraction of client requests to those rescue servers. To serve redirected client requests, a rescue server retrieves the PHP scripts dynamically from its origin server, caches the scripts locally, and accesses the origin database server directly. We have implemented a prototype of the DotSlash rescue system for the LAMP configuration, and tested our implementation using the RUBBoS bulletin board benchmark [7]. Experiments show that by using DotSlash, a dynamic content web site can completely remove its web/application server bottleneck, and can support a request rate constrained only by the capacity of its database server.

5.2 Related Work

Various approaches have been proposed to cache dynamic content. Web caching can cache entire HTML pages or page fragments at proxies [42], web servers [34], application servers [63, 20], and edge servers [5]. Database caching [6, 26, 72] can cache data from the back-end database at database caches closer to the application server. Complementary to existing caching mechanisms, DotSlash allows a web site to expand its capacity dynamically as load increases without administrator intervention. In particular, DotSlash allows a web server to obtain additional computing capacity on demand and replicate scripts dynamically.

In edge computing [5], application components can be offloaded to edge servers, but manual configuration is needed to choose the components to be offloaded and where to deploy applications. In ACDN [102], applications can be deployed and re-deployed dynamically, but manual administration is still involved such as creating a meta-file for each application to be replicated. In contrast, DotSlash is self-managing by replicating each script file on demand and fully automatically.

5.3 Dynamic Script Replication

To support load migration for dynamic content, we enhance DotSlash with dynamic script replication, which allows a rescue server to dynamically replicate scripts from its origin servers, and cache the scripts locally. The motivation is that running scripts consumes a fair amount of CPU cycles, and the CPU often becomes the bottleneck for dynamic content web sites [7].

In DotSlash, an origin web server uses both DNS round robin and HTTP redirect to offload a fraction of client requests to its rescue servers [150]. For clarity, we omit

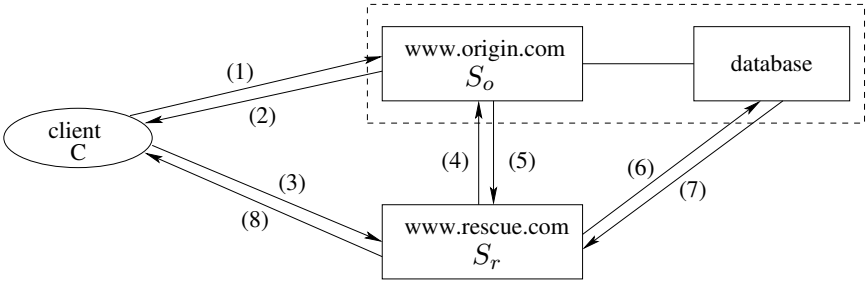


Figure 5.3: An example for dynamic script replication

the DNS name resolution steps, and give an example that uses HTTP redirect to illustrate how dynamic script replication works. In Figure 5.3, the origin server S_o is *www.origin.com*, and the rescue server S_r is *www.rescue.com*. The client C takes the following steps to retrieve *http://www.origin.com/search.php?name=x*.

1. C makes an HTTP request to S_o using
   http://www.origin.com/search.php?name=x.

2. S_o sends an HTTP redirect to C as
   http://vh1.www.rescue.com/search.php?name=x.
   Note that S_o has already set up a rescue relationship with S_r.

3. C makes an HTTP request to S_r using
   http://vh1.www.rescue.com/search.php?name=x.

4. S_r makes an HTTP request to S_o using *http://www.origin.com/search.php* because of a cache miss for the script file *search.php*.

5. S_o sends the script file *search.php* to S_r, and S_r caches *search.php* locally.

6. S_r runs *search.php?name=x* to access the corresponding database.

7. S_r gets the query results from the database.

8. S_r sends the query results to C.

5.3.1 Operations at the Rescue Server

When the rescue server S_r receives a request Q, it checks whether Q is a redirected request for dynamic content. If Q uses an alias of S_r such as *vh1.www.rescue.com*, or uses an origin server's name such as *www.origin.com*, then Q is a redirected request. If the requested file extension matches one of the configured script extensions, then Q requests dynamic content. In Apache, script extensions are configured using the directive *AddType*, e.g., files with an extension of *php* or *phtml* can be configured as PHP scripts using "*AddType application/x-httpd-php .php .phtml*".

If Q is a redirected request for dynamic content such as *http://vh1.www.rescue.com/search.php?name=x*, then S_r maps Q's URI to a script file, and sets the needed environment variables for retrieving PHP scripts if there is a cache miss. In Apache, environment variables for sub-processes are set in a per request table *subprocess_env*. DotSlash sets three environment variables: *Origin_Server* which specifies the origin server's name, *Origin_Port* which specifies the origin server's port number for web requests, and *Script_Root* which specifies the root directory for replicated scripts. Since S_r may need to retrieve and cache scripts from multiple origin servers, a request URI is mapped to its script file as *Script_Root/Origin_Server/URI_Path*, where *URI_Path* is the path part of the request URI. For example, Q's URI is mapped to a script file Q_f as *Script_Root/www.origin.com/search.php*.

If Q_f exists, the script will be executed normally; otherwise, a "file not found" error will be triggered, and be handled by a 404 handler as follows. If *Script_Root* is set (i.e., a redirected request for dynamic content), the DotSlash inclusion function *dots_include* is invoked; otherwise, a regular "file not found" message is returned.

Dynamic script replication is performed by *dots_include* using the following steps.

1. Retrieve the script file from S_o using *http://Origin_Server:Origin_Port/URI_Path*;
2. Add a header H to the retrieved script file for handling file inclusions (see Section 5.3.3 for details);
3. Set query variables (extracted from the query part of the request URI) in *$_GET* or *$_POST*;
4. Run the script by invoking the native PHP *include*.

File locking is used to ensure that partially retrieved script files are not used by concurrent requests.

5.3.2 Operations at the Origin Server

When the origin server S_o receives a request Q, it checks whether Q is from a rescue server (based on its rescue server list, see Section 4.4.1 for details), and whether Q is for dynamic content. If so, S_o will return the script file to the rescue server instead of running the script.

5.3.3 File Inclusions in Replicated Scripts

In PHP, file inclusions are supported via *include*, *require*, *include_once*, and *require_once* statements. The *include* and *require* statements are identical in every way except how they handle failures: *include* produces a warning while *require* results in a fatal error. The *include_once* and *require_once* statements are used to ensure that any file is included just once. A challenging issue here is that a replicated script running at a rescue server may include files located at the origin server.

We investigated two options for handling file inclusions in replicated scripts: renaming and error handler. The renaming approach is to rename each PHP inclusion statement to the DotSlash inclusion function *dots_include* after a script is replicated from the origin server to the rescue server. This approach is applicable to all PHP inclusion statements, but it needs to parse each replicated script file. The error handler approach is to use a customized error handler for each replicated script file. In PHP, if a file to be included does not exist, an error will be triggered. Thus, we can use a customized error handler to catch file inclusion errors, and replicate needed script files dynamically. Note that the error handler approach is applicable to the *include* and *include_once* statements only because a missing file for the *require* or *require_once* statement will halt processing of the script.

We employ the error handler approach in DotSlash mainly because it is easier to build. We add a header H to each replicated script file, which uses *set_error_handler* to set the error handler to the DotSlash error handler *dots_error*. As a wrapper function of *dots_include*, *dots_error* implements the PHP error handler API, and invokes *dots_include* in case of a file inclusion error.

5.3.4 Implementation

DotSlash functions, *dots_include* and *dots_error*, can be implemented as PHP user functions written in PHP scripts, or as PHP native functions written in C and compiled as the DotSlash extension to the PHP module. For efficiency considerations, we have implemented *dots_include* and *dots_error* as PHP native functions for PHP 4.3.6.

5.4 Evaluation

We use R to denote the maximum request rate supported by a web server. Our goal is to improve a web server's R by using DotSlash.

5.4.1 Experimental Setup

We performed experiments in our local area network, where we used a cluster of 30 Linux machines connected via 100 Mb/s fast Ethernet. These machines had two different configurations. The low-end configuration (*LC*) had a 1 GHz Intel Pentium III CPU and 512 MB of memory, whereas the high-end configuration (*HC*) had a 2 GHz AMD Athlon XP CPU and 1 GB of memory. They all ran Redhat 9.0 with Linux kernel 2.4.20-20.9.

We ran a varying number of web servers in different experiments. All web servers ran Apache 2.0.49, configured with PHP 4.3.6, *worker* multi-processing module, proxy modules, cache modules, and our DotSlash module. The PHP module included our DotSlash extension, which implemented the DotSlash inclusion function *dots_include* and the DotSlash error handler *dots_error*. In all experiments, we ran one database server on an *HC* machine denoted as *DB_HC*. The database server ran MySQL 4.0.18. To support a large number of concurrent connections, we configured MySQL with *open_files_limit=65535* and *max_connections=2048*.

We tested our prototype system using the RUBBoS bulletin board benchmark [7]. RUBBoS is modeled after an online news forum like Slashdot [120]. It consists of 19 PHP scripts, and the size of script files varies between 1 KB and 7 KB. The database has a size of 439 MB, and contains 500000 users and 2 years of stories and comments. There are 15 to 25 stories per day, and 20 to 50 comments per story. The length of story and comment bodies is between 1 KB and 8 KB.

We used RUBBoS clients to generate workloads. Each RUBBoS client can simulate a few hundred HTTP clients. An HTTP client issues a sequence of requests using a think time that follows a negative exponential distribution, with an average of 7 seconds [128]. If the request rate to be generated is high, multiple RUBBoS clients are used, each running on a separate machine. We use 7 seconds [29] as the timeout value for getting the response for a request. If more than 10% [29] of issued requests time out, the web server is considered as being overloaded.

5.4.2 Effectiveness

To show the effectiveness of DotSlash, we measured R at an origin web server from the client side in different cases, based on whether DotSlash was used or not, and whether the origin server ran on an *HC* machine or on an *LC* machine.

For the first experiment, we ran the origin web server on an *HC* machine denoted as *Orig_HC*, and DotSlash was disabled. Figure 5.4 shows the experimental results. We denote the total number of HTTP clients as N_c. The request rate at *Orig_HC* increased as N_c increased. The measured R was 118 requests/second obtained when $N_c = 900$. When N_c reached 1100, 11% of requests timed out. At this workload, the CPU utilizations of *Orig_HC* and *DB_HC* were 100% and 45%, respectively. Clearly, the web server was the bottleneck, although it had the same hardware configuration as the database server.

In the second experiment, the origin web server still ran on *Orig_HC*, but DotSlash was enabled, and rescue servers were added automatically as load increased. All rescue web servers ran on *LC* machines. We also show the experimental results in Figure 5.4, but for $N_c \geq 500$ only since *Orig_HC* did not use any rescue server when $N_c \leq 400$. The measured R was 245 requests/second obtained when $N_c = 1900$, and

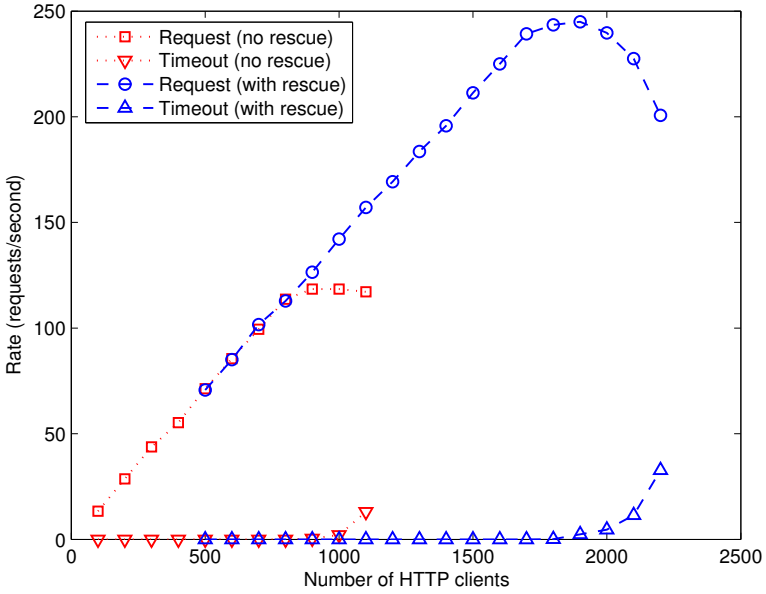


Figure 5.4: The request rate and timeout rate for the origin web server *Orig_HC* in two cases, namely without using DotSlash verses using DotSlash.

Orig_HC used 9 rescue servers. When N_c reached 2200, the database server *DB_HC* got overloaded, where 16% of requests timed out, and *Orig_HC* used 10 rescue servers. At this workload, the CPU utilizations of *Orig_HC* and *DB_HC* were 60% and 100%, respectively, and the CPU utilizations of all rescue servers were below 60%.

Comparing the above two experiments, we have two results. First, in terms of the R supported by *Orig_HC*, we have $245/118 > 2$, meaning that we doubled the performance by using DotSlash. Secondly, based on the CPU utilization, we can observe that when DotSlash is used, the origin web server is no longer a bottleneck, and the performance is constrained only by the database server. To further verify this observation, we repeated the above two experiments by running the origin web server on an *LC* machine denoted as *Orig_LC* so that we had a low-end origin web server and a high-end database server. To save space, we summarize the experimental

results as follows without showing figures.

Without using DotSlash, the measured R was 49 requests/second obtained when $N_c = 500$. *Orig_LC* got overloaded when N_c reached 600, where 21% of requests timed out. When DotSlash was used, the measured R was 245 requests/second obtained when $N_c = 1900$, and *Orig_LC* used 10 rescue servers. *DB_HC* got overloaded when N_c reached 2200, where 16% of requests timed out, and *Orig_LC* used 12 rescue servers. Thus, we have $245/49 = 5$, meaning that we improved the R at *Orig_LC* by 500% by using DotSlash. The reason for using 10 rescue servers to get this improvement is that the origin server and rescue servers have a CPU utilization close to 50% because we have configured the desired load region in our experiments as $[45\%, 70\%]$. More specifically, *Orig_LC* can support a rate of 49 requests/second with 100% CPU utilization. Thus, to support a rate of 245 requests/second, we need 5 such web servers with 100% CPU utilization, or equivalently, we can use 11 such web servers (i.e., 1 origin server and 10 rescue servers) with $5 * 100\%/11 = 45\%$ CPU utilization.

From the above experiments, we can observe that using DotSlash can completely remove the web server bottleneck, and the performance of a dynamic content web site is constrained only by its database server. Also, when DotSlash is used, it does not make much difference as to using a high-end web server or a low-end web server. For example, to support a rate of 245 requests/second, *Orig_HC* uses 9 rescue servers whereas *Orig_LC* uses 10 rescue servers, where the performance ratio of *Orig_HC*/*Orig_LC* is about 2.

5.4.3 Workload Control and Migration

DotSlash monitors workload by maintaining a number of counters for outbound HTTP traffic and CPU utilization, and allows these counter values to be retrieved

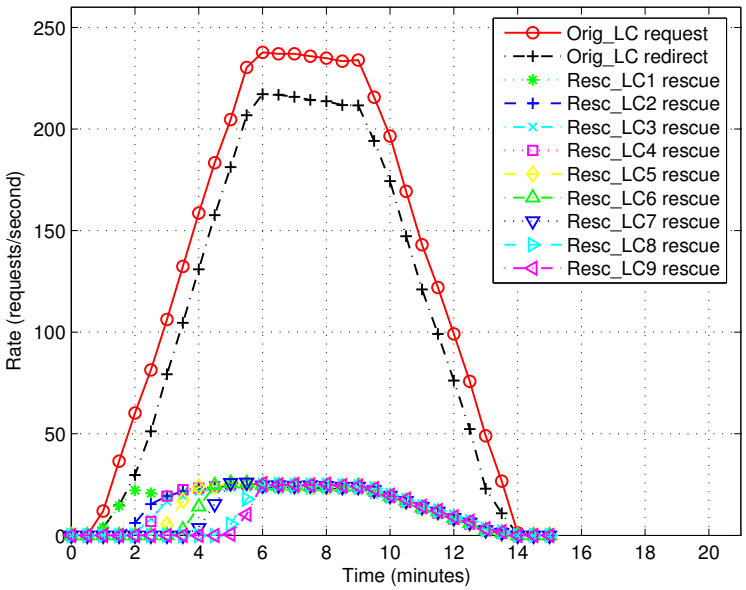


Figure 5.5: The request rate and redirect rate at the origin server *Orig_LC* and the rescue rate at the 9 rescue servers (*Resc_LC1*, ..., *Resc_LC9*)

conveniently via *http://host.domain/dotslash-status?auto*. By sampling these counters at a desired interval, we can calculate the needed average values of request rate, redirect rate, rescue rate, and CPU utilization.

To show how workload is controlled and migrated at the server side, we performed the following experiment. The origin web server ran on *Orig_LC*, and all rescue web servers ran on *LC* machines, denoted as *Resc_LC1*, ..., *Resc_LCn*. DotSlash was enabled, and rescue servers were added automatically as load increased. We ran 5 RUBBoS clients, all using the same workload profile to issue requests to *Orig_LC*. Each RUBBoS client simulated 340 HTTP clients, thus a total of 1700 HTTP clients were simulated. We started one RUBBoS client at a time, with an interval of 1 minute, and each RUBBoS client ran for 8 minutes. We ran a shell script to get the DotSlash status from all servers at an interval of 30 seconds.

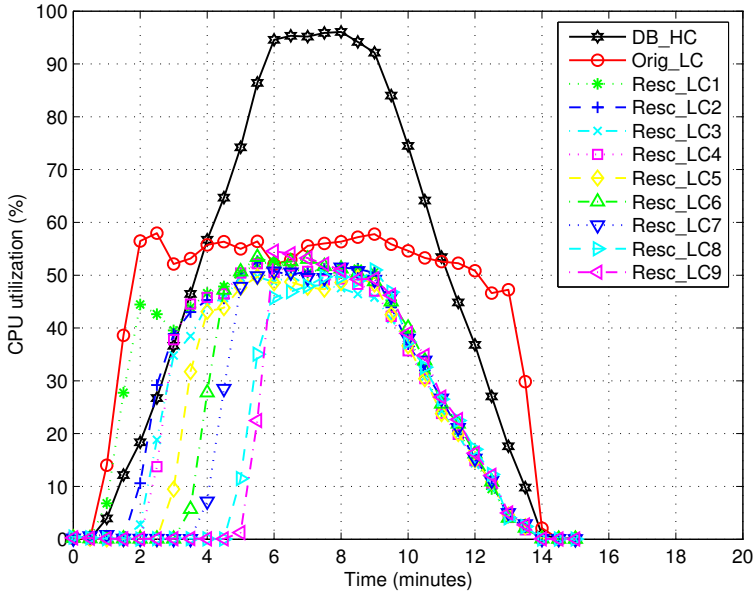


Figure 5.6: The CPU utilization for the origin server *Orig_LC*, the 9 rescue servers (*Resc_LC1*, ..., *Resc_LC9*), and the database server *DB_HC*.

Figure 5.5 shows the request rate and redirect rate at *Orig_LC* and the rescue rate at the 9 rescue servers in a duration of 15 minutes. We can observe the following results. First, the redirect rate at *Orig_LC* increases as the request rate increases, meaning that excess workload is migrated from *Orig_LC* to its rescue servers via redirects. Secondly, the serving rate (i.e., the request rate minus the redirect rate) at *Orig_LC* decreases as the redirect rate increases because redirects consume CPU cycles. Also, the serving rate should be 22–34 requests/second for the desired CPU load region of $[45\%, 70\%]$ and a capacity rate of 49 requests/second, but the real serving rate is bit smaller, 20–30 requests/second, due to the redirect overhead. Finally, the rescue rate at all 9 rescue servers is about 25 requests/second, which is the workload that drives the CPU utilization to about 50% at the rescue servers.

Figure 5.6 shows the CPU utilization for *Orig_LC*, the 9 rescue servers, and

DB_HC. We can observe the following results. First, *Orig_LC* has successfully controlled its CPU utilization to stay within 50–60%. Secondly, all rescue servers have a CPU utilization of 45–55%, being close to 50% mostly. Finally, when N_c reaches 1700, *DB_HC* has a CPU utilization around 95%, meaning that without relieving the database server bottleneck, there is not much potential to further increase the request rate.

5.5 Summary

This chapter described using DotSlash to perform hotspot rescue for dynamic content. By supporting dynamic script replication, DotSlash can completely remove the web/application server bottleneck at dynamic content web sites. Although we discussed DotSlash in the context of LAMP configuration, we expect that similar techniques can be applied to other types of dynamic content web sites.

Chapter 6

Hotspot Rescue for Dynamic Content by Using On-demand Distributed Query Result Caching

The previous two chapters described DotSlash rescue services, which enable a web site to build an adaptive distributed web server system on the fly and replicate application programs dynamically, effectively relieving a spectrum of bottlenecks ranging from access network bandwidth to web servers and application servers [150, 151]. This chapter describes DotSlash *Qcache* services that allow a web site to use on-demand distributed query result caching, greatly reducing the workload at read-mostly databases [154]. In this chapter, we first introduce the issue of database scalability for web applications. Then, we describe the design of our on-demand distributed query result caching in details. After presenting an extensive performance evaluation for our prototype system, we discuss related work and give a summary.

6.1 Introduction

Database scalability is an important issue for web applications. First, the database can be the most constrained resource in certain web applications such as on-line bookstores [7]. Secondly, after other bottlenecks have been removed, the database server will become a bottleneck at certain point if the load continues to increase. There has been a large body of research work on database replication, partition, caching, and clustering for improving database scalability [100, 118, 6, 26, 72, 32]. However, existing systems often involve manual configuration, making them difficult to be deployed dynamically to new servers. This chapter describes DotSlash *Qcache* services that allow a web site to set up on-demand distributed query result caching on the fly, which can greatly reduce the workload at read-mostly databases. The novelty of this work is that our query result caching is on demand, and operated based on load conditions: caching remains inactive as long as the load is normal, but is activated once the load is heavy. This approach offers good data consistency for normal load, and good scalability with relaxed data consistency under heavy load. Furthermore, our query result caching is self-configuring and transparent to web users and applications. DotSlash Qcache services complement DotSlash rescue services; together they provide a comprehensive solution to address different bottlenecks at multi-tier web sites. We have prototyped our system for the common LAMP (Linux, Apache, MySQL, and PHP) configuration, and performed an extensive evaluation using the RUBBoS bulletin board benchmark [111]. Our experiments show that DotSlash can increase a web site's maximum request rate supported by a factor of 10 for the RUBBoS read-only mix.

6.2 System Design

This section describes our system design. We first outline our major design goals, chosen scalability mechanism, the application model, and our system architecture. We then give details about our on-demand query result caching, caching-enhanced data driver, and flexible caching storage engine.

6.2.1 Design Goals

Our design goals are dynamic scalability, self-configuration, and transparency. First, we aim to provide a mechanism that can be deployed to new servers on demand so as to improve database scalability dynamically for web applications. Since deploying a scalability mechanism dynamically incurs an overhead at the origin server, we need to reduce this overhead as much as possible. Secondly, our system is designed to be self-configuring, handling dramatic load spikes autonomically without any administrative intervention. Finally, our system aims to be transparent to web users and applications. Without the need to change existing applications and user browsers, our system is easy to deploy.

6.2.2 Scalability Mechanisms

A spectrum of mechanisms can be used to improve database scalability. In general, caching and replication are good for read-mostly databases, whereas partitioning may be useful when updates are frequent. For the purpose of handling web hotspots, we focus on read-mostly databases, which are common for web applications such as content management systems (CMS), blogs, and web forums. Compared to replication, caching is easier to deploy dynamically, and incurs lower overhead at the origin server because cached objects are distributed from the origin server to caches on-demand,

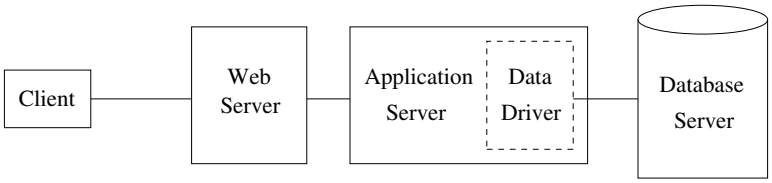


Figure 6.1: DotSlash Application Model

avoiding unnecessary data transfers. Thus, we narrow down our option to database caching.

In terms of database caching, we have two main design choices, namely table level caching and query result caching. Although table level caching [6, 26, 72] is more efficient in that it can answer arbitrary queries on cached tables, query result caching [119] is much simpler and can save expensive computations on cache hits. Thus, we chose to use query result caching in DotSlash.

6.2.3 Application Model

We consider the standard three-tier web architecture, shown in Figure 6.1. Application programs running at the application server access application data stored in the database server through a data driver, which is normally a system component of the application server. The data driver provides a standard API for web applications to store and retrieve data in the back-end database. In our prototype system, we use the common LAMP (Linux, Apache, MySQL, and PHP) configuration, where the PHP module resides in the Apache web server.

6.2.4 System Architecture

DotSlash rescue services allow an origin server to draft and release rescue servers fully automatically based on its load conditions. An origin web server discovers suitable

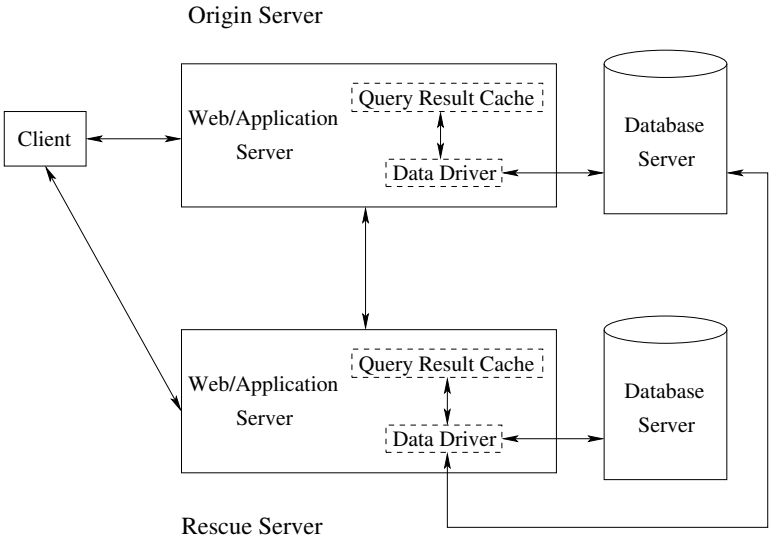


Figure 6.2: Enabling query result caching in DotSlash

rescue servers via wide-area service location, either among peer servers or from a dedicated pool of rescue servers, allocates them for temporary use, and redirects client requests to them. DotSlash uses DNS round robin as the first level crude load distribution, and uses HTTP redirect as the second level fine-grained load balancing. When a rescue relationship is set up between two web servers, the rescue server assigns a unique virtual host name to the origin server, which is used by the origin server in its HTTP redirects to the rescue server. Also, the origin server adds the rescue server's IP address to its local DNS for round robin. In DotSlash, a rescue server can serve the content of its origin server on the fly. In addition to caching static content from the origin server, a rescue server replicates PHP scripts dynamically from the origin server, and accesses databases at the origin server.

DotSlash Qcache services allow an origin server and its rescue servers to use on-demand query result caching to reduce the database workload at the origin server. Since the data driver (as shown in Figure 6.1) intercepts all database queries, we enhance it with query result caching without changing the application API and da-

Table 6.1: Three configurations in using DotSlash

Configuration	Bottlenecks Addressed	Used By
Dots_Apache	Network and web server	All sites
Dots_Apache + Dots_PHP	Network, web server, and application server	Dynamic sites
Dots_Apache + Dots_PHP + Dots_MySQL	Network, web server, application server, and database server	Dynamic sites

tabase interface. In our prototype system, we extend the original PHP data driver for MySQL databases with a query result cache. Figure 6.2 illustrates how to enable query result caching in DotSlash. Note that a client request can be redirected from the origin server to the rescue server via either DNS round robin or HTTP redirect. Also note that a rescue server may need to access remote databases at the origin server in addition to its local databases. We will discuss DotSlash data driver control in details in Section 6.2.6.

Our open-source prototype implementation of DotSlash [144] has three major components, namely Dots_Apache, Dots_PHP, and Dots_MySQL. *Dots_Apache* is an Apache module that supports basic DotSlash functions including workload monitoring, rescue server discovery, rescue relationship management, request redirection, dynamic virtual hosting, and dynamic DNS update. *Dots_PHP* is an extension for the PHP module of Apache that supports replicating PHP scripts dynamically. *Dots_MySQL* is a caching-enhanced PHP data driver for MySQL databases that supports caching database query results on demand. DotSlash can be used in three different configurations as shown in Table 6.1, where Dots_Apache and Dots_PHP provide DotSlash rescue services, and Dots_MySQL provides DotSlash Qcache services.

6.2.5 Caching Features

On-demand query result caching is a unique feature of DotSlash Qcache services: caching remains inactive as long as the load is normal, but is activated once the load is heavy. The control of our on-demand query result caching is based on two factors, namely the web server's DotSlash state and load region. A web server has three DotSlash states: *SOS state* if it gets rescue services from others, *rescue state* if it provides rescue services to others, and *normal state* otherwise. DotSlash uses two configurable parameters, lower threshold ρ^l and upper threshold ρ^u, to define three load regions: light load region $[0, \rho^l)$, desired load region $[\rho^l, \rho^u]$, and heavy load region $(\rho^u, 100\%]$. DotSlash measures utilization of multiple resources, e.g., our current prototype system measures network and CPU utilization. A web server's load region is determined as follows: the server is in the heavy load region if *any* resource is heavily loaded, in the light load region if *all* resources are lightly loaded, and in the desired load region otherwise.

We show the control of our on-demand query result caching in Figure 6.3. Caching is activated if a web server is in the SOS state (i.e., an origin server), or if a web server is in the rescue state (i.e., a rescue server), or if a web server is in the normal state and its load is above the upper threshold. On the other hand, caching is de-activated when an origin server switches from the SOS state to the normal state, or when a rescue server switches from the rescue state to the normal state, or when a web server is in the normal state and its load is below the lower threshold.

Self-configuration is an important feature of our system. When an origin server sets up its rescue servers, it passes the query result caching control parameters to its rescue servers. By doing so, a rescue server can manage cached objects based on the instructions from the origin server. In this way, an origin server can set up a

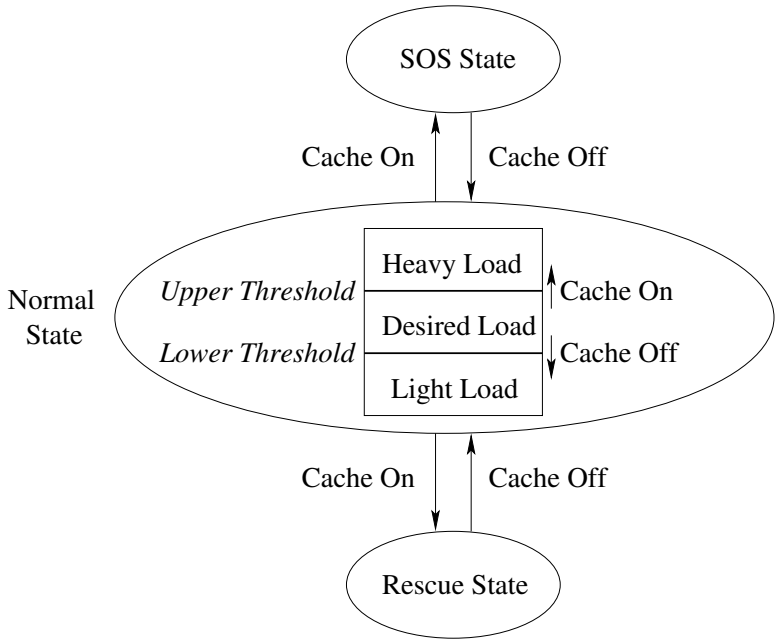


Figure 6.3: DotSlash on-demand query result caching, where caching is activated (cache on) or de-activated (cache off) based on the web server's DotSlash state (normal, SOS, or rescue) and load region (desired load, heavy load, or light load).

distributed query result caching system on the fly using one set of control parameters.

Distributed caching is a natural feature of our system. Each web/application server has its own, co-located query result cache by default. An origin server can obtain more query result caches as it drafts more rescue servers. Using co-located query result caches is well-suited for DotSlash in terms of resource utilization efficiency because our query result caching is on demand, and the cache server is idle most of time. Note that our system can use a dedicated query result cache server which is shared among an origin server and its rescue servers, or among a subset of rescue servers. Doing so can reduce the workload at the origin database server. However, a shared cache may become a potential performance bottleneck, and accessing a remote cache incurs longer delays (see Section 6.3.4 for experimental results).

Our query result caching is transparent to web users and applications. Without

the need to change client-side web browsers and server-side application programs, our system is easy to deploy. Furthermore, we provide a way for web users to bypass our query result caching. Our current prototype system uses the HTTP *Cache-Control* header for this purpose as follows. If there is *no-cache* or *max-age=0* in the HTTP Cache-Control header of a client request, DotSlash will handle that request without using query result caching.

6.2.6 Data Driver

Our caching-enhanced data driver is designed with the following considerations. First, rescue servers only handle read-only database queries; all write database queries are handled by the origin web server. This is mainly for security reasons because an origin server is unlikely to allow rescue servers to update its databases. Secondly, under heavy load we turn off write queries temporarily for regular users, but still allow site administrators (or a small group of premium users) to perform necessary updates. This is mainly for scalability considerations because database systems often use locking (e.g., table locking in MySQL) to control concurrent read/write accesses to the same database table, and a large number of read/write contentions can seriously degrade the database performance (see Section 6.3.5 for experimental results). Finally, under heavy load we provide different consistency guarantees for site administrators and regular users. The former can continue to perform both read and write queries, and get an up-to-date view of database states without using query result caching, whereas the latter can only perform read-only queries, and get a delayed view of database states by using query result caching. We use an application-specific caching TTL to bound the staleness of cached objects. Note that this design targets hotspot rescue for read-mostly databases, which are common for content management systems

(CMS), blogs, and web forums. It does not aim to be applicable to all web applications, e.g., it should not be used by e-commerce sites (modeled by benchmarks such as RUBiS [112] and TPC-W [128]) that have frequent updates and strong consistency requirements.

Our caching-enhanced data driver handles database queries based on three factors, namely the web server's query result caching state, the client request HTTP Cache-Control header, and the client request type. Our query result caching is a per-server state, which is on or off as illustrated in Figure 6.3. A client request can bypass our query result caching using the HTTP Cache-Control header as described in Section 6.2.5. A rescue server distinguishes two types of client requests, *regular* and *rescue*, based on the request's HTTP *Host* header. If the HTTP Host header uses an origin server name such as *www.origin.com*, or an assigned virtual host name such as *vh1.www.rescue.com*, then the request is treated as a rescue request, otherwise as a regular request.

We show the control of our caching-enhanced data driver in Table 6.2. There are four cases. For case 1, query result caching is off. Then any database query is handled normally by forwarding the query directly to the database. For case 2, query result caching is on and caching is not bypassed. Then any write query (i.e., the SQL *insert*, *update*, or *delete* statement) is turned off, and an error message, such as "Due to heavy load, write operations to databases at web site *http://www.origin.com* have been temporarily turned off", is returned. At the same time, any read-only query (i.e., the SQL *select* statement) is handled as follows. The query is checked against the query result cache. If there is a cache hit, the query result is obtained from the cache and returned to the application immediately. In case of a cache miss, the query is submitted to the corresponding database, which can be a local database or a remote database at the origin server; then the query result is obtained from

Table 6.2: DotSlash caching-enhanced data driver, which handles database queries based on the web server's query result caching state (on or off), the client request HTTP Cache-Control header (bypass caching or not), and the client request type (rescue or regular).

Case	Cache On	Bypass Caching	Rescue Request	Database Write	Database Read
1	no	–	–	normal	normal
2	yes	no	–	turn off	cache+DB
3	yes	yes	no	normal	DB+cache
4	yes	yes	yes	redirect	redirect

the database, saved to the query result cache, and returned to the application. For case 3, query result caching is on, caching is bypassed, and the request is a regular request. Then any database query is forwarded directly to the database. For a read-only query, the query result is saved to the query result cache before being returned to the application. For case 4, query result caching is on, caching is bypassed, and the request is a rescue request. Then the request is redirected back to the origin web server via HTTP redirect, which ensures that a client request that needs to bypass caching can always be handled by the origin web server. For this purpose, an origin server does not apply HTTP redirect to client requests that need to bypass caching. However, client requests could be distributed to rescue servers due to the origin server's DNS round robin. This is why we need to use HTTP redirect in case 4. Note that a rescue server uses the origin server's IP address in its HTTP redirects to bypass the origin server's DNS round robin mechanism.

6.2.7 Query Result Cache

We keep the query result cache as a separate component from the data driver. The advantage of doing this is that we can experiment and use different engines as our

caching storage.

The data driver uses the query result cache via two interface functions: *check_in* and *check_out*. The *check_in* function takes the query string, query result, and caching TTL as input parameters, serializes the query result into a byte stream, and saves it to the caching storage engine. The *check_out* function takes the query string as the input parameter and retrieves the query result. For a cache hit, it de-serializes the query result byte stream into the original query result data structure and returns a pointer to the result structure. In case of a cache miss, it returns a NULL pointer.

Both disk and memory can be used as our caching storage engine. Due to performance considerations, we choose to use a memory storage engine, *memcached* [80], which employs a client-server model. At the server side, a daemon maintains cached objects in dynamically allocated memory. Each cached object is a key-value pair with an expiration time. At the client side, we use an open-source C library *libmemcache* [78] to access the cache. In the *check_in* function, we first use the ELF hash algorithm [22] to map the query string into a cache key, and then store the query string and the query result as the cache value, using the caching TTL as the expiration time. Note that different query strings might be mapped into the same cache key with a small probability, which is less than 1% in our experiments. To handle this type of hash conflicts, we let the new query and its result overwrite the old one. This strategy keeps our system simple without losing much performance. In the *check_out* function, we use the same ELF hash algorithm to map the query string into a cache key. If a cached object is found for the key, we check whether the stored query string matches the input query string. If so, it is a cache hit; otherwise, it is a cache miss.

6.3 Evaluation

We use the maximum request rate supported by a web site as the major performance metric. Our goal is to show the performance differences in four cases, namely without using DotSlash, using DotSlash rescue services only, using DotSlash Qcache services only, and using DotSlash rescue and Qcache services together.

6.3.1 Benchmark Description

We evaluate our prototype system using the RUBBoS bulletin board benchmark [111], which is modeled after an online news forum like Slashdot [120], and has been used in a number of systems [119, 90, 151].

RUBBoS supports discussion threads. Each thread has a story at its root, and a number of comments for that story, which may be nested. There are two types of users in RUBBoS: regular users who browse and submit stories and comments, and moderators who in addition review stories and rate comments. The PHP version of RUBBoS consists of 19 PHP scripts, and the size of script files varies between 1 and 7 KB. The database has a size of 439 MB, and contains $500,000$ users and 2 years of stories and comments. There are 15 to 25 stories per day, and 20 to 50 comments per story. The length of story and comment bodies is between 1 and 8 KB.

We use RUBBoS clients to generate workloads. Each RUBBoS client can emulate a few hundred HTTP clients. An HTTP client issues a sequence of requests using a think time that follows a negative exponential distribution, with an average of 7 seconds [128]. If the request rate to be generated is high, multiple RUBBoS clients are used, each running on a separate machine. We use 7 seconds [29] as the timeout value for getting the response for a request. If more than 10% [29] of issued requests time out, the web server is considered as being overloaded.

RUBBoS has two major workload mixes, read-only and submission. The read-only mix invokes browse scripts, story/comment view scripts, and search scripts with a probability of 2/3, 1/6, and 1/6, respectively. The submission mix invokes update scripts with a probability of 1/10. The update scripts have both read and write database queries. As a result, 2% of the total database queries in the submission mix are write queries. A special property of the RUBBoS workload mixes is that for the same request rate, its read-only mix causes a higher workload at the database than its submission mix. This is due to two reasons. First, each pre-generated story has 20 to 50 comments, whereas a newly submitted story has only a few comments or no comments at all. Secondly, each emulated RUBBoS client always starts with, and often returns to the *Stories Of The Day* page, which has the most recent 10 stories.

6.3.2 Experimental Setup

Since we use query result caching to address the database server CPU bottleneck, we performed experiments in our local area network. We used a cluster of Linux machines connected via 100 Mb/s fast Ethernet. These machines had three different configurations. Each web/application server had a 3 GHz Intel Pentium 4 CPU and 2 GB of memory, running Red Hat Enterprise Linux AS v.3 with Linux kernel 2.4.21-32.0.1.EL. The database server had a 2 GHz AMD Athlon XP CPU and 1 GB of memory, running Red Hat 9.0 with Linux kernel 2.4.20-20.9. Each client emulator machine had a 1 GHz Intel Pentium III CPU and 512 MB of memory, running Red Hat 9.0 with Linux kernel 2.4.20-20.9.

We ran a varying number of web/application servers in different experiments. All web/application servers ran Apache 2.0.49, configured with PHP 4.3.6, *worker* multi-processing module, proxy modules, cache modules, and our DotSlash module

Dots_Apache. The PHP module incorporated our *Dots_PHP* and *Dots_MySQL*, where *Dots_PHP* is our DotSlash extension for PHP to support dynamic script replication, and *Dots_MySQL* is our caching-enhanced data driver for MySQL databases to support query result caching. By default, each web/application server had a co-located cache server running *memcached* with a storage space limit of 200 MB. When a cache server was shared among several web/application servers, it had a storage space limit of 1 GB.

The database server ran MySQL 4.0.18. Based on our evaluation, the default MySQL storage engine MyISAM delivered a better performance than the InnoDB storage engine for our chosen benchmark RUBBoS. Thus, we used MyISAM tables in all our experiments. To enhance the performance of MyISAM tables under heavy updates, we configured MySQL with *delay_key_write=all* to delay the writing of index data to disk [142]. To support a large number of concurrent connections, we configured MySQL with *open_files_limit=65535* and *max_connections=8192*. MySQL has a warm-up stage to load the table index information into memory. To obtain consistent results in the steady state of MySQL, we restarted MySQL after each run of our experiments, and warmed up MySQL before each experiment using the read-only mix with 1400 emulated clients. This workload caused the database server CPU to be loaded around 70%. After each run of the submission mix, the RUBBoS database was restored to its original content so that all experiments started with the same database content.

We used our *dot-slash.net* domain for dynamic DNS name registrations, and used the enhanced Service Location Protocol [145] for rescue server discovery.

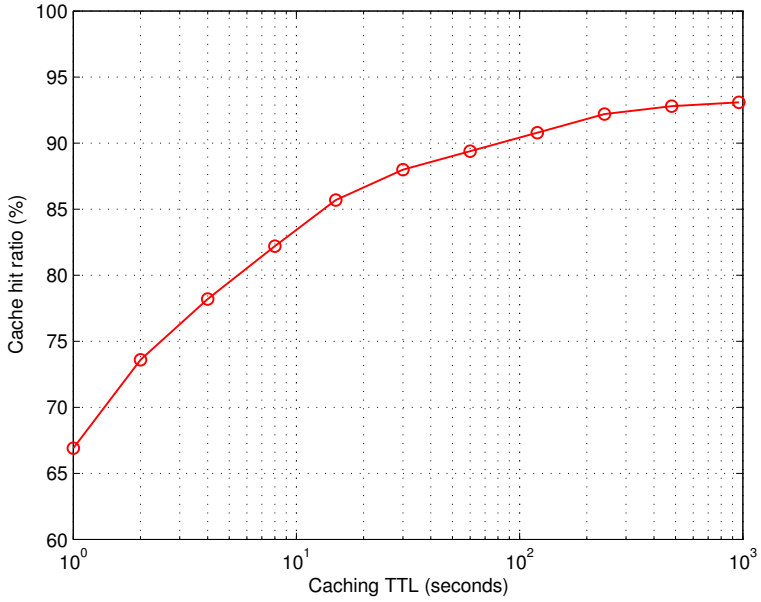


Figure 6.4: The relationship between the caching TTL and query result cache hit ratio in a set of 10-minute experiments for the RUBBoS read-only mix

6.3.3 Caching TTL

In DotSlash, each web server has a configurable parameter called caching TTL, which is used to control how long query results can be cached. This parameter is passed from an origin server to all its rescue servers; and a rescue server caches query results from an origin server based on the origin server's caching TTL parameter.

In general, the caching TTL for query results is an application-dependent parameter since different applications may need to use different caching TTLs based on their data consistency requirements. For RUBBoS, we use 60 seconds as the caching TTL because it is good enough to bound the staleness of cached objects in RUBBoS.

There is a trade-off in choosing the caching TTL parameter: decreasing this parameter will improve data consistency, whereas increasing this parameter will improve caching performance. Figure 6.4 shows the relationship between the caching TTL and

query result cache hit ratio in a set of 10-minute experiments for the RUBBoS read-only mix. We observe that the cache hit ratio increases as the caching TTL increases. For our chosen caching TTL 60 seconds, the cache hit ratio is 89.4%.

6.3.4 Results for the RUBBoS Read-only Mix

We first tested our system using the RUBBoS read-only mix. Depending on whether rescue servers are available, whether query result caching is enabled, and whether each web/application server has a co-located cache or uses a shared cache server running on a separate machine, we have five test cases for the read-only mix as follows.

- $READ$: no rescue, no cache.
- $READ_c$: no rescue, with a co-located cache.
- $READ_r$: with rescue, no cache.
- $READ_{r,c}$: with rescue, with a co-located cache.
- $READ_{r,sc}$: with rescue, with a shared cache.

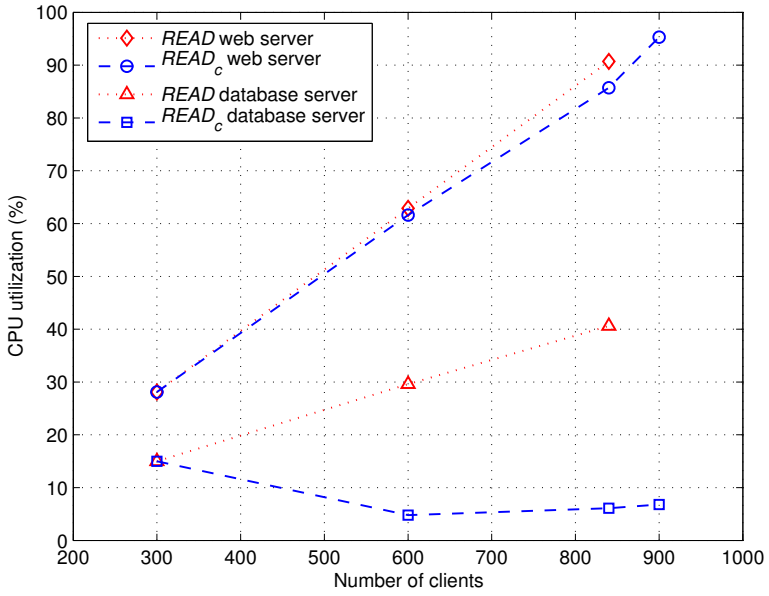
Table 6.3 summarizes our experimental results for the RUBBoS read-only mix. Without using DotSlash rescue and Qcache services, a web server can only support a request rate of 117 requests/second. The request rate supported increases to 249 requests/second by only using DotSlash rescue services with 4 rescue servers, and increases to 1151 requests/second by using DotSlash rescue and Qcache services together with 15 rescue servers. Compared to $READ$, $READ_r$ and $READ_{r,c}$ achieve an improvement of 213% and 984%, respectively. Compared to $READ_r$, $READ_{r,c}$ achieves an improvement of 462%. Next, we give details for each test case using the read-only mix.

Table 6.3: Summary of experimental results for the RUBBoS read-only mix

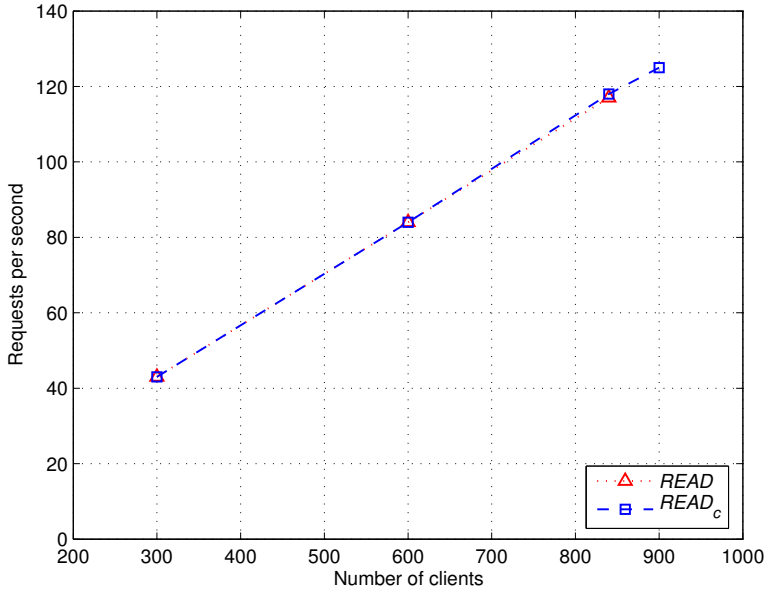
Test Case	Max Rate (reqs/sec)	Compared to $READ$	Compared to $READ_r$	Rescue Servers
$READ$	117	100%		
$READ_c$	125	107%		
$READ_r$	249	213%	100%	4
$READ_{r,c}$	1151	984%	462%	15
$READ_{r,sc}$	828	708%	333%	13

Figure 6.5 shows the experimental results for $READ$ and $READ_c$, where rescue servers are not available. We give the CPU utilization for the web server and database server in Figure 6.5(a), and present the request rate supported in Figure 6.5(b). We observe that the web server CPU is the bottleneck. When the load is light with 300 clients, caching is not activated. Thus, we have the same CPU utilization for $READ$ and $READ_c$. When the load is heavy with 840 clients, caching is turned on, and we can observe a big difference in CPU utilization. The database server CPU utilization is 41% in $READ$, but is only 6% in $READ_c$, meaning that caching is very effective in reducing the database workload. At the same time, the web server CPU utilization decreases from 91% to 86% by using caching, indicating that getting query results from the cache incurs less cost than accessing the database directly. The maximum request rate supported is 117 and 125 requests/second in $READ$ and $READ_c$, respectively. The cache hit ratio is 91% in $READ_c$. In summary, even without using rescue servers, query result caching is useful under heavy load. However, caching itself cannot remove the web server bottleneck.

Figure 6.6 shows the experimental results for $READ_r$, $READ_{r,c}$, and $READ_{r,sc}$, where a varying number of rescue servers are used. By using a sufficient number of rescue servers, the origin web server is no longer a bottleneck. We give the CPU



(a) The CPU utilization for the web server and database server



(b) The request rate supported

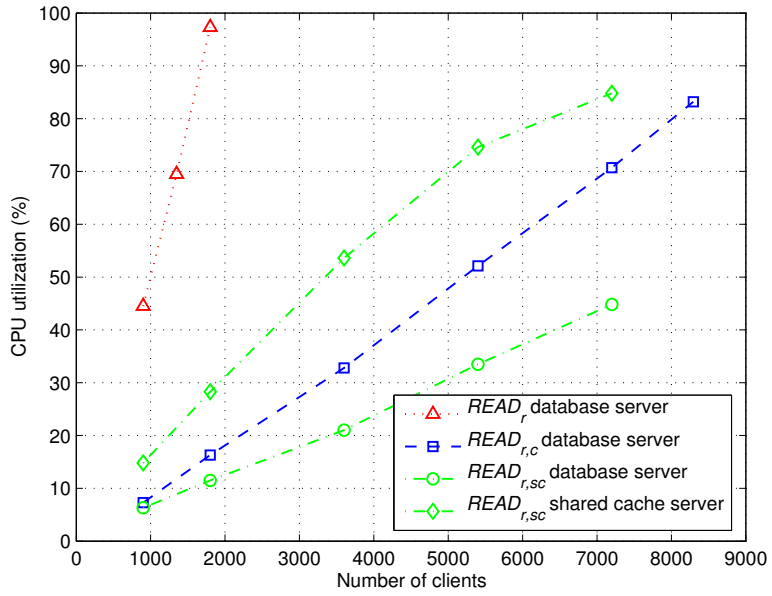
Figure 6.5: Experimental results for the RUBBoS read-only mix when rescue servers are not available

utilization for the origin database server and the shared cache server used in $READ_{r,sc}$ in Figure 6.6(a), present the request rate supported in Figure 6.6(b), and display the average response time in Figure 6.6(c).

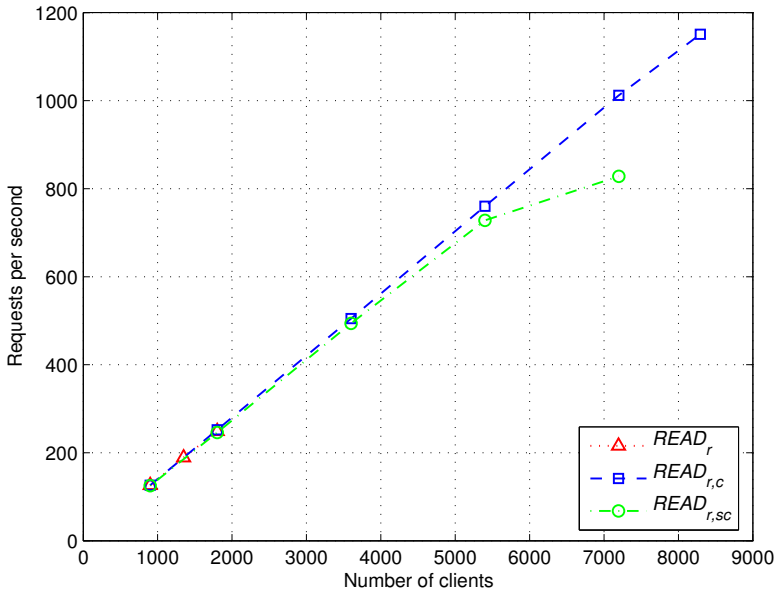
For $READ_r$, the origin database server gets overloaded quickly without using query result caching. The maximum request rate supported is 249 requests/second, obtained using 1800 clients and 4 rescue servers. Under this load, the origin database server CPU utilization is 97%.

For $READ_{r,c}$, each web/application server uses a co-located query result cache, which greatly reduces the database workload. For 1800 clients, the origin web server uses 4 rescue servers, and the measured request rate is 252 requests/second. Under this load, the origin database server CPU utilization is only 16%, which is a huge reduction compared to 97% CPU utilization in $READ_r$. The maximum request rate supported is 1151 requests/second, obtained using 8295 clients and 15 rescue servers. Under this load, the origin database server CPU utilization is 83%, and the origin web server cache hit ratio is 87%. For an experiment of this scale with 8295 clients, we use 38 machines: 21 for emulating clients, 15 as rescue servers, 1 as the origin web server, and 1 as the origin database server.

For $READ_{r,sc}$, all web/application servers use a shared query result cache server running on a separate machine, which can further reduce the database workload. For 5400 clients, the origin database server CPU utilization is only 34%, compared to 52% CPU utilization in $READ_{r,c}$. However, the shared cache server itself becomes a bottleneck since it gets loaded more quickly than the origin database server does. The maximum request rate supported is 828 requests/second, obtained using 7200 clients and 13 rescue servers. Under this load, the CPU utilization for the origin database server and the shared cache server is 45% and 85%, respectively, and the cache hit ratio at the shared cache server is 93%. In Figure 6.6(c), the average response time

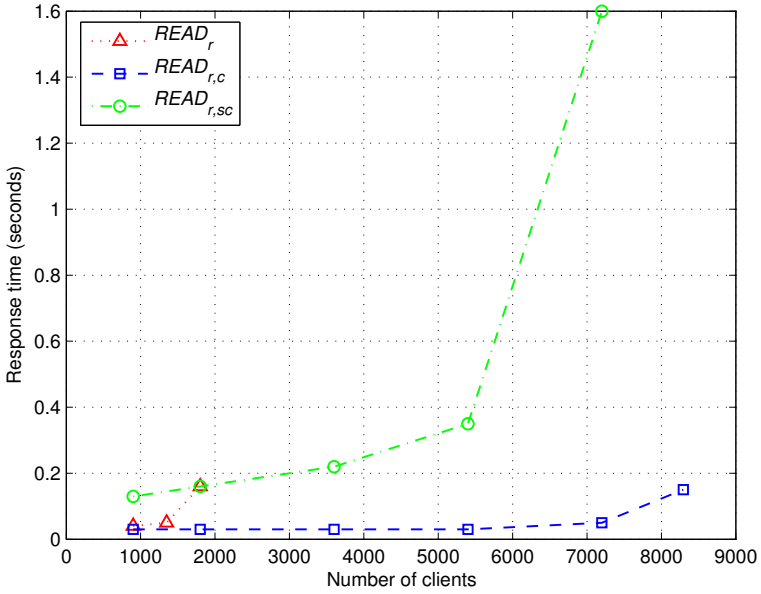


(a) The CPU utilization for the origin database server and the shared cache server used in $READ_{r,sc}$



(b) The request rate supported

Figure 6.6: Experimental results for the RUBBoS read-only mix when rescue servers are available



(c) The average response time

Figure 6.6: Experimental results for the RUBBoS read-only mix when rescue servers are available (Continued)

in $READ_{r,sc}$ is much longer than that in $READ_{r,c}$ since using a shared cache incurs longer delays for remote cache accesses. In general, a shared cache is a single point of failure and a potential performance bottleneck, and it incurs longer delays. Note that it is possible to divide rescue servers into groups, and use a separate shared cache in each group, which has the potential to keep the shared cache in each group from being overloaded, and reduce the database workload as much as possible. However, this method incurs administrative overhead in forming groups and determining the right size of each group. As our goal is to build an autonomic system, we will not explore this approach in more depth.

In summary, by using DotSlash rescue and Qcache services together, a web site can improve its maximum request rate supported by a factor of 10 for the RUBBoS read-only mix. Although the major performance gain comes from the Qcache services,

the rescue services are the fundamental framework upon which the Qcache services are built. Moreover, the efficiency of the Qcache services depends on the cache hit ratio.

6.3.5 Results for the RUBBoS Submission Mix

Based on Section 6.2.6, DotSlash turns off database write queries temporarily for regular users under heavy load. We disable this feature in testing our system against the RUBBoS submission mix, which has about 2% write queries. We choose to do so for two reasons. First, turning off all write queries will convert the submission mix into a read-only mix, which we have evaluated in the last section. Secondly, allowing site administrators to perform necessary updates in our system is roughly equivalent to having a small percentage of write queries in the submission mix. Depending on whether rescue servers are available and whether query result caching is enabled, we have four test cases for the submission mix as follows.

- SUB: no rescue, no cache.
- SUB_c: no rescue, with cache.
- SUB_r: with rescue, no cache.
- $SUB_{r,c}$: with rescue, with cache.

Table 6.4 summarizes our experimental results for the RUBBoS submission mix. Without using DotSlash rescue and Qcache services, a web server can only support a request rate of 180 requests/second. The request rate supported increases to 580 requests/second by only using DotSlash rescue services with 4 rescue servers, and increases to 871 requests/second by using DotSlash rescue and Qcache services together with 8 rescue servers. Compared to SUB, SUB_r and $SUB_{r,c}$ achieve an improvement

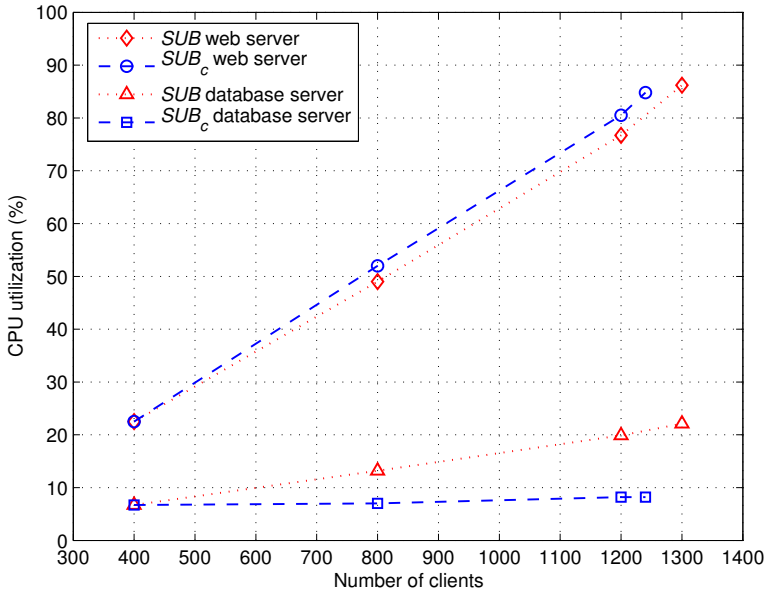
Table 6.4: Summary of experimental results for the RUBBoS submission mix

Test Case	Max Rate (reqs/sec)	Compared to SUB	Compared to SUB_r	Rescue Servers
SUB	180	100%		
SUB_c	174	97%		
SUB_r	580	322%	100%	4
$SUB_{r,c}$	871	484%	150%	8

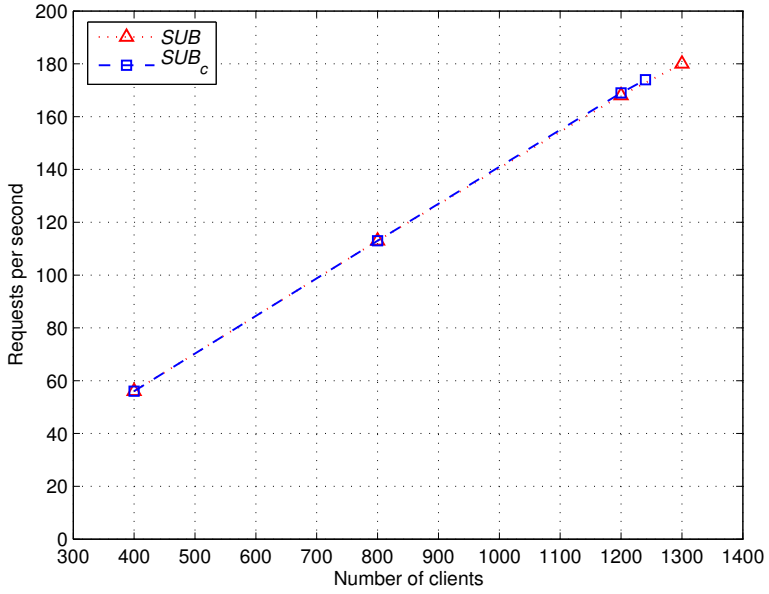
of 322% and 484%, respectively. Compared to SUB_r, $SUB_{r,c}$ achieves an improvement of 150%. Next, we give details for each test case using the submission mix.

Figure 6.7 shows the experimental results for SUB and SUB_c, where rescue servers are not available. We give the CPU utilization for the web server and database server in Figure 6.7(a), and present the request rate supported in Figure 6.7(b). We observe that the web server CPU is the bottleneck. When the load is light with 400 clients, caching is not activated. Thus, we have the same CPU utilization for SUB and SUB_c. When the load is heavy with 1200 clients, caching is turned on. However, the performance is not improved by only using query result caching because it reduces the database workload but increases the web server workload due to a low cache hit ratio, and the web server is the bottleneck. The maximum request rate supported is 180 and 174 requests/second in SUB and SUB_c, respectively. Note that the number of clients supported is 1300 in SUB and 1240 in SUB_c. The cache hit ratio is 76% in SUB_c, which is much lower compared to around 90% cache hit ratio in the RUBBoS read-only mix.

Figure 6.8 shows the experimental results for SUB_r and $SUB_{r,c}$, where a varying number of rescue servers are used. By using a sufficient number of rescue servers, the origin web server is no longer a bottleneck. We give the origin database server CPU utilization in Figure 6.8(a), present the request rate supported in Figure 6.8(b), and



(a) The CPU utilization for the web server and database server



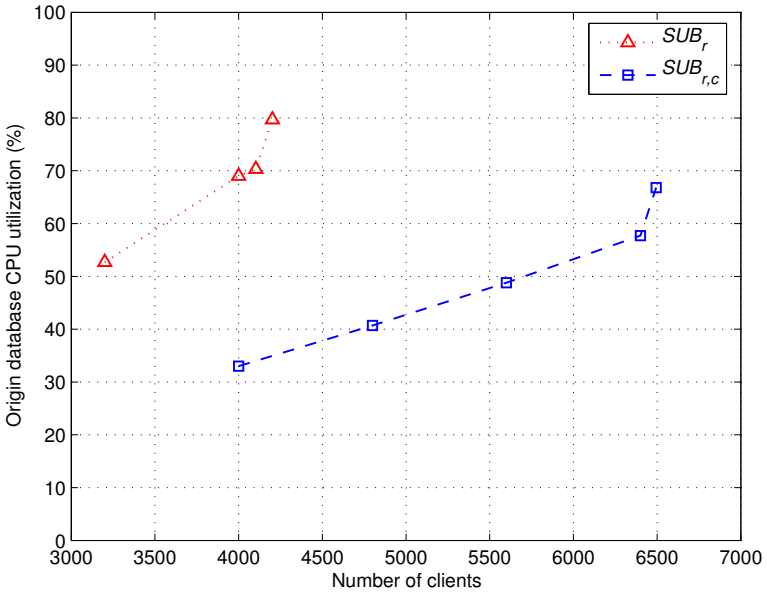
(b) The request rate supported

Figure 6.7: Experimental results for the RUBBoS submission mix when rescue servers are not available

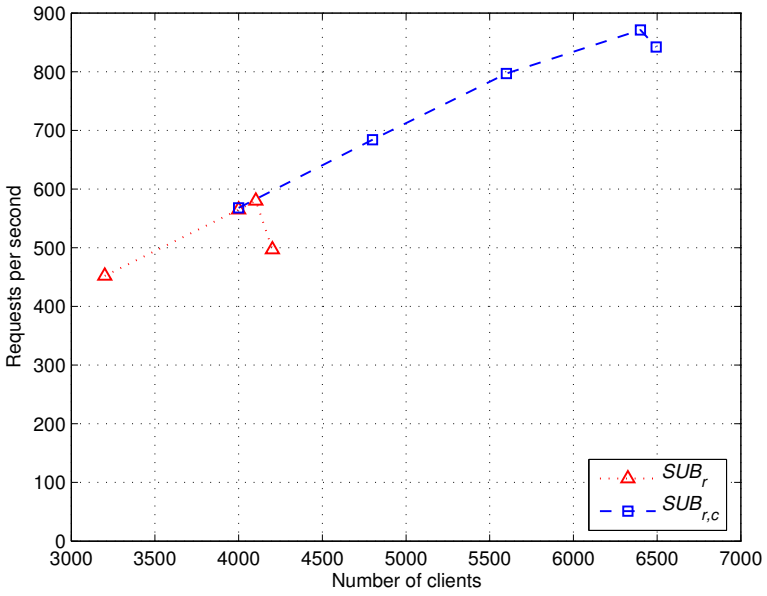
display the rate of locks waited for at the origin database server in Figure 6.8(c).

Based on Figure 6.8(a) and 6.8(b), we observe that the origin database server CPU utilization at the peak rate is only 58% and 70% in $SUB_{r,c}$ and SUB_r, respectively, which are much lower compared to over 80% CPU utilization in the RUBBoS read-only mix. This leads us to locate other bottlenecks in the system besides the database CPU utilization. In fact, for the RUBBoS submission mix, the rate of database locks waited for becomes a performance bottleneck well before the database CPU gets overloaded. MySQL uses table locking in its default storage engine MyISAM to control concurrent read/write accesses to the same database table. Table locking allows many threads to read from a table at the same time; but a thread must get an exclusive write lock to write to a table. During an update to a database table, all other threads that need to access this particular table must wait until the update is done. In MySQL, the number of table access contentions caused by table locking is indicated by a status variable called *table_locks_waited*. In the RUBBoS submission mix, both read and write access rates go up as the number of clients increases. As a result, the rate of locks waited for increases. At certain point, the number of table access contentions increases abruptly, which causes the database performance to degrade seriously. Using query result caching reduces the read access rate to the origin database, which in turn reduces the number of table access contentions as well as the database workload.

For SUB_r, query result caching is not used. As the load increases, the read access rate to the origin database increases quickly along with the write access rate. The maximum request rate supported is 580 requests/second, obtained using 4103 clients and 4 rescue servers. Under this load, the origin database server has a 70% CPU utilization, and an average of 4 locks waited for per second.

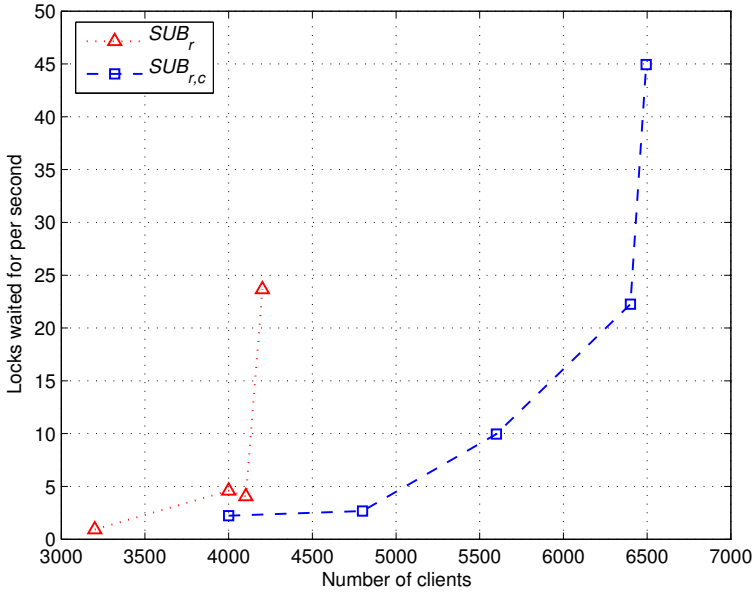


(a) The origin database server CPU utilization



(b) The request rate supported

Figure 6.8: Experimental results for the RUBBoS submission mix when rescue servers are available



(c) The rate of locks waited for at the origin database server

Figure 6.8: Experimental results for the RUBBoS submission mix when rescue servers are available (Continued)

For $SUB_{r,c}$, each web/application server uses a co-located query result cache, which greatly reduces the read access rate to the origin database. The maximum request rate supported is 871 requests/second, obtained using 6400 clients and 8 rescue servers. Under this load, the origin database server has a 58% CPU utilization, and an average of 22 locks waited for per second. The origin web server cache hit ratio is 70%.

In summary, by using DotSlash rescue and Qcache services together, a web site can improve its maximum request rate supported by a factor of 5 for the RUBBoS submission mix, where the major performance gain comes from the rescue services. Comparing this performance improvement with that of the read-only mix, we observe a difference of a factor of 2. Write queries not only reduce the cache hit ratio, but also increase database access contentions. To allow single-server databases to survive

web hotspots, DotSlash turns off write queries temporarily for regular users under heavy load.

6.4 Related Work

Dynamic scalability for databases is largely an open issue. Recently, Olston et al. [90] proposed a scalability service for databases using multicast-based consistency management, Although their system aims at broader web applications, its scalability gain under heavy load is unclear. Moreover, their service is not transparent since clients need to connect to their proxy servers in order to use the service. In contrast, DotSlash targets read-mostly databases and scales well under dramatic load spikes. Also, our system is transparent to web users. Amza et al. [8] evaluated transparent scaling techniques for dynamic content web sites. Their results show that query result caching can significantly increase performance for read-mostly databases, whereas content-aware scheduling is effective for write-intensive databases. The performance evaluation of our prototype system confirmed that query result caching works well for read-mostly databases.

Replication is a widely used mechanism for database scalability. Ganymed [100] separates update from read-only transactions, and routes updates to a main database server and queries to read-only database copies. GlobeDB [118] uses partially replicated databases based on data partition to reduce update traffic. Our current prototype uses a single back-end database server, which can be extended to support distributed database servers by incorporating database replication into our system.

Database caching [119, 6, 26, 72] can cache data from back-end databases at caches closer to application servers, which is very effective in reducing the workload at back-end databases. While caching in existing systems is active in all cases, our

query result caching is activated only under heavy load, which minimizes the effect of caching on consistency while improving the system scalability.

Database clustering [32, 91, 85] is a mechanism to pool database servers together so as to provide high availability and performance. While Oracle Real Application Clusters [91] uses a shared cache architecture, MySQL Cluster [85] is built on a shared-nothing architecture. Clustered JDBC [32] implements the Redundant Array of Inexpensive Databases concept, and provides a single virtual database to the application through the JDBC interface. Generally speaking, database clustering is a solution at the database server tier for high availability and performance. In contrast, DotSlash is a solution at the web/application server tier for dynamic scalability. Thus, our system and database clustering are orthogonal, and they can be used together at dynamic content web sites.

Load balancing among a set of geographically distributed database servers has been proposed. In the WARD architecture [103], dispatchers at an overloaded Internet Data Center (IDC) can redirect requests for dynamic content to a geographically remote IDC, that is to schedule database requests among replicated database servers for load balancing. DotSlash on-demand distributed query result caching is orthogonal to database load balancing, and they can be used together at dynamic content web sites.

6.5 Summary

This chapter described DotSlash Qcache services that allow a web site to use on-demand distributed query result caching. Our query result caching is self-configuring and transparent to web users and applications. Through our experimental results, we have demonstrated that using DotSlash rescue and Qcache services together is very

effective for read-mostly databases, e.g., a web site can improve its maximum request rate supported by a factor of 10 for the RUBBoS read-only mix.

Chapter 7

Web Traffic Prediction for Overload Prevention

This chapter describes a prediction algorithm for estimating the upper bound of future web traffic volume [148]. It provides insight into characterizing traffic of web hotspots, and is useful for web server overload prevention systems such as DotSlash. In this chapter, we first introduce the problem of web traffic prediction for overload prevention and discuss related work. Then, we describe the design motivation and our prediction algorithm. After discussing the algorithm parameter selection and presenting the experimental results, we give a summary.

7.1 Introduction

To provide expected quality of service, a web site needs to predict its future traffic volume: long-term prediction (e.g., in months or years) is useful for capacity planning, and short-term prediction (e.g., in seconds or minutes) is useful for overload prevention. As the web becomes popular and has a huge number of potential users,

a web site may receive highly bursty requests and get overwhelmed by web hotspots. Short-term traffic prediction is important for a web site to take needed actions in advance when it anticipates an approaching peak load which is likely to exceed its capacity or a preset threshold.

For overload prevention, we only need to predict an upper bound of the future traffic volume, i.e., whether the future traffic volume will exceed the server's capacity or a preset threshold. As long as the future traffic volume is below the predicted bound, the exact volume does not matter much here. Moreover, over-prediction has less penalty than under-prediction because a false alarm only incurs unnecessary overhead, but a missed prediction of excess traffic can cause the server to be overloaded.

To estimate the upper bound of future web traffic volume, we designed a prediction algorithm [148]. We employ a multiple-time-scale approach by using traffic information at a smaller time scale to forecast traffic volume at a larger time scale. Moreover, we utilize traffic statistical properties other than curve fitting to forecast future traffic volume. This algorithm is simple and effective in upper bound prediction for short-term bursty web traffic, which is useful for web hotspot rescue because a web server needs to take rescue actions early, such as setting up rescue servers, to shed load.

7.2 Related Work

Traffic characterization for web hotspots is an important issue. Adler [4] points out the Slashdot effect for three Internet publications; Schwarz [115] describes dramatic traffic surges at a US geological survey web site after earthquakes; Wessels [141] studies the effect of Ken Starr's report on NALAR web caches; Jung et al. [65] investigate and compare the properties of flash events and that of denial of service

(DoS) attacks; Barford et al. [18] analyze high volume traffic anomalies; and Arlitt et al. [12] study the workload of the 1998 World Cup web site. These studies show how traffic changes during web hotspots so as to properly identify web hotspots and predict overload conditions. Distinguishing web hotspots from other high volume traffic anomalies such as DoS attacks allows the web server to take different actions in different situations.

Properly measuring workload level and service quality is useful for web hotspot rescue. Banga and Druschel [16] study how to correctly measure the capacity of a web server using bursty traffic. The degradation of client-perceived service quality may suggest a web hotspot. Olshefski et al. [89] describe a mechanism for inferring client response time at the web server.

7.3 Motivation

Traditionally, traffic predictions [19] are carried out at a single time scale using curve fitting. A difficult issue here is to choose the right number (denoted as N_h) of history intervals used for predictions: if N_h is too large, then predictions are based upon less relevant information in history, whereas if N_h is too small, then predictions are made from incomplete information; both cases lead to poor predictions. Usually better predictions can be achieved by varying N_h dynamically, but it is hard to derive the right N_h.

To avoid the trouble of choosing N_h, we seek a new approach to traffic prediction by only using traffic information in current interval. More specifically, we try to correlate how traffic changes within the current interval at a smaller time scale (e.g., one tenth of the current time scale) with that at the current time scale. Previous work on self-similarity [40] has shown that statistical correlations exist for web traffic

at different time scales. From a first look, it seems that self-similarity is not useful for traffic predictions since it is a property for stationary processes, whereas predictions are more useful when traffic volume changes quickly and dramatically. A careful re-consideration reveals that no matter how quickly a traffic changes, at sufficiently small time scales, the change between adjacent intervals will be small. Equivalently, we can regard the mean of traffic volume in adjacent intervals as unchanged and the real change as variability. For example, for three consecutive intervals, I_1, I_2, and I_3, we can view that I_1 and I_2 have the same mean $\mu_{1,2}$, and I_2 and I_3 have the same mean $\mu_{2,3}$; but $\mu_{1,2}$ and $\mu_{2,3}$ can be unequal (i.e., I_1 and I_3 can have different means). In this way, we can perform predictions by utilizing self-similarity within two adjacent intervals at sufficiently small time scales. A good fit here is that self-similarity is measured in terms of statistical correlations between two different time scales, which are just what we need to predict the upper bound of future traffic volume.

7.4 Prediction Algorithm

We formulate our prediction problem as follows: given a time scale T (such as 100 seconds), we want to predict the upper bound of traffic volume in the next interval based on traffic information in current interval. Note that the length of each interval is T. Let v_c and v_{c+1} denote the traffic volume in current interval (I_c) and next interval (I_{c+1}), respectively, and d_c denote the difference between v_{c+1} and v_c (i.e., $d_c = v_{c+1} - v_c$). If we can find an upper bound (denoted as b_c) of d_c, then we can estimate that $v_{c+1} < v_c + b_c$. In other words, predicting an upper bound for v_{c+1} is equivalent to estimating b_c. Next, we show how to use statistical properties and self-similarity to estimate b_c.

Let the random variable $D(T)$ denote the difference of traffic volume between

adjacent intervals at time scale T, and $\mu(T)$ and $\sigma(T)$ denote the mean and standard deviation of $D(T)$, respectively. If we assume that $D(T)$ follows a normal distribution, we can estimate the bound of $D(T)$ using $\mu(T)$ and $\sigma(T)$. For example, since about 95% of the samples of $D(T)$ fall into the range of $[\mu(T) - 2\sigma(T), \mu(T) + 2\sigma(T)]$, we can say that a sample of $D(T)$ will be less than $\mu(T) + 2\sigma(T)$ with more than 95% probability. In order to derive b_c, we divide I_c into n equal sub-intervals with length of $T' = T/n$, and look at $D(T')$ in these n sub-intervals. With a sufficient number of samples (e.g., $n \geq 10$), we can have an estimate for $\mu(T')$ and $\sigma(T')$. If assuming that the traffic is self-similar with Hurst parameter H within the period of I_c and I_{c+1}, then we have $\mu(T) = n^H \mu(T')$, and $\sigma(T) = n^H \sigma(T')$. With $\mu(T)$ and $\sigma(T)$, we can estimate b_c as $\mu(T) + 2\sigma(T)$. Note that here we choose $\mu(T) + 2\sigma(T)$ rather than $\mu(T) + 3\sigma(T)$ mainly because we want to have a closer upper bound estimation to avoid unnecessary false alarms. Also note that since our prediction is based on statistical properties, the predicted upper bound is correct only with a high probability.

7.5 Parameter Selection

Several parameters affect the prediction performance. The first one is the prediction interval T. As we use self-similarity to derive statistical correlations between two different time scales, the mean of traffic volume should be roughly unchanged within the period of $2T$. Thus, our prediction algorithm has an inherent restriction on how far we can predict into the future. Based on our experimental results described in Section 7.6, usually T should not exceed 100 seconds. The second parameter is the scaling factor n between the two different time scales T and T'. As this parameter determines the number of samples in time scale T' that are used for deriving statistical properties,

n should be no less than 10. The third one is the Hurst parameter H. Since we do not know the correct H in advance, and using a larger H tends to overestimate whereas using a smaller H tends to underestimate, the general guidances are as follows: (1) the burstier the traffic, the larger H [75]; (2) a right H will result in roughly the same prediction performance when n changes; and (3) use a little bit larger H if not sure, usually in the range of $[0.8, 0.9]$.

7.6 Experimental Results

To evaluate our prediction algorithm, we applied it to the 1998 World Cup data set [12], which included 1.35 billion requests made to 30 servers at four different regions during a period of 92 days. We ran our algorithm for three servers on three days. The three chosen servers were server5, server41, and server64, which were selected from three different regions since servers in the same region had very similar traffic curves. The three chosen days were June 29 (day65), July 7 (day73) and July 8 (day74), which were among the busiest days in the data set. In each day, we chose a period of three hours that included a dramatic traffic spike.

We carried out our experiments in three steps. For preparation, we calculated the number of requests at the following time scales: 1, 2, 5, 10, 12, 15, 18, 20, 30, 40, 50, 60, 100, 120, 150, 180, 200, 300, 400, 600 seconds. In different experiments, we varied T, n and H to evaluate how they affect predictions. After each experiment, we calculated the percentage of prediction intervals in which the real traffic volume falls below the predicted upper bound.

In the first experiment, we fixed $n = 10$ and $H = 0.85$, but varied T from 10 to 600 seconds. We got consistent prediction performance across all nine different server-day combinations. For clarity, we only show the results for the three servers

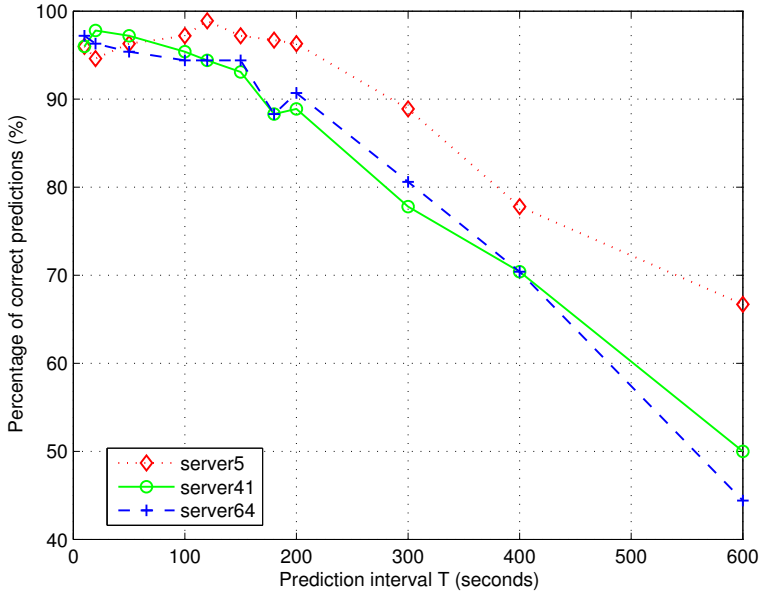


Figure 7.1: Prediction results for day74 of the 1998 World Cup data set, where $n = 10$, $H = 0.85$, and the percentage of correct predictions is computed as the percentage of prediction intervals in which the real traffic volume falls below the predicted upper bound.

on day74 in Figure 7.1. As we anticipated, the prediction performance changes as T increases: stays around 95% when $T \leq 100$ seconds, degrades slowly to around 90% when $T \in (100, 200)$ seconds, and falls down quickly when $T \geq 200$ seconds.

In Figure 7.2, we give the detailed prediction results for server41 on day65, where $T = 100$ seconds, $n = 10$, and $H = 0.85$. For ease of comparison, we show real traffic volume, predicted upper bound of increase, and real change altogether in this figure, from which we can draw two conclusions as follows. First, the real change is below the predicted upper bound of increase in most cases (more than 95%). Secondly, the predicted upper bound of increase is relatively small compared with the real traffic volume. In other words, our predictions can bound the increases with a high probability of correctness without overestimating the increases too much. In the

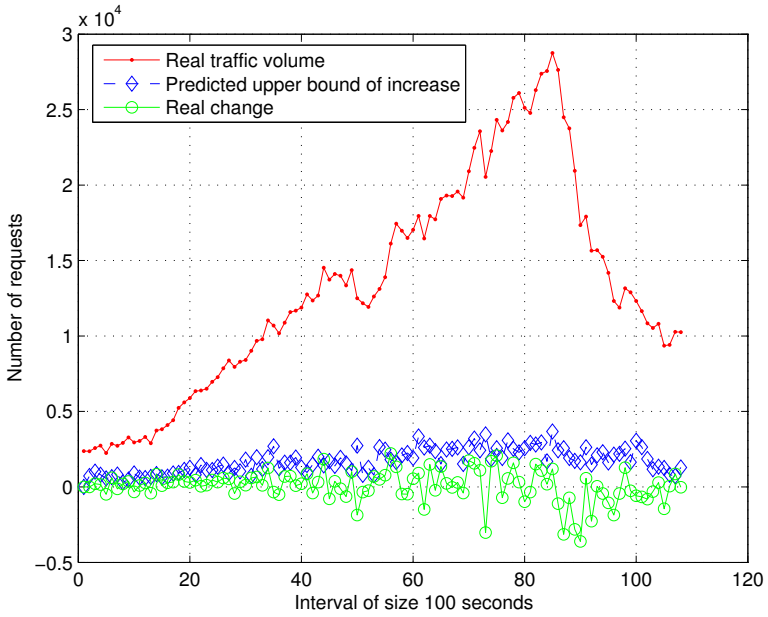


Figure 7.2: Detailed prediction results for server41 on day65 of the 1998 World Cup data set, where $n = 10$, $H = 0.85$, and $T = 100$ seconds.

future, we will quantify overestimates more precisely.

Since the finest time scale in the data set is 1 second, and a good prediction interval $T \leq 100$ seconds, we have $n \in [10, 100]$. In the second experiment, we fixed $H = 0.85$ and $T = 100$ seconds, but varied n. We predicted using $n = 10, 20, 50, 100$, respectively, and got roughly the same results. For example, for server41 on day65, the three predictions using $n = 10, 50, 100$ all had 97.2% correctness, while the prediction using $n = 20$ had 96.3% correctness. This validates that with a right H, predictions using different n (within a certain range) are roughly equivalent, and that $H = 0.85$ appears to be the right value for this data set.

In the last experiment, we fixed $T = 100$ seconds, but varied both n and H. Our goal is to determine a right H based on how prediction performance changes as n increases. Figure 7.3 shows our prediction results for server41 on day65, where

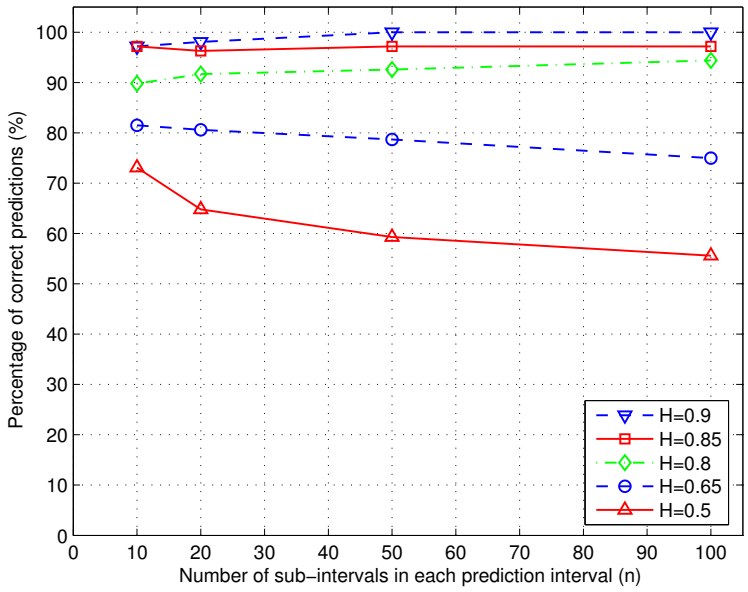


Figure 7.3: Prediction results for server41 on day65 of the 1998 World Cup data set, where $n = 10, 20, 50, 100$, $H = 0.5, 0.65, 0.8, 0.85, 0.9$, and $T = 100$ seconds.

$n = 10, 20, 50, 100$, and $H = 0.5, 0.65, 0.8, 0.85, 0.9$. From this figure, we can observe that the percentage of correct predictions increases as H increases. More importantly, when $H = 0.5$ or 0.65, the prediction performance degrades as n increases, meaning that these two H values are too small for this data set. On the other hand, when $H = 0.8$, 0.85, or 0.9, the prediction performance is roughly unchanged or goes up slightly as n increases, meaning that these three H values are fine for this data set. Also, $H = 0.85$ seems to be the best choice among them because $H = 0.9$ tends to overestimate too much whereas $H = 0.8$ cannot achieve 95% prediction correctness.

7.7 Summary

This chapter described a web traffic prediction algorithm using a multiple-time-scale approach. Through our experiments, we have demonstrated that our simple algorithm

is effective in upper bound prediction for short-term bursty web traffic. This algorithm provides insight into characterizing traffic of web hotspots, and is useful for web server overload prevention.

Chapter 8

Conclusions

This chapter summarizes the thesis, reviews the thesis contributions, and discusses future work.

8.1 Thesis Summary

This thesis described the techniques we developed for service discovery and web hotspot rescue, which fall into four categories: enhancements to the Service Location Protocol, selective anti-entropy for high availability partial replication, DotSlash—an automated web hotspot rescue system, and web traffic prediction for overload prevention. The focus of this thesis research is to build self-configuring and dynamically scalable computing systems. Service discovery allows end systems to discover desired services on network automatically, eliminating administrative configuration. Web hotspot rescue enables web sites to scale dynamically as needed, handling short-term dramatic load spikes autonomously without human intervention.

The Service Location Protocol (SLP) [59] is an IETF (Internet Engineering Task Force) proposed standard for service discovery in IP networks, and it is flexible,

lightweight, and powerful. To address the challenges in service discovery such as scalability and new discovery scenarios, we choose to enhance SLP instead of designing a new service discovery system from scratch so as to leverage existing efforts in service discovery and make our proposed techniques more likely to be used in real applications. We made four enhancements to SLP [153, 156, 157, 158, 152, 155, 146]: mesh enhancement that simplifies Service Agent registrations and improves the consistency of peer Directory Agents, remote service discovery that enables SLP users to discover services at remote DNS domains, preference filters that facilitate processing of search results at SLP servers, and global attributes that allow using a single query to search services across multiple types. These enhancements improve SLP efficiency and scalability, and enable SLP to better support new and advanced discovery scenarios. The SLP mesh enhancement (mSLP), remote service discovery, and preference filters are now experimental RFCs (Request for Comments) [157, 156, 158]. We expect that similar techniques can be applied to other service discovery systems as well.

Anti-entropy [44] is an important mechanism to achieve eventual consistency among a set of replicas, where an update is accepted by one replica first, and then the update is propagated asynchronously to the remaining replicas. Traditional anti-entropy [96, 54] only supports full replication. We enhanced it to support partial replication by allowing two replicas to selectively reconcile inconsistent data in a session. Using our proposed selective anti-entropy [147], a replica can choose to reconcile inconsistent data in any number of subsets. When all subsets are chosen, selective anti-entropy is equivalent to traditional anti-entropy. The main benefit of selective anti-entropy is that it allows the summary vector mechanism in traditional anti-entropy to be applicable to partial replication by using safe sessions. As a generalization of traditional anti-entropy, selective anti-entropy is flexible and can support both full replication and partial replication.

Web hotspots, also known as flash crowds or the Slashdot effect [4], are short-term dramatic load spikes that can seriously degrade the service quality of affected web sites. Web hotspots may trigger a large load increase but only last for a short period. For such situations, over-provisioning a web site is not only uneconomical but also difficult since the peak load is hard to predict. To handle web hotspots effectively, we developed DotSlash [150, 149, 151, 154], a self-configuring and scalable web hotspot rescue system. DotSlash works autonomously. It uses service discovery to allocate resources dynamically from a server pool distributed globally, and uses adaptive overload control to automate the whole rescue process. DotSlash is a cost-effective mechanism for small to medium-sized web sites to handle short-term dramatic load spikes. As a comprehensive solution, DotSlash enables a web site to build an adaptive distributed web server system on the fly, replicate application programs dynamically, and set up distributed query result caching on demand. These techniques relieve a spectrum of bottlenecks ranging from access network bandwidth to web servers, application servers, and database servers.

For overload prevention, we need to predict whether the future traffic volume will exceed the server's capacity or a preset threshold. Thus, we developed a prediction algorithm for estimating the upper bound of future web traffic volume [148]. We employ a multiple-time-scale approach by using traffic information at a smaller time scale to forecast traffic volume at a larger time scale. Moreover, we utilize traffic statistical properties other than curve fitting to forecast traffic volume. Our prediction algorithm provides insight into characterizing traffic of web hotspots, and is useful for web server overload prevention systems such as DotSlash.

8.2 Thesis Contributions

This thesis contributes to the area of networking and distributed systems. In particular, it contributes to the emerging area of autonomic computing systems. We recapitulate the thesis contributions as follows.

- **Enhancements to the Service Location Protocol.** We made four enhancements to the Service Location Protocol (SLP): mesh enhancement that simplifies Service Agent registrations and improves the consistency of peer Directory Agents, remote service discovery that enables SLP users to discover services at remote DNS domains, preference filters that facilitate processing of search results at SLP servers, and global attributes that allow using a single query to search services across multiple types. These enhancements improve SLP efficiency and scalability, and enable SLP to better support new and advanced discovery. The SLP mesh enhancement (mSLP), remote service discovery, and preference filters are now experimental RFCs (Request for Comments) [157, 156, 158]. We expect that similar techniques can be applied to other service discovery systems as well.

- **Selective anti-entropy.** We developed selective anti-entropy [147], a generic mechanism for high availability partial replication. Traditional anti-entropy [96, 54] only supports full replication. We enhanced it to support partial replication by allowing two replicas to selectively reconcile inconsistent data in a session. As a generalization of traditional anti-entropy, selective anti-entropy is flexible and applicable to both full replication and partial replication.

- **DotSlash—An automated web hotspot rescue system.** We developed DotSlash [150, 151, 154, 144], a self-configuring and scalable rescue system for

handling web hotspots effectively. DotSlash works autonomously. It uses service discovery to allocate resources dynamically from a server pool distributed globally, and uses adaptive overload control to automate the whole rescue process. DotSlash is a cost-effective mechanism for small to medium-sized web sites to handle short-term dramatic load spikes. As a comprehensive solution, DotSlash enables a web site to build an adaptive distributed web server system on the fly, replicate application programs dynamically, and set up distributed query result caching on demand. These techniques relieve a spectrum of bottlenecks ranging from access network bandwidth to web servers, application servers, and database servers.

- **Web traffic prediction for overload prevention.** We developed a prediction algorithm for estimating the upper bound of future web traffic volume [148], which is simple and effective for short-term bursty web traffic. We employ a multiple-time-scale approach, and utilize traffic statistical properties to forecast traffic volume. Our prediction algorithm provides insight into characterizing traffic of web hotspots, and is useful for web server overload prevention.

8.3 Future Work

For the research work described in this thesis, there are several interesting subjects worthy of further investigation.

- **Location-based service discovery.** Currently, services are discovered mainly based on network proximity using multicast and based on domain (e.g., administrative domain and application domain) specific registries. Another useful dimension is to discover services based on proximity of geographic locations

[114, 11, 76], which is not well supported by existing network infrastructure. In general, there are two types of geographic information: geospatial coordinates (longitude/latitude) and civic addresses (country, state, county, etc.). To build geographic information infrastructure in a distributed and scalable way, a mechanism similar to DNS may be used. To support location-based service discovery in SLP, we can define a global attribute "service-geographic-location" and associate a handler `GeoLoc` with it. A service provides its geographic location in its registration, and a client specifies its geographic location via a preference filter. The proximity of geographic locations between a client and a service is calculated by the `GeoLoc` handler, which may need to look up a geographic location database.

- **Hotspot rescue for different Internet servers.** DotSlash is an automated hotspot rescue system for handling short-term dramatic load spikes effectively. Currently, DotSlash focuses on regular web sites that have a majority of small web objects with a size in the order of kilobytes. For these web sites, we use an on-demand policy for replication, and a weighted round robin policy for request redirection. To perform hotspot rescue for large objects with a size in the order of megabytes (or even gigabytes) such as software packages and MP3 music files, we need to incorporate other polices in request redirection such as locality of requested objects at rescue servers and proximity between the client and rescue servers. Also, we need to incorporate other polices in replication such as partitioning the set of objects to be replicated among rescue servers. So far, we discuss hotspot rescue in the context of web servers; it is an interesting issue to apply similar techniques to other Internet servers such as FTP servers, game servers, and SIP proxy servers [117].

- **Different usage models for DotSlash.** We outlined three usage models for DotSlash in Section 4.3.1, namely open communities, closed communities, and flood-insurance closed communities. Currently, our prototype system [144] only supports open communities. We can add security and insurance mechanisms to DotSlash to support the other two usage models. First, we can add authentication to DotSlash to support closed communities. Authentication is needed for accepting service registrations, choosing rescue server candidates, and accepting rescue requests. Secondly, in addition to authentication, we can add tokens to DotSlash to support flood-insurance closed communities. An authorized web server can obtain tokens from the community authority by paying an insurance premium. To secure a rescue relationship, the origin server needs to transfer one token to the rescue server.

Bibliography

[1] T. Abdelzaher and N. Bhatti. Web server QoS management by adaptive content delivery. In *International Workshop on Quality of Service (IWQoS)*, London, England, June 1999.

[2] A. Acharya and A. Shaikh. Using mobility support for request routing in IPv6 CDNs. In *International Workshop on Web Content Caching and Distribution (WCW)*, Boulder, Colorado, August 2002.

[3] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *ACM Symposium on Operating Systems Principles (SOSP)*, Kiawah Island Resort, South Carolina, December 1999.

[4] S. Adler. The Slashdot effect: An analysis of three Internet publications. http://ssadler.phy.bnl.gov/adler/SDE/SlashDotEffect.html.

[5] Akamai homepage. http://www.akamai.com/.

[6] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan. DBProxy: A dynamic data cache for web applications. In *International Conference on Data Engineering (ICDE)*, Bangalore, India, March 2003.

[7] C. Amza, E. Cecchet, A. Chanda, A. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel. Specification and implementation of dynamic web site benchmarks. In *International Workshop on Web Content Caching and Distribution (WCW)*, Boulder, Colorado, August 2002.

[8] C. Amza, A. L. Cox, and W. Zwaenepoel. A comparative evaluation of transparent scaling techniques for dynamic content servers. In *International Conference on Data Engineering (ICDE)*, Tokyo, Japan, April 2005.

[9] Apache. HTTP server project. http://httpd.apache.org/.

[10] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D. P. Pazel, J. Pershing, and B. Rochwerger. Océano – SLA based management of a computing utility. In *IFIP/IEEE International Symposium on Integrated Network Management (IM)*, Seattle, Washington, May 2001.

[11] K. Arabshian and H. Schluzrinnes. GloServ: Global service discovery architecture. In *International Conference on Mobile and Ubiquitous Systems: Networking and Services (Mobiquitous)*, Boston, Massachusetts, August 2004.

[12] M. Arlitt and T. Jin. Workload characterization of the 1998 World Cup web site. Technical Report HPL-1999-35R1, HP Labs., 1999.

[13] M. Aron, D. Sanders, P. Druschel, and W. Zwaenepoel. Scalable context-aware request distribution in cluster-based network servers. In *Annual Usenix Technical Conference*, San Diego, California, June 2000.

[14] M. Bakke, J. Hufferd, K. Voruganti, M. Krueger, and T. Sperry. Finding Internet small computer systems interface (iSCSI) targets and name servers by using

service location protocol version 2 (SLPv2). RFC 4018, Internet Engineering Task Force, April 2005.

[15] W. Balke and M. Wagner. Towards personalized selection of web services. In *International World Wide Web Conference (WWW)*, Budapest, Hungary, May 2003.

[16] G. Banga and P. Druschel. Measuring the capacity of a web server. In *USENIX Symposium on Internet Technologies and Systems (USITS)*, Monterey, California, December 1997.

[17] A. Barbir, B. Cain, R. Nair, and O. Spatscheck. Known content network (CN) request-routing mechanisms. RFC 3568, Internet Engineering Task Force, July 2003.

[18] P. Barford and D. Plonka. Characteristics of network traffic flow anomalies. In *ACM SIGCOMM Internet Measurement Workshop*, San Francisco, California, November 2001.

[19] Y. Baryshnikov, E. Coffman, D. Rubenstein, and B. Yimwadsana. Traffic prediction on the Internet. Technical Report EE2002-05-141, Columbia University, May 2002.

[20] BEA WebLogic. http://www.bea.com/products/weblogic/server/.

[21] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform resource identifiers (URI): Generic syntax. RFC 2396, Internet Engineering Task Force, August 1998.

[22] Andrew Binstock. Hashing rehashed. Dr. Dobb's, April 1996.

[23] BitTorrent homepage. http://bitconjurer.org/BitTorrent/.

[24] Bluetooth homepage. http://www.bluetooth.com/.

[25] Bonjour homepage. http://developer.apple.com/networking/bonjour/.

[26] C. Bornhovd, M. Altinel, C. Mohan, H. Pirahesh, and B. Reinwald. Adaptive database caching with DBCache. *IEEE Data Engineering Bulletin*, 27(2):11–18, June 2004.

[27] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible markup language (XML) 1.0 (second edition). W3C Recommendation REC-xml-20001006, World Wide Web Consortium (W3C), October 2000. Available at http://www.w3.org/XML/.

[28] V. Cardellini, E. Casalicchio, M. Colajanni, and P. Yu. The state of the art in locally distributed web-server systems. *ACM Computing Surveys*, 34(2):263–311, June 2002.

[29] V. Cardellini, M. Colajanni, and P. S. Yu. Geographic load balancing for scalable distributed web systems. In *International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, San Francisco, California, August 2000.

[30] R. L. Carter and M. E. Crovella. On the network impact of dynamic server selection. *Computer Networks*, 31(23–24):2529–2558, December 1999.

[31] P. Castro, B. Greenstein, R. Muntz, C. Bisdikian, P. Kermani, and M. Papadopouli. Locating application data across service discovery domains. In *ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom)*, Rome, Italy, July 2001.

[32] E. Cecchet. C-JDBC: A middleware framework for database clustering. *IEEE Data Engineering Bulletin*, 27(2):16–26, June 2004.

[33] E. Cecchet, A. Chanda, S. Elnikety, J. Marguerite, and W. Zwaenepoel. Performance comparison of middleware architectures for generating dynamic web content. In *ACM/IFIP/USENIX International Middleware Conference*, Rio de Janeiro, Brazil, June 2003.

[34] J. Challenger, P. Dantzig, A. Iyengar, M. Squillante, and L. Zhang. Efficiently serving dynamic data at highly accessed web sites. *IEEE/ACM Transactions on Networking*, 12(2):233–246, April 2004.

[35] S. Cheshire, B. Aboba, and E. Guttman. Dynamic configuration of IPv4 link-local addresses. RFC 3927, Internet Engineering Task Force, May 2005.

[36] S. Cheshire and M. Krochmal. DNS-based service discovery. Internet draft, Internet Engineering Task Force, June 2005. Work in progress.

[37] Cisco DistributedDirector. http://www.cisco.com/warp/public/cc/pd/cxsr/dd/tech/dd_wp.htm.

[38] E. Coffman, P. Jelenkovic, J. Nieh, and D. Rubenstein. The Columbia hotspot rescue service: A research plan. Technical Report EE2002-05-131, Department of Electrical Engineering, Columbia University, May 2002.

[39] E. Cohen and S. Shenker. Replication strategies in unstructured peer-to-peer networks. In *SIGCOMM Symposium on Communications Architectures and Protocols*, Pittsburgh, Pennsylvania, August 2002.

[40] M. E. Crovella and A. Bestavros. Self-similarity in world wide web traffic: Evidence and possible causes. *IEEE/ACM Transactions on Networking*, 5(6):835–846, December 1997.

[41] S. Czerwinski, B. Zhao, T. Hodes, A. Joseph, and R. Katz. An architecture for a secure service discovery service. In *ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom)*, Seattle, Washington, August 1999.

[42] A. Datta, K. Dutta, H. Thomas, D. Vandermeer, and K. Ramamritham. Proxy-based acceleration of dynamically generated content on the world wide web: An approach and implementation. *ACM Transactions on Database Systems*, 29(2):403–443, June 2004.

[43] M. Day, B. Cain, G. Tomlinson, and P. Rzewski. A model for content internetworking (CDI). RFC 3466, Internet Engineering Task Force, February 2003.

[44] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 1–12, Vancouver, Canada, August 1987.

[45] Digital island. http://www.digitalisland.com/.

[46] R. Droms. Dynamic host configuration protocol. RFC 2131, Internet Engineering Task Force, March 1997.

[47] S. G. Dykes, K. A. Robbins, and C. L. Jeffery. An empirical evaluation of client-side server selection algorithms. In *Proceedings of the Conference on Computer Communications (IEEE Infocom)*, Anchorage, Alaska, April 2001.

[48] P. Felber, T. Kaldewey, and S. Weiss. Proactive hot spot avoidance for web server dependability. In *IEEE Symposium on Reliable Distributed Systems (SRDS)*, Florianopolis, Brazil, October 2004.

[49] R. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. RFC 2068, Internet Engineering Task Force, January 1997.

[50] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the Grid: Enabling scalable virtual organizations. *International Journal of High Performance Computing Applications*, 15(3):200–222, August 2001.

[51] G. F. Franklin, J. D. Powell, and A. Emami-Naeini. *Feedback Control of Dydamic Systems*. Prentice Hall, 2006.

[52] M. J. Freedman, E. Freudenthal, and D. Mazires. Democratizing content publication with Coral. In *USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, California, March 2004.

[53] Gnutella homepage. http://www.gnutella.com.

[54] R. Golding. *Weak-consistency group communication and membership*. PhD thesis, Computer Science, University of California at Santa Cruz, December 1992.

[55] Google homepage. http://www.google.com.

[56] A. Gulbrandsen, P. Vixie, and L. Esibov. A DNS RR for specifying the location of services (DNS SRV). RFC 2782, Internet Engineering Task Force, February 2000.

[57] E. Guttman. The serviceid: URI scheme for service location. Internet draft, Internet Engineering Task Force, August 2002. Work in progress.

[58] E. Guttman, C. E. Perkins, and J. Kempf. Service templates and service: schemes. RFC 2609, Internet Engineering Task Force, June 1999.

[59] E. Guttman, C. E. Perkins, J. Veizades, and M. Day. Service location protocol, version 2. RFC 2608, Internet Engineering Task Force, June 1999.

[60] R. Herriot, S. Butler, and P. G. Moore. Internet printing protocol/1.1: Encoding and transport. RFC 2910, Internet Engineering Task Force, September 2000.

[61] T. Howes. The string representation of LDAP search filters. RFC 2254, Internet Engineering Task Force, December 1997.

[62] T. Howes, M. Wahl, and A. Anantha. LDAP control extension for server side sorting of search results. RFC 2891, Internet Engineering Task Force, August 2000.

[63] IBM WebSphere. http://www-306.ibm.com/software/websphere/.

[64] D. Johnson and S. Deering. Reserved IPv6 subnet anycast addresses. RFC 2526, Internet Engineering Task Force, March 1999.

[65] J. Jung, B. Krishnamurthy, and M. Rabinovich. Flash crowds and denial of service attacks: Characterization and implications for CDNs and web sites. In *International World Wide Web Conference (WWW)*, Honolulu, Hawaii, May 2002.

[66] Kazaa homepage. http://www.kazaa.com.

[67] J. Kempf, R. Moats, and P. St. Pierre. Conversion of LDAP schemas to and from SLP templates. RFC 2926, Internet Engineering Task Force, September 2000.

[68] J. Kempf and G. Montenegro. Finding an RSIP server with SLP. RFC 3105, Internet Engineering Task Force, October 2001.

[69] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer Magazine*, 36(1):41–50, January 2003.

[70] K. Kong and D. Ghosal. Mitigating server-side congestion in the Internet through pseudoserving. *IEEE/ACM Transactions on Networking*, 7(4):530–544, August 1999.

[71] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems*, 10(4):360–391, November 1992.

[72] P. Larson, J. Goldstein, H. Guo, and J. Zhou. MTCache: Mid-tier database caching for SQL server. *IEEE Data Engineering Bulletin*, 27(2):35–40, June 2004.

[73] E. Leach, M. Mealling, and R. Salz. A universally unique identifier (UUID) URN namespace. RFC 4122, Internet Engineering Task Force, July 2005.

[74] W. LeFebvre. CNN.com: Facing a world crisis. Invited talk at USENIX LISA, December 2001.

[75] W. E. Leland, M. S. Taqqu, W. Willinger, and D. V. Wilson. On the self-similar nature of Ethernet traffic (extended version). *IEEE/ACM Transactions on Networking*, 2(1):1–15, February 1994.

[76] Q. Li. An architecture for geographically-oriented service discovery on the Internet. Master's thesis, Computer Science, University of Waterloo, 2002.

[77] Q. Li and B. Moon. Distributed cooperative Apache web server. In *International World Wide Web Conference (WWW)*, Hong Kong, May 2001.

[78] libmemcache homepage. http://people.freebsd.org/~seanc/libmemcache/.

[79] B. Melcher and B. Mitchell. Towards an autonomic framework: Self-configuring network services and developing autonomic applications. *Intel Technology Journal*, 8(4):278–291, November 2004.

[80] memcached homepage. http://www.danga.com/memcached/.

[81] D. Meyer. Administratively scoped IP multicast. RFC 2365, Internet Engineering Task Force, July 1998.

[82] P. Mockapetris. Domain names - concepts and facilities. RFC 1034, Internet Engineering Task Force, November 1987.

[83] P. Mockapetris. Domain names - implementation and specification. RFC 1035, Internet Engineering Task Force, November 1987.

[84] D. Mosberger and T. Jin. httperf—a tool for measuring web server performance. In *Workshop on Internet Server Performance (WISP)*, Madison, Wisconsin, June 1998.

[85] MySQL cluster. http://www.mysql.com/products/database/cluster/.

[86] Napster homepage. http://www.napster.com.

[87] J. Naugle, K. Kasthurirangan, and G. Ledford. TN3270E service location and session balancing. RFC 3049, Internet Engineering Task Force, January 2001.

[88] Netcraft. Web server survey.
http://news.netcraft.com/archives/web_server_survey.html.

[89] D. P. Olshefski, J. Nieh, and D. Agrawal. Inferring client response time at the web server. In *ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, Marina del Rey, California, June 2002.

[90] C. Olston, A. Manjhi, C. Garrod, A. Ailamaki, B. M. Maggs, and T. C. Mowry. A scalability service for dynamic web applications. In *The Conference on Innovative Data Systems Research (CIDR)*, Asilomar, California, January 2005.

[91] Oracle real application clusters (RAC).
http://www.oracle.com/technology/products/database/clustering/index.html.

[92] V. Padmanabhan and K. Sripanidkulchai. The case for cooperative networking. In *International Workshop on Peer-to-Peer Systems (IPTPS)*, Cambridge, Massachusetts, March 2002.

[93] C. Partridge, T. Mendez, and W. Milliken. Host anycasting service. RFC 1546, Internet Engineering Task Force, November 1993.

[94] K. M. Passino and S. Yurkovich. *Fuzzy Control*. Addison-Wesley, 1998.

[95] C. Perkins and E. Guttman. DHCP options for service location protocol. RFC 2610, Internet Engineering Task Force, June 1999.

[96] K. Petersen, M. J. Spreizer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 288–301, Saint Malo, France, October 1997.

[97] G. Pierre and M. v. Steen. Globule: A platform for self-replicating web documents. In *International Conference on Protocols for Multimedia Systems*, Enschede, Netherlands, October 2001.

[98] G. Pierre, M. v. Steen, and A. S. Tanenbaum. Dynamically selecting optimal distribution strategies for web documents. *IEEE Transactions on Computers*, 51(7):637–651, June 2002.

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

[100] C. Plattner and G. Alonso. Ganymed: Scalable replication for transactional web applications. In *ACM/IFIP/USENIX International Middleware Conference*, Toronto, Canada, October 2004.

[101] Todd Poynor. Automating infrastructure composition for Internet services. In *Systems Administration Conference (LISA)*, San Diego, California, December 2001.

[102] M. Rabinovich, Z. Xiao, and A. Aggarwal. Computing on the edge: A platform for replicating Internet applications. In *International Workshop on Web Caching and Content Distribution (WCW)*, Hawthorne, New York, September 2003.

[103] S. Ranjan, R. Karrer, and E. Knightly. Wide area redirection of dynamic content by Internet data centers. In *Proceedings of the Conference on Computer Communications (IEEE Infocom)*, Hong Kong, China, March 2004.

[104] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *SIGCOMM Symposium on Communications Architectures and Protocols*, San Diego, California, August 2001.

[105] Y. Rekhter and T. Li. A border gateway protocol 4 (BGP-4). RFC 1771, Internet Engineering Task Force, March 1995.

[106] Resource description framework (RDF). http://www.w3.org/RDF/.

[107] F. Reynolds. An RDF framework for resource discovery. In *Semantic Web Workshop*, Hong Kong, China, May 2001.

[108] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. R. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: session initiation protocol. RFC 3261, Internet Engineering Task Force, June 2002.

[109] 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)*, Heidelberg, Germany, November 2001.

[110] RRDtool homepage. http://people.ee.ethz.ch/~oetiker/webtools/rrdtool/.

[111] RUBBoS: Rice university bulletin board system. http://www.cs.rice.edu/CS/Systems/DynaServer/RUBBoS/.

[112] RUBiS: Rice university bidding system. http://www.cs.rice.edu/CS/Systems/DynaServer/RUBiS/.

[113] P. Rzewski, M. Day, and D. Gilletti. Content internetworking (CDI) scenarios. RFC 3570, Internet Engineering Task Force, July 2003.

[114] H. Schulzrinne. Location-to-URL mapping protocol (LUMP). Internet draft, Internet Engineering Task Force, October 2005. Work in progress.

[115] S. Schwarz. Web servers, earthquakes, and the Slashdot effect. http://pasadena.wr.usgs.gov/office/stans/slashdot.html.

[116] A. Shaikh, R. Tewari, and M. Agrawal. On the effectiveness of DNS-based server selection. In *Proceedings of the Conference on Computer Communications (IEEE Infocom)*, Anchorage, Alaska, April 2001.

[117] K. Singh and H. Schulzrinne. Failover and load sharing in SIP telephony. In *International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*, Philadelphia, Pennsylvania, July 2005.

[118] S. Sivasubramanian, G. Alonso, G. Pierre, and M. v. Steen. GlobeDB: Autonomic data replication for web applications. In *International World Wide Web Conference (WWW)*, Chiba, Japan, May 2005.

[119] S. Sivasubramanian, G. Pierre, M. v. Steen, and G. Alonso. GlobeCBC: Content-blind result caching for dynamic web applications. Submitted for publication, Vrije Universiteit, October 2005.

[120] Slashdot homepage. http://slashdot.org/.

[121] T. Stading, P. Maniatis, and M. Baker. Peer-to-peer caching schemes to address flash crowds. In *International Workshop on Peer-to-Peer Systems (IPTPS)*, Cambridge, Massachusetts, March 2002.

[122] A. Stavrou, D. Rubenstein, and S. Sahu. A lightweight, robust P2P system to handle flash crowds. In *IEEE International Conference on Network Protocols (ICNP)*, Paris, France, November 2002.

[123] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream control transmission protocol. RFC 2960, Internet Engineering Task Force, October 2000.

[124] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and Sonesh Surana. Internet indirection infrastructure. In *SIGCOMM Symposium on Communications Architectures and Protocols*, Pittsburgh, Pennsylvania, August 2002.

[125] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *SIGCOMM Symposium on Communications Architectures and Protocols*, San Diego, California, August 2001.

[126] X. Tang and S. T. Chanson. Coordinated en-route web caching. *IEEE Transactions on Computers*, 51(6):595–607, June 2002.

[127] L. Titchkosky, M. Arlitt, and C. Williamson. A performance comparison of dynamic web technologies. *ACM SIGMETRICS Performance Evaluation Review*, 31(3):2–11, December 2003.

[128] Transaction processing performance council. http://www.tpc.org/tpcw/.

[129] Universal description, discovery and integration (UDDI) homepage. http://www.uddi.org/.

[130] Universal plug and play (UPnP) homepage. http://www.upnp.org.

[131] A. Vakali and G. Pallis. Content delivery networks: Status and trends. *IEEE Internet Computing*, 7(6):68–74, December 2003.

[132] P. Vixie, S. Thomson, Y. Rekhter, and J. Bound. Dynamic updates in the domain name system (DNS UPDATE). RFC 2136, Internet Engineering Task Force, April 1997.

[133] M. Wahl, T. Howes, and S. Kille. Lightweight directory access protocol (v3). RFC 2251, Internet Engineering Task Force, December 1997.

[134] J. Waldo. The Jini architecture for network-centric computing. *Communications ACM*, 42(7):76–82, July 1999.

[135] J. Wang. A survey of web caching schemes for the Internet. *ACM Computer Communication Review (CCR)*, 29(5):36–46, October 1999.

[136] L. Wang, V. Pai, and L. Peterson. The effectiveness of request redirection on CDN robustness. In *Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, Massachusetts, December 2002.

[137] L. Wang, K. Park, R. Pang, V. Pai, and L. Peterson. Reliability and security in the CoDeeN content distribution network. In *Annual Usenix Technical Conference*, Boston, Massachusetts, June 2004.

[138] S. Waterhouse, D. M. Doolin, G. Kan, and Y. Faybishenko. Distributed search in P2P networks. *IEEE Internet Computing*, 6(1):68–72, February 2002.

[139] C. Weider, A. Herron, A. Anantha, and T. Howes. LDAP control extension for simple paged results manipulation. RFC 2696, Internet Engineering Task Force, September 1999.

[140] M. Welsh and D. Culler. Adaptive overload control for busy Internet servers. In *USENIX Conference on Internet Technologies and Systems (USITS)*, Seattle, Washington, March 2003.

[141] Duane Wessels. Report on the effect of the independent council report on the NLANR web caches. http://www.ircache.net/Statistics/ICreport/, September 1998.

[142] J. D. Zawodny and D. J. Balling. *High Performance MySQL: Optimization, Backups, Replication, and Load Balancing.* O'Reilly, 2004.

[143] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, January 2004.

[144] Weibin Zhao and Henning Schulzrinne. DotSlash—an automated web hotspot rescue system project. http://dotslash.sourceforge.net/.

[145] Weibin Zhao and Henning Schulzrinne. Service location protocol enhancements project. http://mslp.sourceforge.net/.

[146] Weibin Zhao and Henning Schulzrinne. Improving SLP efficiency and extendability by using global attributes and preference filters. In *International Conference on Computer Communications and Networks (ICCCN)*, Miami, Florida, October 2002.

[147] Weibin Zhao and Henning Schulzrinne. Selective anti-entropy. In *ACM Symposium on Principles of Distributed Computing (PODC) (brief announcement)*, Monterey, California, July 2002.

[148] Weibin Zhao and Henning Schulzrinne. Predicting the upper bound of web traffic volume using a multiple time scale approach. In *International World Wide Web Conference (WWW) (poster session)*, Budapest, Hungary, May 2003.

[149] Weibin Zhao and Henning Schulzrinne. Building an adaptive distributed web server system on the fly for handling web hotspots. In *ACM Symposium on Principles of Distributed Computing (PODC) (brief announcement)*, St. John's, Newfoundland, Canada, July 2004.

[150] Weibin Zhao and Henning Schulzrinne. DotSlash: A self-configuring and scalable rescue system for handling web hotspots effectively. In *International Workshop on Web Caching and Content Distribution (WCW)*, Beijing, China, October 2004.

[151] Weibin Zhao and Henning Schulzrinne. DotSlash: Handling web hotspots at dynamic content web sites. In *IEEE Global Internet Symposium*, Miami, Florida, March 2005.

[152] Weibin Zhao and Henning Schulzrinne. Enabling global service attributes in the service location protocol. Internet draft, Internet Engineering Task Force, October 2005. Work in progress.

[153] Weibin Zhao and Henning Schulzrinne. Enhancing service location protocol for efficiency, scalability and advanced discovery. *Journal of Systems and Software*, 75(1):193–204, February 2005.

[154] Weibin Zhao and Henning Schulzrinne. DotSlash: An automated web hotspot rescue system with on-demand query result caching. In *IEEE International Conference on Autonomic Computing (ICAC) (poster session)*, Dublin, Ireland, June 2006.

[155] Weibin Zhao, Henning Schulzrinne, and E. Guttman. mSLP - mesh-enhanced service location protocol. In *International Conference on Computer Communications and Networks (ICCCN)*, Las Vegas, Nevada, October 2000.

[156] Weibin Zhao, Henning Schulzrinne, and E. Guttman. Mesh-enhanced service location protocol (mSLP). RFC 3528, Internet Engineering Task Force, April 2003.

[157] Weibin Zhao, Henning Schulzrinne, E. Guttman, C. Bisdikian, and W. Jerome. Select and sort extensions for the service location protocol (SLP). RFC 3421, Internet Engineering Task Force, November 2002.

[158] Weibin Zhao, Henning Schulzrinne, E. Guttman, C. Bisdikian, and W. Jerome. Remote service discovery in the service location protocol (SLP) via DNS SRV. RFC 3832, Internet Engineering Task Force, July 2004.