DataSlicer: A Hosting Platform For Data-Centric Network Services

by

Congchun He

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

New York University

May 2006

Approved: ________________________

Research Advisor: Vijay Karamcheti

© Congchun He

All Rights Reserved 2006

To my mother and father, and to my loving wife.

Acknowledgment

First and foremost, I would like to take this opportunity to express my gratitude to my advisor, Professor Vijay Karamcheti, who is a great researcher with keen interests in science and possesses great personalities. Without his guidance, encouragement and collaboration, this dissertation would not have been possible.

I would also like to thank my proposal and dissertation committees for their insightful opinions and suggestions on my research work. My thanks also go to the staff in the Computer Science Department, whose efforts made my study in the Ph.D. program a pleasant process.

I am eternally grateful to my parents who consistently believe in and stand behind me. Most of all, I thank my loving wife, Aiting Hou, for her great care, unconditional support, and continuous encouragement. Because of her, my life has been more colorful and joyful.

Abstract

As the Web evolves, the number of network services deployed on the Internet has been growing at a dramatic pace. Such services usually involve a massive volume of data stored in physical or virtual back-end databases, and access the data to dynamically generate responses for client requests. These characteristics restrict use of traditional mechanisms for improving service performance and scalability: large volumes prevent replication of the service data at multiple sites required by content distribution schemes, while dynamic responses do not support the reuse required by web caching schemes.

However, many deployed data-centric network services share other properties that can help alleviate this situation: (1) service usage patterns exhibit locality of various forms, and (2) services are accessed using standard protocols and publicly known message structures. When properly exploited, these characteristics enable the design of alternative caching infrastructures, which leverage distributed network intermediaries to inspect traffic flowing between clients and services, infer locality information dynamically, and potentially improve service performance by taking actions such as partial service replication, request redirection, or admission control.

This dissertation investigates the nature of locality in service usage patterns for

two well-known web services, and reports on the design, implementation, and evaluation of such a network intermediary architecture, named *DataSlicer*. DataSlicer incorporates four main techniques: (1) Service-neutral request inspection and locality detection on distributed network intermediaries; (2) Construction of oriented overlays for clustering client requests; (3) Integrated load-balancing and service replication mechanisms that improve service performance and scalability by either redistributing the underlying traffic in the network or creating partial service replicas on demand at appropriate network locations; and (4) Robustness mechanisms to maintain system stability in a wide-area network environment.

DataSlicer has been successfully deployed on the PlanetLab network. Extensive experiments using synthetic workloads show that our approach can: (1) create appropriate oriented overlays to cluster client requests according to multiple application metrics; (2) detect locality information across multiple dimensions and granularity levels; (3) leverage the detected locality information to perform appropriate load-balancing and service replication actions with minimal cost; and (4) ensure robust behavior in the face of dynamically changing network conditions.

Contents

Dedication **iii**

Acknowledgment **iv**

Abstract **v**

List of Figures **xii**

List of Tables **xvi**

1 Introduction **1**

1.1 Motivation 1

1.2 Thesis and Methodology 7

1.3 Contributions 10

1.4 Thesis Organization 13

2 Problem Description **14**

2.1 Data-centric Services 14

2.1.1 Motivating Example: Maps Service 16

2.2 Problem Statement 18
2.2.1 Abstract Service Architecture 18
2.2.2 Challenges 18
2.2.3 Existence of Service Usage Locality 19
2.2.4 Construction of Oriented Overlay Networks 20
2.2.5 Request Inspection and Locality Detection 20
2.2.6 Load-Balancing and Service Replication Problem 21
2.2.7 System Robustness 22
2.3 DataSlicer 22
2.3.1 Router Functionality 25
2.3.2 Replica Functionality 28
2.4 Summary 29
3 Background and Related Work 30
3.1 Network Services and Related Technologies 30
3.1.1 Transport Protocols 31
3.1.2 Network Service Architectures 33
3.1.3 The Underlying Networks 37
3.2 System-level Approaches 38
3.2.1 Component-based Service Integration 39
3.2.2 Web Caching and Replication 41
3.3 Summary 55
4 Service Usage Locality and its Detection 56
4.1 Existence of Locality 56

4.1.1 Web-trace Structure 58
4.1.2 Locality Characterization 63
4.1.3 Methodology 66
4.1.4 Results 70
4.1.5 Discussion 79
4.2 In-network Request Inspection and Locality Detection 81
4.2.1 Service Registration 81
4.2.2 Cell Structure 86
4.2.3 Request Processing 88
4.2.4 Discussion 90
4.3 Summary 91

5 Oriented Overlays Construction and Maintenance 92
5.1 "Zone-based" Oriented Overlays 92
5.1.1 Basic Design 93
5.1.2 Construction Protocols 96
5.1.3 Maintenance Protocols 97
5.1.4 Overlay Properties 100
5.1.5 Extensions 101
5.2 Implementation 104
5.3 Discussion 105
5.4 Summary 108

6 Load-balancing and Service Replication 109
6.1 Load-Balancing 110

6.1.1 Problem Formulation 112
6.1.2 Approaches 115
6.1.3 Discussion 118
6.2 Service Replication 120
6.2.1 Problem Formulation 120
6.2.2 Algorithms 128
6.2.3 Discussion 140
6.3 Summary 141

7 System Robustness 142

7.1 Liveness Monitoring and Repair 143
7.2 Link-level Flow Control. 147
7.3 Non-blocking/Asynchronous Communication 151
7.4 Smoothing out Statistical Fluctuations in Network Measurements . . . 153
7.5 Summary 157

8 Evaluation 158

8.1 Experimental Environment 159
8.1.1 C# Prototype 160
8.1.2 C Prototype 161
8.2 Evaluation of C#/.NET Prototype 162
8.2.1 Configuration of the Emulated Network 162
8.2.2 Client Workload 164
8.3 Evaluation of C Prototype 173
8.3.1 Oriented Overlays Characteristics 174

8.3.2 Performance on Small-scale Networks 179
8.3.3 Performance on Large-scale Networks 184
8.4 Comparison with Other Approaches 192
8.4.1 Barebones System 192
8.4.2 Traditional Caching System 193
8.4.3 Proxy Server System 195
8.5 Summary 199

9 Conclusions and Future Work **201**
9.1 Summary 201
9.2 Conclusions 203
9.3 Future Work 204

Bibliography **207**

List of Figures

1.1 An illustrative example of data-centric network services in a wide-area network environment. 4

2.1 An overview of DataSlicer architecture 23

2.2 Functional Components of DataSlicer Router 26

4.1 Greedy computation of “load fraction”. 69

4.2 IP_3_PP_XY10_TS_All: Spatial locality for requests accessing the *x_rect.asp* service in the SkyServer trace. 70

4.3 IP_3_PP_TSXY_TS_All: Spatial locality for requests accessing the *tile.ashx* service in the TerraServer trace. 71

4.4 Workload locality for SkyServer’s *x_rect.asp* service. 72

4.5 Workload locality for SkyServer’s *x_rect.asp* service (cont’d). 73

4.6 Workload locality for TerraServer’s *tile.ashx* service. 74

4.7 Workload locality for TerraServer’s *tile.ashx* service (cont’d). 75

4.8 *GetMap* specification 84

4.9 DataSlicer Service Registration Interface 85

4.10 A Dynamic Data Structure: *Cell* 87
4.11 Service Request Processing at DataSlicer Routers 89

5.1 Overview of an oriented overlay for data-centric network services. The slim dotted lines show the connections of the constructed overlays. 95
5.2 Distributed algorithm for construction and maintenance of oriented overlays (Independently run for each origin server) 99
5.3 View of hyper-zone in oriented overlays with multi-metrics. 102

6.1 Load-Balancing: two illustrative scenarios. 111
6.2 Load-Balancing: challenges. 114
6.3 Response time measurement. 117
6.4 Computation of the response time for a request at a router. 122
6.5 Chain-based hierarchical routing network. 125
6.6 Centralized algorithm for service replication on a tree-topology network. 131
6.7 An illustrative example of applying the centralized algorithm for service replication on a tree-topology network 131
6.8 Distributed algorithm for service replication on a tree-topology network. 134
6.9 Distributed algorithm for service replication on a DAG-topology network. 139

7.1 Token-based flow control algorithm 149
7.2 An illustrative example for token-based flow control algorithm 150
7.3 Network latency measurements on the PlanetLab network. 155

8.1 Configuration of the emulated network. 163

8.2 Usage of parameter α and β in generation of client workloads. 165
8.3 Performance on an unloaded network. 166
8.4 Performance seen for a workload that exhibits distributed grouping and low spatial locality. 167
8.5 Performance seen for a workload that exhibits distributed grouping and high spatial locality workload. 169
8.6 Performance seen for a workload that exhibits centralized grouping and high spatial locality. 172
8.7 An oriented overlay with 3 origin servers constructed on the PlanetLab network. 174
8.8 An oriented overlay with a single origin server constructed on the PlanetLab network. 175
8.9 Examples of clustering in oriented overlays, evaluated using overlap score. 178
8.10 Configuration of the small-scale network on the PlanetLab network. . 180
8.11 Performance observed by a client at UCSB in the small-scale network experiment. 182
8.12 Percentage of satisfaction of clients vs. Experiment time 187
8.13 Statistical median response times observed by clients at Purdue and UCSB. 188
8.14 Sub-networks rooted at multiple replica sites in a large-scale experiment. 190
8.15 The running average of response times observed by a client at NYU in the barebones system. 194
8.16 The manually configured system using 4 proxy servers. 196

8.17 Comparison of client satisfaction. 198

List of Tables

4.1 Entry structure in the two web traces. 60
4.2 Fraction of static and dynamic content in the two web-traces. 61
4.3 Types of services showing request and client IP statistics. 62
4.4 Logical views of the two services. 65
4.5 Range of parameter values over which locality characteristics were investigated. 67

8.1 Network metrics on the PlanetLab Network and the Click-based emulated WAN. 164
8.2 Node distribution in the constructed oriented overlay with a single origin server 177
8.3 Latency dilation in the constructed oriented overlay with a single origin server. 177
8.4 Clustering in the constructed oriented overlay with a single origin server. 179
8.5 Load-balancing on the small-scale network. 183
8.6 Parameters of experiments running on the large-scale network. 185
8.7 Number of replicated cells for different response sizes. 197

Chapter 1

Introduction

1.1 Motivation

In the past decade, the Internet has evolved from its information centric roots into an infrastructure providing a wide variety of sophisticated Web services to its users. Most such services satisfy client requests by dynamically generating responses based on information stored in backend physical or virtual databases, which typically contain massive volumes of data. In fact, such *data-centric* services dominate the landscape, with examples ranging from search services such as Google, to e-commerce services such as Amazon, to imagery services such as Microsoft's MapPoint [1], TerraServer [2] and SkyServer [3], and emerging software-as-a-service trends exemplified by the recently released Windows Live products [4] from Microsoft. In large part due to the need to support both human user- and programmatic access, such services tend to be accessed using standard protocols (such as HTTP, SOAP, and/or XML-RPC) and have request message structures that are publicly known. As example,

most of the functionality at Google's and Amazon's web sites is accessible programmatically using the Google Web APIs [5] and Amazon Web Services [6] interfaces, respectively.

The service providers usually host such *data-centric* services on one or a small number of server sites. For example, Google hosted its cached data on about 10 data centers before January 2004. In 2005, this number grew to about 61, but some of them were pulled offline later, resulting in about 26 data centers remaining active. On the other end, the SkyServer service is hosted only at two websites as of this writing, one at FermiLab [7] and the other at John Hopkins University. End-clients of data-centric network services are typically geographically distributed, accessing the services across a large-scale, heterogenous wide-area network (WAN). Additionally, end-clients access the services using a variety of devices ranging from supercomputers and PCs to hand-held devices, presenting various expectations about the quality of service (QoS). According to [8], the primary performance concern for the Internet service users is *request response time*. Providing good performance to end-clients is crucial for such services: in 2001, a Zona Research study [9] penned a 8-second download rule to assess user patience in accessing e-commerce sites, and revealed that, due to user impatience with long response time, the revenue losses in e-commerce sales was estimated to be 21 billion dollars (such revenue losses was estimated to be 1.9 billion dollars in 1998 [10], and approximately 4.35 billion in 1999 [11]).

The response delay seen by a service request consists of two parts: (1) the processing time that the web server takes in response generation, which is determined by the processing capacity of the server; and (2) the traversal time of messages (in-

cluding the request and the response) across the network, which is determined by the quality of network transmission. In general, server optimization and network traversal time optimizations are complementary and both need to be done. However, while the former is somewhat better understood and one where the service provider has complete control, the latter, because of the use of public, shared networks, is typically less supported and requires more effort. Approaches proposed for optimizing server performance include upgrading hardware of the server and increasing the number of servers to form a Web-farm, to better utilizing existing server resources. However, the ultimate effectiveness of approaches in this category is somewhat limited because of the server becoming a potential bottleneck, considering the dramatic expansion of the Web including both the volume of available information and the number of end-clients. Approaches proposed to optimizing network traversal times have attempted to move the services, as well as the data, closer to end-clients. Proposed approaches include content delivery networks (CDNs) that deliver to the edge servers the whole website content, and web caching infrastructures that deliver to the edge servers either the data or a part of the application functionality, e.g. *servlet* execution for dynamic web content.

However, for the *data-centric* services that are the focus of this thesis, the dynamic nature of responses and the massive volume of involved data imply that such services do not benefit as much from caching and content distribution infrastructures, which are geared towards static web content. Instead, to improve scalability and client-perceived response times, such services have typically relied on manually-controlled replication strategies to create replicas at multiple network locations, and applied some minimal locality-aware techniques, e.g. Akamai-like DNS redirection,

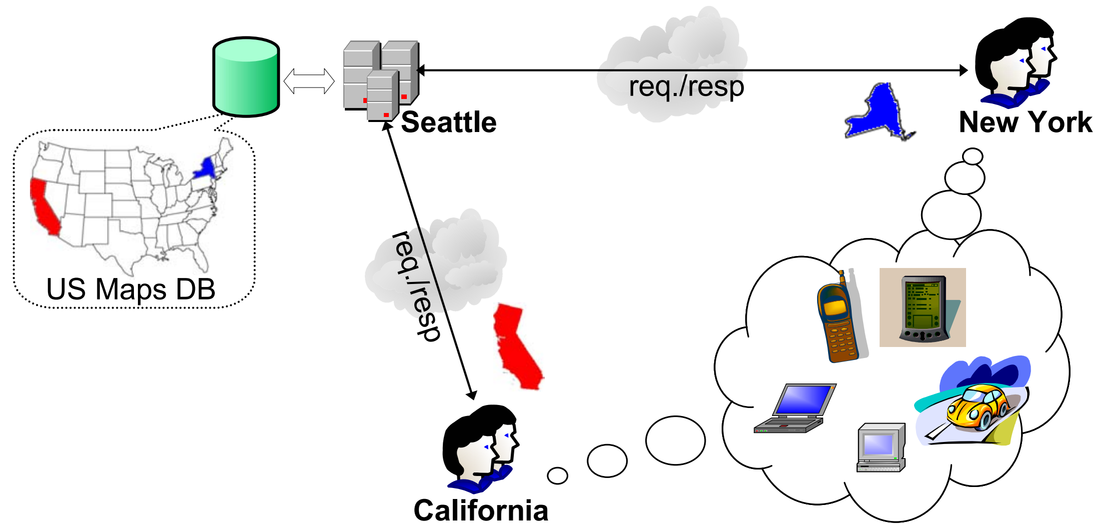


Figure 1.1: An illustrative example of data-centric network services in a wide-area network environment.

to redirect client requests to these replicas. Unfortunately, the inability of service providers to predict the exact effects of the manual replication strategies on service performance for a given client load pattern accounts for the fact that static provisioning is often used in a "best-guess" manner. As a consequence, static provisioning can be both wasteful in resources (if not all locations see the same load patterns for all data) and may not be responsive enough (if certain network regions or data items start seeing higher than anticipated demand).

An attractive alternative is to be able to dynamically tune decisions about the exact portions of the service data that need to be made available at various network locations. Fundamental to being able to make such decisions is the premise that service requests exhibit substantial locality in access patterns. Fortunately, we expect this property to be indeed true for the services of interest, for example one expects locality at the level of popular keywords and phrases for search services and at both

the network and data-space levels for imagery services. This property can be better understood by considering the following illustrative maps service. As demonstrated in Figure 1.1, for a maps service hosted at Seattle, the users in New York (NY) tend to request maps around NY vicinities, and similarly, the users in California (CA) tend to request maps around CA vicinities. Additionally, users might access the maps service using a variety of devices, demonstrating different access patterns in terms of the used device types. For instance, users with hand-held devices might tend to request text-based information or small, low-resolution maps, and users with desktops might tend to request large, high-resolution maps. A desirable replication strategy would create a replica nearby NY which contains only the service portion corresponding to the maps of the NY area, and a replica nearby CA which contains only the service portion corresponding to the maps of CA vicinities. The replicated data on the replicas might be further differentiated among the different types of client-devices. Subsequent client requests can then be redirected to the near replica to achieve better response times.

As the example shown, if locality does in fact exist, one can then do request monitoring or automatic decision making to improve service scalability and performance. The challenge here is how to detect and exploit such locality patterns in the face of dynamic client request loads at runtime, so that actions like service replication and request redirection can be taken in an on-demand manner to satisfy a variety of client QoS expectations. To address this challenge, one can imagine both centralized and decentralized schemes. A centralized scheme, because it sees information about all requests, may discover better locality patterns, but is likely more costly in terms of storage and computation, and less reactive to dynamic changes of both network metrics and client loads. The decentralized scheme, involving a network of nodes each

of whom only sees a portion of the requests, requires that the nodes have sufficient knowledge of the services and the back-end database to permit locality analysis. On the other hand, it permits easier monitoring of run-time information about network metrics and client quality expectations, which in turn helps derive a "good" service replication strategy. An additional advantage of the decentralized scheme is the fact that the standard structure of service requests enables service providers to offload locality analysis and replication decisions to a trusted third-party, which offers scale advantages by maintaining a distributed set of "service-neutral" network intermediaries.

The key questions that determine the utility of this overall approach include:

- whether locality does in fact exist in service access patterns?
- whether one can build an infrastructure to take advantage of this locality?
- whether this infrastructure yields the expected benefits?
- whether the infrastructure and the benefits are available in real-world wide-area networks with changing client load characteristics?

This dissertation investigates these four questions and answers each in the affirmative. We first justify the existence of locality in service access patterns with a thorough analysis of webtraces of two well-known Web services, and then report on our experiences in designing and implementing a decentralized network intermediary architecture, called *DataSlicer*, to improve performance and scalability for *data-centric* network services. We demonstrate that DataSlicer is able to yield the expected benefits by evaluating the performance of our infrastructure on a real-world network, the

PlanetLab testbed.

1.2 Thesis and Methodology

This dissertation investigates the thesis that in-network traffic inspection can help detect and exploit locality in service access patterns for data-centric network services, enabling improvements in service scalability and performance.

The question of whether locality exists in service access patterns motivated our investigation into the characteristics of workloads on two large-sized data-centric web sites, SkyServer and TerraServer. Our interest is in identifying the characteristics of Web Service requests in terms of *Temporal Locality*, *Spatial Locality* and *Network Locality*. We apply three techniques in our investigation: (1) unifying the multiple representations of service requests, (2) grouping the large-sized population of users, and (3) modelling the accesses to a large-sized, multi-attributed database.

The results from this investigation motivate the construction of an infrastructure which can take advantage of the existence of locality. As part of this research, we have designed and implemented the DataSlicer infrastructure, which leverages a distributed collection of network intermediaries, augmented with some service-specific knowledge and serving as application-level routers for message (request or response) relaying in the network, to dynamically detect service locality, and coordinate with the service to improve scalability and performance by on-demand replicating, at appropriate network locations, the portions of services ("data slices") that represent usage locality.

To achieve the desired goal, DataSlicer relies on four techniques:

- Request inspection and locality detection on distributed network intermediaries.
- Construction of an oriented overlay network to cluster client requests for data-centric network services.
- Load-balancing that routes clients requests through the overlay according to the network status, and service replication that on-demand replicates on network intermediaries a small portion of the service data best representing the usage locality.
- Robustness schemes that ensure the stability of the infrastructure in wide-area network environments.

These techniques are implemented as integral parts of the DataSlicer infrastructure, which provides a hosting platform for data-centric network services. A new service participates in the infrastructure by first registering its service-specific information with an origin server maintained by DataSlicer. The origin server then propagates this information to the participating network intermediaries. Upon receiving such information, each network intermediary constructs the corresponding service handler which takes responsibility for overlay construction, request inspection, locality detection, load-balancing and service replication. After initialization, network intermediaries publish, using UDDI or a similar protocol, their abilities to receive client requests for the service. Clients access the service by first locating the closest network intermediary and sending it their requests.

DataSlicer is able to automatically construct underlying oriented overlay networks, which cluster client requests for data-centric services using multiple application-specific

metrics. Client requests get routed through the overlay network and inspected at various network intermediaries on the way to an origin server/service replica. DataSlicer exploits a dynamically growing data structure to permit locality detection across multiple dimensions at different levels of granularity. The detected locality information is leveraged by our infrastructure to determine appropriate replication strategies to create replicas in a wide-area network environment with minimum cost. The load-balancing technique allows DataSlicer to re-balance the requests distributed in the overlay network towards multiple service replicas to obtain better performance. The load-balancing and service replication techniques cooperate with each other to provide QoS assurance to end-clients. Finally, our robustness scheme ensures that DataSlicer continues to function stably despite the existence of various node and network outages and inaccuracies in network metric-measurements.

The benefits of DataSlicer are evaluated by implementing and deploying our infrastructure on the PlanetLab network, and running it on configurations that involve 200 nodes or more for extended periods of time coping with network outages, node failures, etc. Our experiments with a synthetic maps service show that (1) DataSlicer's oriented overlay construction scheme provides good clustering properties; (2) DataSlicer is able to detect service locality among client requests; (3) DataSlicer can take appropriate actions including load-balancing, service replication and request redirection to satisfy client QoS requirements in situations where they are not met; and (4) DataSlicer is robust to a variety of network outages and node failures and can adapt itself to provide stable behavior.

1.3 Contributions

The high level contribution of this thesis is the validation of existence of locality in client access patterns for data-centric network services and an integrated set of techniques that permit a novel caching infrastructure to leverage such locality information to improve performance and scalability for data-centric network services in wide-area network environments.

The results of our investigation of workloads of two well-known Web services show that both workloads exhibit a high degree of locality in terms of *Temporal Locality*, *Spatial Locality* and *Network Locality*, which imply that it is possible to replicate a small portion of the service to a small number of network locations, so as to satisfy a large fraction of client QoS expectations.

To take advantage of the locality present in the service access patterns, DataSlicer exploits four techniques to dynamically discover the locality patterns and take appropriate actions to satisfy a variety of client QoS expectations. The main contributions of these techniques are described below:

In-network request inspection and locality detection. This technique assumes a semantic structure for the service, namely that its requests can be treated as accessing relations in a logical multi-attribute data space, and assumes that service providers will provide the service-specific information, including the service interfaces, the meta-information about the backend database, and the mapping scheme that transforms a request into a region of the targeted data space. With the necessary service-information available, DataSlicer routers are able to recognize the service requests

from the underlying traffic flows, inspect the requests to extract the service access information, and aggregate these requests to dynamically infer the service access locality across several dimensions: Temporal Locality, Spatial Locality and Network Locality.

Oriented overlays construction and maintenance. This technique is designed with the awareness that service locality at the network level can only be discovered if related requests flow along similar paths in the network. To facilitate this goal, the oriented overlay construction scheme defines common paths to route through the network the requests that exhibit common application-specific clustering preferences, e.g., network proximity, network bandwidth requirements, etc. Furthermore, such common paths do not adversely affect client-perceivable metrics, e.g., the path latencies.

Load-balancing and service replication. In our constructed oriented overlays, client requests might be routed through multiple paths to reach one or more service replicas, with different paths exhibiting different performance characteristics. DataSlicer's load-balancing technique allows each intermediary to make local decisions on how to distribute requests to different paths in the overlay to achieve globally balanced traffic flow, and therefore to be able to satisfy client QoS requirements without requiring any service replication. In situations where load-balancing is insufficient to meet client QoS expectations, DataSlicer has to take service replication action. The underlying problem here is one of *minimum cost replica placement*, i.e., deciding which regions of the data space to replicate at which locations (given a set of replica sites) to minimize overall replication cost, while satisfying a predefined client QoS requirements.

Given our problem setting, we assume a fairly general cost function where the cost of a replication is monotonically determined by the volume of involved data, the distance of the service replica from the origin server, and the cumulative cost of consistency maintenance. The above problem is NP-hard, and we develop a heuristic solution that satisfies the client QoS requirements *independently* for each of the data space regions.

System robustness. DataSlicer runs robustly in large-scale, dynamic wide-area networks, coping with network failures and outages, e.g., node crashes, connection breaking, packet losses. Such wide-area network environments are usually very dynamic, resulting in the network metrics measured at the network intermediaries being very erratic. DataSlicer overcomes these characteristics of the environment by employing the following mechanisms to achieve system robustness: (1) continuous monitoring and repair of liveness of nodes, connections and client requests; (2) use of token-based link-level flow control to ensure that client requests don't get dropped in the middle due to buffer overflow; (3) use of non-blocking/asynchronous techniques for communicating between network intermediaries to provide scale performance at network intermediaries for high-load situations; and (4) smoothing out the inaccuracies/noise of the metric-measurements to provide stable performance.

Additional contributions include implementing these techniques in the context of a working infrastructure, and evaluating them using a synthetic maps service on the PlanetLab network.

1.4 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 describes a system overview of DataSlicer architecture and gives a formal description of the problem addressed by this thesis. Chapter 3 provides background information and an overview of related work. Chapter 4 presents our investigation on locality patterns in data-centric network services and describes the techniques for dynamically detecting such locality patterns using in-network request inspection. Chapters 5, 6, and 7 introduce techniques for addressing the challenges of building our infrastructure. Chapter 8 presents an evaluation of the performance of DataSlicer using a synthetic maps service on PlanetLab and discusses the lessons we learned from designing, implementing, and deploying our network service architecture. Chapter 9 concludes this thesis and outlines possible directions for future research.

Chapter 2

Problem Description

This chapter describes the characteristics of data-centric network services and introduces a representative example service that will be used throughout this document to explain and evaluate the four techniques introduced in Section 1.2. It describes an abstract service architecture for improving performance and scalability for data-centric network services and highlights the challenges that such an architecture must address. It also describes the DataSlicer infrastructure, which is a concrete realization of the abstract service architecture.

2.1 Data-centric Services

Data-centric network services are defined as the services that involve a massive volume of data at one or a small number of back-end database(s) (virtual or physical), and satisfy client requests by dynamically generating responses using data from one or more of these databases. We assume that these services possess the following char-

acteristics:

- A well-known structure for service requests and responses: the interactions between clients and the services are usually conducted on top of a standard protocol such as HTTP or SOAP, where a request to a service is either presented as a query in a HTTP request (in the GET URL or the content for a POST) or a “Body” element in a SOAP request.

- A semantic view of a service request: although the overall volume of data at the back-end database is massive, an individual client request typically touches only a very small portion of the overall data space; such a portion is often identified by the request parameters, either explicitly or by using some service-specific mapping.

- Read-mostly access pattern: most of client requests are read-only; the service providers are responsible for the data maintenance of the back-end databases, including data update and insertion; these operations usually occur at a much lower-frequency, compared with client access rates.

- Existence of service usage locality: the client accesses against the backend database demonstrate locality patterns across one or more of the following dimensions: time epoches, data space, network regions.

These characteristics permit one to build a semantic structure of service usage in terms of the regions of a logical “view” of the underlying database (defined say in terms of the exercised database attributes) accessed by a group of requests, where an individual client request can be treated as a point in this multi-dimensional data space. They also

permit the creation of in-network replicas, which contain a small portion of service data representing service usage locality, which achieve reuse benefits.

2.1.1 Motivating Example: Maps Service

A maps service, such as Microsoft's MapPoint Web Service, provides to its end-clients location-based services such as maps, driving directions and proximity searches, and allows advanced developers to integrate such services into their applications or business processes. The service provider usually hosts the service on one (or a small number of) web site(s), and stores the raw data of the maps at a back-end database. The service responds to a request by querying against the back-end database to fetch the corresponding raw data and generating a response based on the fetched raw data. Clients of the maps service are usually geographically distributed, use a variety of devices to access the service across a wide-area network environment, and have diverse QoS requirements.

The usage of a maps service possesses the following important characteristics:

- The interactions between clients and the service are conducted on top of a standard protocol like HTTP or SOAP using well-formatted message structures.
- An individual client request usually touches only a small region of the back-end database. For example, a request for a street level map around a particular location would only require access to the portion of the database that contains the relevant information.
- Client requests are read-only requests. The maps data maintenance is usually conducted only by the maps service provider.

- Requests to a maps service demonstrate a high degree of locality in service access patterns. For example, one would expect to see service locality in terms of network proximity, i.e., people who reside in the same geographical area might tend to request maps in and around their living regions. Additionally, one might also expect to see locality in terms of client-device commonality, e.g., clients using hand-held devices are likely to be interested in text-based information or small, low-resolution maps, while clients using desktops are likely to be interested in large, high-resolution maps.

Although the interactions between clients and a maps service follow a well formatted message structure, the client requests could demonstrate a "polymorphic structure", i.e. there could be different requests (in terms of the invoked interfaces and the supplied parameters) that access the same map image. Taking Microsoft's MapPoint service as an example, to retrieve a map, a request can take three different kinds of parameters as input: *ViewByHeightWidth*, *ViewByScale* and *ViewByBoundingRectangle*, with another parameter indicating the image size shown on screen. The *ViewByHeightWidth* determines a map by a center point (in latitude and longitude) and the height and width of maps in *DistanceUnit* (in miles or kilometers); the *ViewByScale* determines a map by a center point and a map scale; and the *ViewByBoundingRectangle* determines a map by the latitude and longitude coordinates that represent the southwest and northeast corners of a minimum bounding rectangle. These three different parameter settings are interchangeable by applying a simple computation to convert one to another. It is desirable that such polymorphically structured messages can be treated uniformly to facilitate request inspection and achieve reuse benefits.

2.2 Problem Statement

2.2.1 Abstract Service Architecture

The abstract service architecture consists of intermediate routers, origin service and service replicas, and end-clients. Clients access the service via the routers which can relay service requests/responses between clients and origin service/service replicas, and most importantly, can inspect underlying traffic to detect locality in service access patterns. Algorithms run on the routers to (1) construct the oriented overlays for request relaying; (2) leverage the detected locality information to determine appropriate service replication actions, i.e., which data space regions need to be replicated and where to create replica; and (3) distribute the traffic in the network to maintain good performance. This architecture assumes that service provider is responsible for data management at service replicas, including replica creation and replacement, and data consistency between replicas and the origin service. The routers merely identify the data space regions that ought to be replicated, notify the service provider with its suggesting service replication actions, and routes client requests to such replicas once they get created.

2.2.2 Challenges

For the abstract service architecture described above to yield utility, several challenges need to be addressed:

- Does service usage locality exist in accesses to data-centric network services? If exists, can one exploit the information in client requests (query string, parameters for invoked service interface, etc.) to infer the locality models?

- How to organize the participating routers into an overlay network in a way that permit aggregation of client requests according to involved application specific clustering preferences?

- How to inspect client requests at the intermediate routers to extract information about service accesses, and how to monitor system performance to determine whether or not client QoS expectation are being met? And how to analyze such information to efficiently detect the usage locality?

- How to relay requests through the constructed overlay network to satisfy client QoS requirements without incurring service replication? And in case that replication is required, how to determine a practical and efficient replication strategy?

- How to maintain system robustness in face of network faults and outages? And how to maintain the stability of the system in face of dynamic changes of network metrics?

Each challenge is briefly discussed in the following sections.

2.2.3 Existence of Service Usage Locality

On one hand, data-centric network services involve a massive volume of data; on the other, service replicas are usually resource-constrained and replicating data across a wide-area network is costly. These factors prevent data-centric network services from replicating site-scale contents at the replica nodes. Ideally, one would like to replicate at appropriate replica nodes only small portions of the service data which have high access rates, such that the overall system performance can be significantly improved.

Key to such a solution is whether such small service portions exist, and if they exist, how to define such service portions in terms of time epochs, data space, and network regions. In other words, one needs to understand to what extent data-centric network services exhibit usage locality across the above dimensions.

2.2.4 Construction of Oriented Overlay Networks

The abstract service architecture offers benefit only when it is able to dynamically detect the service access locality at different levels of network granularity. The underlying challenge is how to group requests originating from geographically distributed clients at various routers such that service access locality can be detected at these routers. Clearly, if requests are routed among routers in a way that makes locality detection hard, the abstract service architecture can not take replication and redirection actions. Similarly, if all requests are forced to go directly through a central server, not much benefits can be expected even if requests exhibit locality at the network level and in the targeted data space. The first observation suggests that client requests must be routed through the network in a way that permits intermediate locations to identify and hopefully exploit request similarity. The second observation requires that these locations be distributed across the network as opposed to being clustered around the origin server(s). The problem is how to tradeoff between these two considerations.

2.2.5 Request Inspection and Locality Detection

The challenge consists of two parts: in-network request inspection and service access locality detection. The former requires routers to have knowledge about the service interfaces, the meta-information about the backend database, and how to map a re-

quest into a region of the targeted data space. The latter requires routers to be able to store client requests in an efficient and compact way and aggregate these requests to infer the service access locality across multiple dimensions at runtime.

2.2.6 Load-Balancing and Service Replication Problem

In the overlay network described above, we call a connection between a router and a service replica (including the origin server) a *path*. A path consists of one or more *links*, where a link is a connection between two consecutive network entities. A router could potentially have multiple paths leading to the origin server or the service replicas. These paths could have different metrics in terms of network latency, bandwidth and node CPU utilization, resulting in different performance seen by the requests forwarded along these paths. Therefore, a router can provide good performance by taking advantage of the asymmetric performance of its outgoing paths, e.g., by shifting requests from paths that are seeing poor performance onto those that are currently experiencing fewer delays. However, doing so requires the router to have knowledge about the performance of these paths, and to ensure such "local" not inadvertently impair the performance on other routers since paths in the overlay network might overlap with each other.

Service replication is costly because it usually involves offloading a significant amount of data across a wide-area network and the subsequent maintenance of data to ensure consistency with the origin service. Therefore, service replication action should be employed only when the load balancing actions prove insufficient. When employed, one would like to ensure that replication actions incur as low cost as possible. The underlying problem can be stated as follows: given a collection of routers or-

ganized into an overlay, a distribution of client accesses at the entry routers, a database that has been split into multiple partitions, find a minimum cost replica placement so as to satisfy a predefined client QoS requirement. The problem is NP-hard, so the interest is in developing appropriate heuristic approaches that work well in practice.

2.2.7 System Robustness

In a wide-area network environment, nodes might crash at any time and take arbitrarily long times to recover. Additionally, network connections might break unpredictably and packets transmitted through the network might get lost. The service architecture should be able to recover itself from such network failures and outages. Furthermore, since the load-balancing and service replication techniques are triggered whenever the performance measured on a router drops below a predefined threshold, these measurements must be robust over expected dynamic changes of metrics in wide-area network environments, which can otherwise lead to unnecessary actions.

2.3 DataSlicer

DataSlicer is a concrete realization of the abstract service architecture. It serves as a prototype and testbed in the context of which solutions to the above challenges are developed and evaluated. Figure 2.1 shows an overview of the DataSlicer architecture, which consists of service-neutral network intermediate routers that interact with one or more service replicas. The DataSlicer architecture makes some structural and simplifying assumptions:

- We assume that the router network and the replica network are maintained sep-

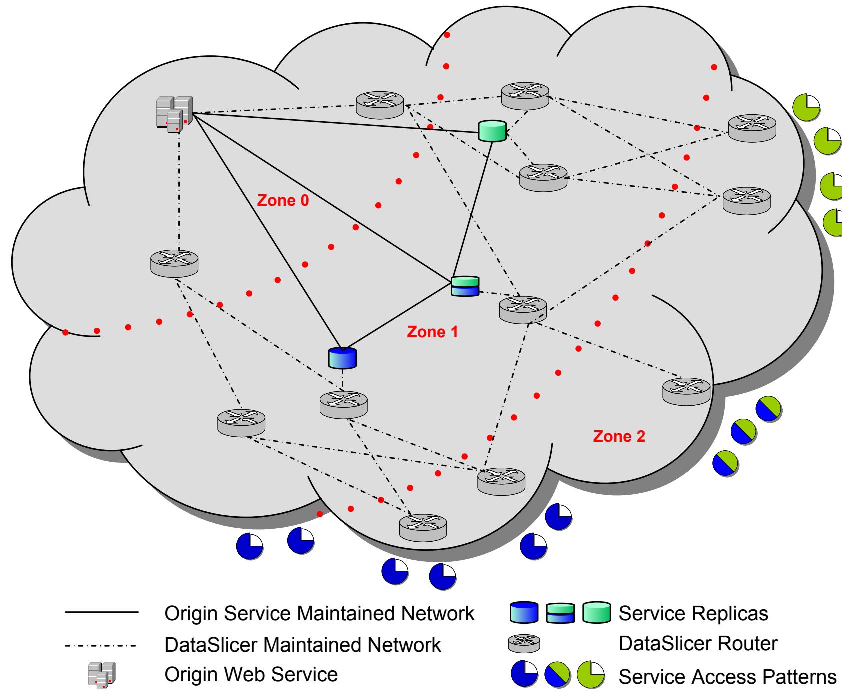


Figure 2.1: An overview of DataSlicer architecture

arately. Specifically, we assume that the service replicas are maintained by the service providers on their own; this permits the service providers to offload the service functionality and the portions of the associated data on-demand onto the replicas without security concerns. On the other hand, the portions of data being offloaded onto or removed from a specific replica are suggested by our distributed service replication algorithms that run on the router network.

- We assume that both of these networks are trusted. It would be straightforward to extend the interaction protocols to deal with situations where this assumption may not hold.

- One of these replicas is assumed to always be active, and acts as the *origin server*.

Clients connect to distinguished routers called *entry routers*; the routers that connect to a service replica are called *exit routers*; and we refer to the remainder as *intermediate routers*. A router can be an entry router, an exit router and an intermediate router at the same time.

The routers are organized in an overlay network and relay requests and responses between clients and service replicas. Such an overlay network is constructed in a way that allows routers that see related client requests for the service, i.e., requests that share similarity in terms of application-specific metrics, to be clustered together. The service similarity shared by the client requests can be determined by many metrics including network proximity, URL-level locality, or client-device commonality.

To host a service, DataSlicer requires the active involvement of the service providers during initialization where the service providers register their services with the architecture. Registration includes information about service interfaces, the underlying logical data space for the service and the mapping scheme that maps a service request into a region in the targeted data space (the two elements of the semantic structure discussed above), and the desired performance metrics (we focus on service response time). The registration step also identifies the entry routers for the service, which publish, using DNS redirection or a similar protocol, their ability to receive service requests. Post-registration, DataSlicer starts routing requests for the service through an appropriately constructed overlay network, inspecting the requests enroute and taking necessary actions to satisfy the client QoS requirements.

In the rest of this section, we briefly discuss the functionality provided by a DataSlicer router, the interactions among the router network, and the interactions across the router network and the replica network. The discussion about the detailed design of the router to support such functionality will be deferred in the later chapters.

2.3.1 Router Functionality

As an application-level router, a DataSlicer router supports the basic routing functionality to relay service requests/responses through the underlying network. Beyond that, the routers are enhanced with additional functionality that allows them to:

- actively participate in the construction of the underlying overlay network
- perform in-network inspection of client requests to infer service locality patterns
- monitor the dynamic network states including network latency, network bandwidth, and node liveness
- take necessary actions to satisfy client QoS requirements
- recover from system faults and outages

To participate in an overlay, a router first associates itself with the closest origin server, and through it with a nearby service replica (if available) and one or more parent routers. Service information retrieved from the origin server results in the instantiation of a service handler, which is built up out of a number of functional components (see Figure 2.2). These components work together to support the functionality described above, guided by supplied auxiliary policies that govern resource sharing and cost-benefit tradeoffs (used by the service replication algorithms).

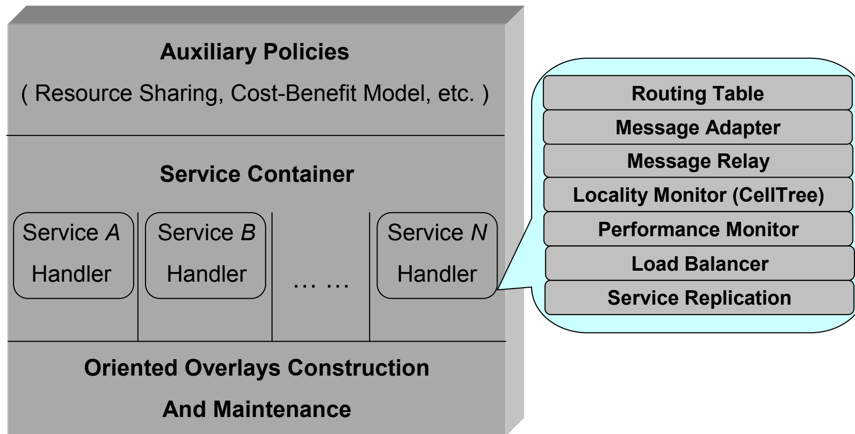


Figure 2.2: Functional Components of DataSlicer Router

Oriented Overlays Construction and Maintenance. This underlying component implements the functionality of construction and maintenance of router network to permit aggregation of client requests. The details are described in Chapter 5.

Routing Table. This component is responsible for storing the information about the connections between a router and its parent routers resulting from the construction of the oriented overlays. Each connection is called a *link*.

Message Adapter. This component is responsible for transforming a service request into a description of a small region in the data space accessed by that request. The transformed requests is passed to *Locality Monitor* for aggregation.

Message Relay. This component is responsible for transmitting a service message (request/response) through the overlay network. It wraps a message when the message is injected into the DataSlicer infrastructure, and unwraps a wrapped message when

it leaves the infrastructure. Additionally, this component provides a link-based flow control in message relaying to avoid buffer-overflow problem, described in Chapter 7.

Locality Monitor. This component leverages a dynamic data structure, called *Cell*, to aggregate groups of the client requests to detect service usage locality on different levels of data granularity within different time epoches. The detected locality information is used by the *Service Replication* component to determine service replication actions.

Performance Monitor. This component is responsible for measuring the performance of the system as seen by requests traversing through the router. The performance measurements are used in the *Locality Monitor* to aggregate the performance seen by a group of requests. They are also used to compute statistics of performance of each of the links; the latter are used by the *Load Balancer* and *Service Replication* modules to trigger appropriate actions.

Load Balancer. This component is responsible for redistributing a group of requests to multiple parent routers to improve average request performance. It is triggered upon the detection of any violations of client QoS requirements for a group of requests, and attempts to shift the traffic from the links that are seeing poor performance to the ones that are seeing good performance.

Service Replication. In situations where load balancing is insufficient, a router can coordinate with other routers to trigger a service replication request, resulting in a service portion being replicated at some appropriate replica. The router then can redirect

(explicitly or implicitly) subsequent client requests to that replica to satisfy client QoS requirements. The distributed algorithms to determine the service replication actions are implemented in the *Service Replication* component.

In addition, all components implement one or more of the robustness techniques described in Chapter 7 to ensure stable system behavior in dynamic network environments.

2.3.2 Replica Functionality

After registration, the only direct interaction between DataSlicer routers and a service is for initiating replication actions. Depending on the outcome of the replication algorithm, a router may request a replica be created for a certain portion of the service that is seeing unsatisfactory performance.

A replication request is sent from a router to its associated replica (if the router does not have an associated replica, the replication request can be forwarded to one of its parents which will then perform the replication task on behalf of the originating router). The replica in turn forwards the replication request to the origin server, which coordinates transfer of the corresponding portion of the service data to the replica (possibly directing the replica to a different location). After the replication process is completed, the replica notifies all of its associated routers, which update the corresponding replication information stored in the *Locality Monitor* component and propagate this information to all of their descendants in the overlay. Similarly, a router that first associates itself to a replica can enquire the replica about the service portions that are replicated there and propagate this information to its descendants.

Service replicas may run into resource constraints and need to evict an existing

replicated service portion. In this case, the replicas are responsible for notifying the associated routers about such changes. Similar to replica creation responses, the routers update their service replication status appropriately and propagate this information to their descendants.

Note that the semantics of how the replica is created and reclaimed and how the replicas are kept consistent with each other are left entirely up to the service providers. The DataSlicer architecture merely identifies the data space regions that ought to be replicated, and routes client requests from network regions that can most benefit from them to such replicas once they get created.

2.4 Summary

This chapter has described the overview of an abstract service architecture to improve performance and scalability for data-centric network services, discussed the challenges that need to be addressed in such an approach, and presented a prototype instantiation of this architecture, called *DataSlicer*. Existing systems that partially address these challenges are described in Chapter 3, which also identifies their shortcomings. The solutions developed in this research to address these shortcomings are presented in Chapters 4, 5, 6 and 7, and evaluated in Chapter 8.

Chapter 3

Background and Related Work

This chapter is divided into two parts. The first part introduces the necessary background to understand the solutions described in this document. The second part describes previous work related to the DataSlicer infrastructure. In this part, we restrict our attention to several representative system-level approaches that have been designed to improve the performance and scalability for network services. Discussion of technique-level approaches that are related to the work in this thesis is deferred to Chapters 4, 5, 6, and 7, which describe the particular techniques and make a possible detailed comparison.

3.1 Network Services and Related Technologies

DataSlicer has been designed to host a variety of network services in a wide-area network environment. Such services rely on a set of standard technologies to publish their access points, define the structured service interfaces, and allow sophisticated in-

teractions between clients and the services. These technologies include standard web protocols such as *HTTP*, *XML-RPC* and *SOAP*, more advanced programmable service architectures such as *XML Web Services* and protocols such as *WSDL* and *UDDI*, and auxillary document processing techniques such as *XSL* and *XSLT*. For completeness, this section briefly discusses these technologies. Readers already familiar with them may wish to skip to Section 3.2.

3.1.1 Transport Protocols

HTTP. *HTTP* [12], standing for *H*yper*t*ext *T*ransfer *P*rotocol, is an application-level protocol for distributed, collaborative, hypermedia information systems. It is a generic, stateless protocol which can be used for many tasks beyond its use for hypertext, such as name servers and distributed object management systems, through extension of its request methods, error codes and headers. A feature of HTTP is the typing and negotiation of data representation, allowing systems to be built independently of the data being transferred.

HTTP employs connection-oriented TCP/IP protocols to support information exchange between clients and servers. A typical HTTP transaction involves the steps below:

- Connection: A client opens a connection to a server.
- Query: The client requests a resource controlled by the server.
- Processing: The server receives and processes the request.
- Response: The server sends the requested resource back to the client.

- Termination: The transaction is done and the connection is closed.

XML-RPC. *XML* [13], standing for e*X*tensible *M*arkup *L*anguage, describes a class of data objects called XML documents and partially describes the behavior of computer programs which process them. As a markup language that allows users to define their own customized tags, XML is platform-independent, language-independent and media-independent. Therefore it is easier for systems in different environment to exchange information using XML. The universality of XML makes it a very attractive way to communicate information between data-centric Web applications, and handle operations such as database exchange, and distribution of processing and business transactions. Furthermore, XML together with XML Namespaces [14] and XML Schemas [15], provides useful mechanisms to deal with structured extensibility in a distributed environment. XML Namespaces provide a simple method for qualifying element and attribute names used in XML documents by associating them with namespaces identified by IRI references, where "an IRI reference is a string that can be converted to a Uniform Resource Identifiers(URI) reference by applying a set of rules" [14]. XML Schemas express shared vocabularies and allow machines to carry out rules made by people by providing a means for defining the structure, content and semantics of XML documents.

XML-RPC [16] is a Remote Procedure Calling protocol that allows software running on disparate operating systems, running in different environments to make procedure calls over the Internet. It uses HTTP as the transport and XML as the encoding: an XML-RPC message is an HTTP-POST request where the body of the request is in XML; a request leads to the execution of a procedure on the server with the value

returned in response also formatted as an XML message that is sent back to the requester.

SOAP. *SOAP* stands for *S*imple *O*bject *A*ccess *P*rotocol. As stated in the specification [17], "SOAP is a lightweight protocol intended for exchanging structured information in a decentralized, distributed environment. It uses XML technologies to define an extensible messaging framework providing a message construct that can be exchanged over a variety of underlying protocols. The framework has been designed to be independent of any particular programming model and other implementation specific semantics." SOAP defines a message structure consisting of an optional *header* and a mandatory *body*. The SOAP header provides a flexible mechanism for extending a message in a decentralized and modular way. Typical examples of extensions that can be implemented as header entries are authentication, transaction management, etc. SOAP provides an XML-RPC-like request/response protocol.

The network entities that are responsible for moving SOAP messages between internal and external networks are called *SOAP Routers*. A SOAP router attempts to make a complex route (multi-hops, multi-protocols) look like a simple route from the view of the initiator by abstracting the latter from knowing the physical address of the ultimate destination and hiding the underlying transport protocols. Features for SOAP routing are described in WS-Routing [18] and WS-Referral [19] protocols.

3.1.2 Network Service Architectures

In general, to allow clients access a service, the service provider needs to define the interfaces for the service and publish them so they are available to the clients. The

interactions between clients and the service can then be conducted over the object-model-specific protocols used in component-based technologies, or the information transfer protocols described above. Due to compatibility and security constraints associated with the former, the web is moving toward an architecture where the latter are extensively used as standard protocols to exchange information between clients and services. Within this context, emerging techniques such as *Web Services*, *WSDL* and *UDDI* programmable infrastructures support service publishing, discovery and integration, enabling programmatic interactions in the web.

In this section, we briefly describe these emerging techniques. Note that the DataSlicer approach described in Chapter 2 is not limited to XML Web Services, but works with general network service architectures as long as they make use of a standard transport protocol such as *HTTP*.

XML Web Services. A *Web Service* [20] is a programmable application logic component accessible using standard Internet protocols. Like other component technologies, Web Services separate interface and implementation. The difference between Web Services and current component technologies is that they are not accessed using object-model-specific protocols, such as Distributed Component Object Model (DCOM), Remote Method Invocation (RMI), or Internet Inter-ORB Protocol (IIOP). Instead, they are accessed via standard, widely accepted web protocols and data formats. In most cases, the web protocol is HTTP and the data format is XML. Web Services that use XML as their data format are referred to as *XML Web Services*.

XML Web Services are becoming the fundamental building blocks in the move towards distributed computing in the World Wide Web, with a variety of sophisti-

cated Web applications being constructed from preexisting XML Web Services without worrying about where they reside or how they were implemented. XML Web Services are characterized by the following properties:

- XML Web Services expose useful functionality to Web users through a standard Web protocol, in most cases, the protocol used is SOAP [17]
- XML Web services provide a way to describe their interfaces in enough detail to allow a user to build a client application to talk to them. This description is usually provided in an XML document called a Web Services Description Language (WSDL) document [21]
- XML Web services are registered so that potential users can find them easily. This is typically done using a protocol called Universal Discovery Description and Integration (UDDI) [22]

WSDL. *WSDL* stands for *W*eb *S*ervices *D*escription *L*anguage, an XML-based language used to define Web Services and describe how to access them. With WSDL, Web Services can be described as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information. The operations and messages are described abstractly, and then bound to a concrete network protocol and message format to define an endpoint. In most cases, the network protocol is HTTP and the message format is SOAP.

A WSDL file is essentially an XML document, therefore it is readable and editable by a human being. But in most cases, it is generated and consumed by software. There are several tools available to read a WSDL file and generate the code required

to communicate with an XML Web Service. The tools can also generate WSDL files from existing programming interfaces.

UDDI. *UDDI* stands for *U*niversal *D*iscovery *D*escription and *I*ntegration [22], and is an industry specification for publishing and locating information about Web services. It defines an information framework that enables users to describe and classify their organizations, the services offered, and the technical details about the interfaces of the Web Services exposed. UDDI also defines a set of Application Programming Interfaces (APIs) that can be used by applications and services to interact with UDDI data directly. A UDDI directory entry is an XML file that describes a business and the services if offers. There are three parts to an entry in UDDI directory: "white pages" describe the company information; "yellow pages" include industrial categories based on standard taxonomies; "green pages" describe the interface to the service exposed. UDDI uses a document, called Type Model or tModel, to define services. In most cases, the tModel contains a WSDL file that describe a SOAP interface to an XML Web Service, although it is flexible enough to describe almost any other service.

Auxillary Document Processing Techniques. *XSL* stands for e*X*tensible *S*tylesheet *L*anguage. In order to display XML documents, it is necessary to have a mechanism to describe how the document should be displayed. XSL is the preferred style sheet language of XML, performing the same role as Cascading Style Sheets (CSS) in HTML. XSL consists of two parts: a method for transforming XML documents and a method for formatting XML documents. The former, called XSL Transformations (*XSLT*), is the part of XSL that is used to transform an XML document into another XML doc-

ument, or another type of document that is recognized by a browser, like HTML and XHTML. XSLT can also add new elements into the output file, or remove elements. It can rearrange and sort elements, and test and make decisions about which elements to display, and a lot more. A common way to describe the transformation process is to say that XSLT transforms an XML source tree into an XML result tree. The second part of XSL is the process of turning the result of an XSL transformation into a tangible form for the reader or listener.

3.1.3 The Underlying Networks

To evaluate the performance of our caching infrastructure, we have implemented and deployed DataSlicer on two kinds of networks: one is a simulated network using the *Click* modular router software [23], the other is a real wide-area network, the *PlanetLab* network [24].

Click is a software router developed by MIT LCS's Parallel and Distributed Operating Systems group, and with ongoing contribution from Mazu Networks, the ICSI Center for Internet Research, and UCLA. It provides a new software architecture for building flexible and configurable routers. A Click router is an interconnected collection of modules called *elements*; elements control every aspect of the router's behavior, from communicating with devices, to packet modification to queueing, dropping policies and packet scheduling. Several features make individual elements more powerful and complex configurations easier to write, including *pull* connections, which model packet flow driven by transmitting hardware devices, and flow-based router context, which helps an element locate other interesting elements.

The *PlanetLab* network is a geographically distributed platform for deploying,

evaluating, and accessing planetary-scale network services. It is a shared community effort by a large international group of researchers at over 300 sites in more than 25 countries. PlanetLab users who wish to deploy applications acquire a *slice*, which is a collection of virtual machines (VMs) spread around the world. The VMs are implemented on physical machines by some operating system mechanism, and controlled by another entity, the *node manager*, which is responsible for creating and destroying slices. There are also special *infrastructure slices* which perform essential functions on each node (such as providing a local site administrators interface to the node). Collectively, the node managers and infrastructure services, together with the (currently centralized) account management and node installation functions, form the control plane of PlanetLab.

3.2 System-level Approaches

Existing system-level approaches proposed to improve performance and scalability for network services fall into two broad categories: component-based service integration systems, and web caching and replication systems. To our knowledge, there does not exist an approach, either in the component-based service integration system category, or from the web caching and replication systems category, which addresses the challenges described in Chapter 2 entirely. To understand where existing approaches fall short, one can view the problem of building a scalable network services infrastructure along three dimensions:

- The first dimension describes how complex is the cacheable content (content complexity). Possible points along this dimension include static content, dy-

namic content, and more sophisticated structured content (e.g. semantic regions of a database).

- The second dimension describes the "reach" of the infrastructure, and ranges from stand-alone proxy caching to more complex networks of cooperating caches.
- The third dimension describes the guarantees provided by such an infrastructure, and span "best-effort" assurances at one end to more complex QoS assurances involving request latency and quality at the other.

Our applications of interest, as exemplified by the motivating example in Chapter 2 fall within a region defined by the most sophisticated functionality along each of these dimensions — they work with structured content, involve a network of cooperating network entities, and require QoS assurances. Along this latter dimension in particular, very few approaches (an exception includes [25]) have looked at providing any QoS assurances over wide-area networks.

3.2.1 Component-based Service Integration

Increasingly, scalable Web applications are being constructed by integrating service components that are individually developed. The web services architecture is one example of such a trend. The scalability and efficiency for such applications is typically achieved by choosing and deploying these components appropriately across the network. Proposed approaches include DCE [26], DCOM [27] and CORBA [28, 28], which rely on static component linkages, as well as CANS [29], Ninja [30], OGSA [31], and PSF [32] which aim at advocating a dynamic model for component

integration at run-time. In this section, we discuss two representative approaches: OGSA and PSF.

Open Grid Services Architecture. The Open Grid Services Architecture (OGSA) [31] explores the advantages of integrating Grid technologies and Web services. This architecture defines semantics of Grid services and related mechanisms for creating, naming, and discovering transient Grid service instances; provides location transparency and multiple protocol bindings for service instances; and supports integration with underlying native platform facilities. OGSA also defines, in terms of WSDL interfaces and associated conventions, mechanisms required for creating and composing sophisticated distributed systems, including lifetime management, change management, and notification. Service bindings can support reliable invocation, authentication, authorization, and delegation, if required.

Partitionable Services Framework. The Partitionable Services Framework (PSF) [32] addresses, within an OGSA-like context, the absence of adaptivity to the performance and security characteristics of the heterogeneous environment. PSF proposes an approach which enables services to be flexibly assembled from multiple components. The approach also facilitates transparent component migration and replication on demand at locations closer to end-users. The framework relies on four key elements: (1) a declarative specification of the service components; (2) a monitoring module; (3) a planning module, which steers the deployment to accommodate underlying environment characteristics; and (4) a deployment infrastructure which realizes the service deployment. PSF integrates a decentralized trust management and access con-

trol system called *dRBAC* [33] with a programming and run-time abstraction called *views* [34], to provide a unifying mechanism for cross-domain authentication and authorization, and support of single sign-on and fine-grained access control.

Although the component-based service integration infrastructures discussed above can gracefully handle service naming, discovery and dynamic assembly, as well as resource allocation, authentication and authorization, they do not address all of the challenges pointed out in Chapter 2. Components are usually viewed as black boxes, hiding internal computation and information about the usage of internal data. So it is difficult for the infrastructure to retrieve information about component usage patterns, which is needed in order to reason about the cost of interaction between components as well as cost of component replication, if applicable. Furthermore, component deployment in such systems has typically been determined in a centralized fashion. In addition to the scaling challenge this presents, centralized schemes may also make it difficult to detect and therefore exploit service locality in a dynamic network environment.

3.2.2 Web Caching and Replication

The other group of related research efforts aim to provide a scalable, high-performance Web Services infrastructure by means of service offloading. Generally speaking, web applications are designed to provide services involving both computation and data processing and delivery. Hence, service offloading approaches need to take both into consideration. Proposed approaches include those that have focused on static content delivery such as [35, 36, 37, 38, 39, 40], and those that address dynamic content caching such as [41, 42, 43, 44, 45].

Web content delivery has attracted a lot of interest in both industrial and academic communities. A simple way to improve content delivery performance is to upgrade the loaded resource: a faster server, a bigger switch, reengineering the network. Unfortunately, such an approach is not always economically feasible and more importantly, it does not consider that today a single web service may consist of many other services that reside in different locations. A better solution is to move web contents closer to end users to reduce latency and improve utilization of network bandwidth. These approaches include caching and replication.

Caching has traditionally been applied in distributed file systems such as AFS [46]. A cache is a temporary storage location containing a set of most commonly accessed objects copied from original servers. A Web cache is a dedicated system which monitors the page or object requests and stores the retrieved pages or objects from the server. On subsequent requests, the cache can serve them from its storage rather than passing requests to the origin server. Studies show that people are apt to access the same popular pages or objects from billions of pages or objects on the web. The advantages of moving such popular objects closer to users are obvious: network latency is reduced, the fixed bandwidth of the upstream link is better utilized, server load is alleviated, etc. Although caching has been a proven technique for improving scalability and performance in distributed file systems, its application to the World Wide Web requires solutions to new problems, for example, managing the deployment of caches, ensuring the consistency of cached data, and handling dynamic content, etc.

Replication is a technique similar to caching. Replication has been commonly applied to distributed systems to improve availability and fault tolerance. Unlike caching, which attempts to store, in a demand-driven fashion, the most commonly

accessed objects as close as possible to end-users, replication distributes, in a more or less static fashion, a site's contents across multiple mirror servers, thereby presenting a larger granularity of the data space. By load balancing requests across multiple replicas, replication systems have very high fault tolerance since in the case that one replica crashes, other replicas or the original server can take over its role to serve requests. Compared with caching, replication is considered to be a mechanism with low flexibility, poor adaptability, increased space consumption and increased difficulty in maintenance. Some recent work such as "materialized views" tries to reduce the amount of replicated contents by defining a "view", which represents a portion of origin database and replicating only that particular portion. However, this mechanism needs to anticipate load or work patterns in order to define appropriate "views", and requires a lot of manual interference in the case that load or work patterns change. [47] presents an early overview on replication and its challenges.

As discussed above, the main distinction between caching and replication has to do with the mechanism (static or dynamic) and the granularity of distribution (small or large). A real application typically involves elements of both approaches. In this document, we use these terms interchangeably.

From the perspective of the underlying network architecture, the research in web caching and replication can be divided into the following categories:

- **Proxy Caching.** A proxy caching server intercepts requests from clients and either generates responses if the requests hit the cache or forwards them to the origin server on behalf of the users, retrieves the responses from origin server, possibly stores them into the cache, and finally returns the response to the

clients. A proxy server is usually deployed at the edge of a network, in order to serve a large number of users internal to an organization, and results in increased availability of static web contents, significant reduction of network latency, and wide area network bandwidth savings. One disadvantage to this design is that the proxy server represents a single point of failure in the network. A proxy caching approach also traditionally required that clients be manually configured to use the appropriate proxy server, although more recent approaches use DNS redirection technologies to avoid this step.

A variant, Reverse Proxy Caching, deploys caches near the origin of contents, as opposed to the client side, such that even when the server receives a large number of requests, it can still provide a high level of quality of service. With Transparent Caching architecture, users are not required to explicitly configure their web browsers. Transparent Caching relies on a HTTP filter which redirects outgoing HTTP requests to web cache servers or cache clusters.

A commercial approach to Proxy Caching, Akamai [48], provides a more general solution. This approach does not require the client to be configured to use a specific caching or proxy server. Instead, the content on the origin site is rewritten such that the embedded links point to a nearby Akamai server. The client retrieves the main page from the origin site and follows the embedded links to retrieve embedded objects from Akamai servers. Additionally, by using the DNS redirection mechanism, the Akamai infrastructure can control the specific server that is selected.

- **Hierarchical and Distributed Caching.** A single cache has a finite size,

and therefore there is a limit to the number of contents that can be cached. A group of caches where every cache shares its cached contents with others can in principle provide better performance. Such caches reside across the network and are organized according to some topology, which can be grouped into two broad categories: Hierarchical Caching and Distributed Caching.

In a Hierarchical Caching scheme, caches are placed at multiple levels of the network, establishing neighborhood relationships with other caches where a parent cache is essentially one level up in a cache hierarchy. A client's request is forwarded up the hierarchy until a cache hit occurs, or, if none occurs, the request is forwarded to the origin server. In [49, 50], several disadvantages of Hierarchical Caching scheme are identified:

- The multiple levels of hierarchy always introduce additional overhead.
- The high level caches in the hierarchy may become bottlenecks.
- The same content being stored at different cache levels introduces redundant storage.

Hierarchical Web Caching was first proposed in the Harvest project [51]. Other examples of similar approaches include Adaptive Web Caching [52] and Access Driven Web Caching [53]. Adaptive Web Caching targets the "hot spot" problem where a web server suffers a sudden surge of traffic due to some particular content being intensively accessed. Adaptive caching consists of multiple, distributed caches which dynamically join and leave cache groups based on the content demand. Adaptive Web Caching developed the Cache Group Manage-

ment Protocol (CGMP) and the Content Routing Protocol (CRP) to provide a caching scheme that is adaptable and self-organizing for heterogeneous demand of web contents.

Distributed Caching was designed as a complementary approach and allows the distribution of caching proxies geographically over large distance. In Distributed Caching, there no longer exist intermediate cache levels: each cache server resides at the bottom level in the network and maintains meta-data information about the cached contents on other caches, used to find the cache that contains the requested contents. For the purpose of distributing the meta-data information efficiently and scalably, a hierarchical distribution mechanism can be applied. Since most requests can be served in the lowest level of the network, Distributed Caching can reduce network latency significantly. The elimination of intermediate cache levels alleviates the redundant storage problem; the distribution of the meta-data information allows better load sharing; and together, they provide a more fault tolerant scheme. The main disadvantages in exploiting Distributed Caching include:

– High penalty for a "cache miss" due to high connection time between cache servers.

– The need for additional network bandwidth to distribute meta-data information

– Complicated administration.

Approaches and protocols for Distributed Caching, include the Internet Cache Protocol (ICP) [54] in the Harvest project, the Cache Array Routing Protocol

(CARP) [55], the distributed Internet Cache by Provey and Harrison [56], Summary Cache [57], etc. [50] gives an overview of these approaches.

Both Hierarchical Caching and Distributed Caching allow the sharing and coordination of cache state among multiple communicating caches to improve system performance and scalability, and fall into a broader category known as "Cooperative Caching". There are two important aspects investigated in most Cooperative Caching research: finding nearby caches which hold the cached content and coordinating the caches while making storage decisions. The first aspect has been widely studied and there are a lot of approaches described above. Taking Summary Cache as an example, it exploits a directory-based scheme where each proxy cache maintains a directory that summarizes the cached content on other proxy caches. Upon a "cache miss", the proxy cache can forward the request to the nearby proxy cache (or the origin server) based on the directory information. The work focusing on the second aspect, coordinated cache placement, includes [58, 59], both of which investigated optimal algorithms for coordinated cache placement problem and proposed a greedy algorithm within a factor (1.1 – 1.5 for the median performance in [58] and 14 for the worst case in [59]) of optimal for practical use. [60] investigated the impact of Cooperative Caching on network performance and scalability and provided a quantitative evaluation based on trace-based analysis and analytic modelling. The results reveal that Cooperative Caching has performance benefits only within limited population bounds (e.g., for small-organization proxies with populations ranging from 200 to 2000 users, Cooperative Caching can increase the average hit rate on a single

proxy by 9% to 17%; however, for large-organization proxies with population sizes larger than 20000, the improvement is between 3.3% and 3.7%). The results also imply that the increased latency of inter-proxy communication in the wide area network overshadows the benefits of Distributed Caching, advocating for a hierarchical variant.

- **Peer-To-Peer Caching.** In contrast to a traditional client-server content delivery network, a peer-to-peer (P2P) content delivery network aims at utilizing a client's spare resources to provide a high performance caching mechanism, presenting an interesting research area for academic and industrial communities [40, 61]. Relying on a peer-to-peer routing protocol, the participant can join or leave the system at any time and any place, providing a highly self-organizing content delivery network.

  Coral [40], is an example of DHT-based peer-to-peer content delivery network. Nodes that volunteer to run Coral automatically replicate contents when users access them and publish these replicated contents through the Coral network by simply prepending a pseudo-hostname to objects' URLs; a peer-to-peer DNS layer is applied to transparently redirect requests to nearby participating cache nodes. Coral relies on a *distributed sloppy hash table* (DSHT) [62, 63] for distributed DNS lookup; DSHT is designed to avoid "hot spot" situations where large numbers of (key, value) pairs in DSHT have the same key by guaranteeing the rate of store requests at the most heavily-loaded node is only logarithmic in the total number of nodes.

  Although a peer-to-peer content delivery network can achieve high performance

by serving the requests from nearby cache nodes, its performance could drop dramatically if the number of volunteers is small. Furthermore, this approach potentially has a higher connection time due to peer-to-peer DNS lookup and request redirection, and does not provide QoS assurance. Finally, the security concern of caching possibly sensitive content on untrusted volunteer nodes remains a big issue.

From the perspective of the types of web content, approaches for web caching and replication can be divided into two categories: *static content* caching and *dynamic content* caching. Traditionally, Web servers have been used to hold pre-generated static content. Yet there exists another sort of information where the documents or components of documents are generated dynamically upon receiving a client request. Such content is referred to as dynamic content, and includes examples such as rotating advertising banners, CGI scripts, ASP/ASPX pages, etc.

Although dynamic content is becoming a large fraction of overall web traffic, existing web architectures have inherent inefficiencies in the delivery of such content. The main challenge is in guaranteeing the freshness, consistency and accuracy of the content in the cache. Traditional caching approaches, such as [35, 36, 37, 38, 39], can only handle static content and dodge the dynamic content caching problem by marking them as "non-cacheable". However, a study of proxy cache effectiveness [60] shows that the fraction of all requests for dynamic content can amount to as much as 50%, so this solution is not adequate.

Studies show that although the content to serve user requests might change over large time duration, this content remains unchanged for certain shorter time periods.

And even when it changes, there are portions that may remain unchanged. Most dynamic content caching approaches take advantage of this fact by tracking freshness of the content stored in the cache and applying an invalidation-based or update-based consistency mechanism. [64, 65] proposed a fragment-based approach for dynamic content caching. This approach views dynamic web pages as being composed out of simpler entities, known as *fragments*. Fragments typically represent parts of Web pages which change together such that when a change to underlying data occurs which affects several Web pages, the fragments affected by the change can be easily identified. This permits the content in the caches to be updated on the level of fragments as opposed to Web pages, improving performance significantly. For such approaches, users or content providers need to specify how Web pages are composed from fragments by creating templates in a markup language. These templates are parsed to determine inclusion relationships among fragments and Web pages, typically represented as an *object dependence graph* (ODG). The ODGs are used to determine how changes should be propagated throughout the Web after one or more fragments change.

We discuss three representative approaches for dynamic content caching below:

- **Active Caching.** Active Caching [66] was first proposed in the WisWeb project at the University of Wisconsin, Madison as a solution for the web personalization problem. A recent study [67] of HTTP traces from a large ISP reveals that about 30% of all user requests carry cookies, the HTTP header elements typically indicating that a request be personalized.

  The key component in Active Caching is an *applet*: a piece of specialized code

provided by the origin server which is attached to the response content. Both content and the associated applets are stored in the cache and upon a subsequent request, the cache can execute applets which customizes the document to serve the request.

The major drawback of this mechanism is that the content providers have to relinquish control over part of the application logic.

- **CONCA.** CONCA [44] addresses two major issues for web content access: dynamic content caching and "on-the-move" access for mobile users.

  The CONCA design proposes two key components which reside on the server side and the client side respectively to provide information about content structure and user content preferences. The server side component associates the content supplied by the server with a "document template" which defines both the structure and form of content. The client side component, called *Personal Assistant*, identifies the kinds of devices the user has access to, the templates for each of these devices, transcoding information about the original content and converted content, and additional information such as consistency linkage between these two kinds of contents. By exposing such information on both the server side and the client side, CONCA automatically bridges the semantic gap between the two to enhance the effectiveness of proxy caches. CONCA applies a distributed client-side proxy caching architecture to support caching of dynamic, transcoded content. For nomadic users, CONCA makes use of a *home cache* which maintains per-user state persistently, and allows the persistent state to be recreated on another cache as the user moves.

The main weakness of CONCA is that it requires significant communication between proxy caches for exchanging cache states. Also, it needs administration to set up the two components on both server and client sides. Furthermore, it is not clear that how much the nomadic user can benefit from the use of the *home cache* since the communication between two caches will introduce additional overhead.

- **IBM WebSphere Edge Server and Akamai EdgeSuite.** As an example of a commercial approach to Dynamic Content Caching, IBM WebSphere Edge Server [68] provides a mechanism to offload application components, such as servlets, Java Server Pages, Enterprise Beans, to the edge server in the network. The edge server, acting as an application-server proxy, can handle some dynamic content requests locally and forward the others to the origin server. The major disadvantage for this scheme is that for data-intensive applications, where the data has to be fetched from the origin server, the improvement of web performance and scalability is quite limited.

  Akamai EdgeSuite [69] leverages IBM WebSphere to allow Akamai's customers to distribute their Web application workload into the network. Akamai relies heavily on Edge Side Includes (ESI), a markup language that breaks Web pages into fragments [64, 65] with a profile describing the ability to cache items. Fragments may be labeled as cacheable for different time scales, such as days, minutes or seconds. Upon request, the fragments are assembled into an HTML page at the edge and only those fragments deemed impossible to cache are retrieved from the origin server. The main drawback of ESI is the overhead required to

re-tag pages to identify cacheable content.

While the approaches outlined above work with all server structures, a growing number of web servers are structured as two parts: an application server and a back-end database, where the application server processes a client request and generates a response by querying against the back-end database; the back-end database is responsible for data fetching and management. Recent web caching and replication approaches have attempted to improve the performance and scalability for such systems by caching portions of the back-end database on edge servers. Proposed approaches that fall into this category include [41, 42, 43, 70, 45]. These approaches usually rely on a query wrapper and a query matcher to intercept the request at the cache, and based on the query matching result, either run a corresponding query against local storage or invoke a remote query for the origin database. The approaches differ in the consistency mechanism and the storage organization of the cache. The consistency mechanism used in [41, 42, 43, 70] is invalidation-based, while in [45] an update-based consistency mechanism is employed. For the storage organization, [41, 42, 43] use an unstructured, form-based mechanism to represent the cached data which results in a redundant storage problem; [70, 45] uses a stand-alone database which contains only a portion of the origin database.

The different strategies for storage organization discussed above can be divided into two categories: *Page Caching* and *Tuple Caching*. Page Caching is widely used and assumes that each query can be broken down to the level of requests for individual pages. The data in the proxy cache is organized as individual pages and upon a "cache miss", the proxy can retrieve the particular pages from the origin server on behalf of

the end user. Tuple Caching maintains the data in the cache in terms of individual tuples, resulting in a higher flexibility compared with Page Caching. The tuples in the cache are indexed using their primary keys.

Although these strategies work when there is an explicit back-end database, they can not be used in situations such as web-based retrieval systems where requests can still be viewed as queries against a database but there may not be a physical database storing the data required for the response. In such cases, the cache contains the responses which may be dynamically generated for the requests, indexed by the query contained in the request. Page Caching fails in such cases because the query used in such information system is keyword-based and the data organization at the servers is completely hidden from the clients (and is not necessarily a page-based organization). Tuple Caching fails because the request contains a filled search form that describes a query instead of a primary key and furthermore, it can not inform the server about the already cached tuples in order to reduce response size.

To overcome these deficiencies, [71, 72, 73, 74, 75] proposed *Semantic Caching*. In Semantic Caching, data in the cache are managed as a collection of *Semantic Regions*. A Semantic Region is a set of tuples that are defined and adjusted dynamically based on the queries posed at the clients. The usage of Semantic Regions not only addresses the limitation of Page Caching in page-based data organization, but also resolves the indexing problem for Tuple Caching. Furthermore, it avoids the high storage overheads in Tuple Caching strategy which has to maintain the replacement information on a per-tuple basis. Upon a query being posted, the query is split into two parts: a *probe query* and a *remainder query*. The former is used to retrieve the portion of the answer available in the local cache and the latter is used to retrieve the

missing data from the origin server [71].

The DataSlicer architecture can be viewed as a generalized approach in the Web Caching and Replication category: it deals with dynamic web content, employs a distributed caching scheme, applies semantic caching technology for service usage modeling and replication, and provides QoS-assured guarantees to end-clients.

3.3 Summary

This chapter has described the background context for building a scalable network service architecture and discussed some representative existing approaches that only address a subset of the the challenges described in Chapter 2. In the next several chapters (Chapter 4 – 7), we describe how DataSlicer exploits the techniques described in Section 1.2 to address these challenges.

Chapter 4

Service Usage Locality and its Detection

The DataSlicer architecture relies on the existence of locality in service usage patterns. The first part of this chapter investigates the characteristics of service workloads with respect to three important patterns: *spatial locality*, *network locality* and *temporal locality*, and demonstrates the existence of such locality in data-centric network service usage patterns. The second part describes how DataSlicer dynamically detects such locality patterns by inspecting the underlying traffic between clients and the services at the distributed routers.

4.1 Existence of Locality

The DataSlicer architecture needs to automatically make decisions about what service regions need to replicate, where to locate these replicas, and when to perform

such actions. One way to help answer these questions is to characterize the service usage patterns across three dimensions: *data space*, *network regions* and *time epochs*. Consequently, the locality existing along each dimension can be called *spatial locality*, *network locality* and *temporal locality*, respectively.

- **Temporal Locality** refers to the property that within a certain time period, a subset of objects of the service database are more frequently accessed than others. Such locality arises from the presence of "hot" objects in the database whose popularity changes with time; for example, on a news service website like CNN.com, such locality may be correlated with some important breaking news or unusual events such as finals of the National Football League (NFL) or world series of the Major League Baseball (MLB). Detection of such locality can help alleviate "hotspots" and flash crowd-like effects at the service.

- **Spatial Locality** refers to the property that certain regions of the database are more frequently accessed than others. For example, certain book categories at an online bookstore like Amazon.com might receive a dominant fraction of all requests. Similarly, objects (web pages or links) related to some "hot" keywords in a web search service provider like Google might see a higher access rate than others. Capturing spatial locality can help define appropriate views of the database to drive the database partitioning and replication process.

- **Network Locality** refers to the property that a certain group of users, co-located in network space, are responsible for a dominant fraction of all requests to the service. Such locality typically arises because of shared interests among a group of users who share one of more of levels of network elements in the

paths their requests take to the service, e.g., one expects map services such as MapPoint.com to exhibit significant geographical locality. Capturing network locality can help identify appropriate locations in the network to place service replicas.

To understand to what extent the above three kinds of locality exists in data-centric network services usage patterns, we investigate the characteristics of workloads from two well-known public services: SkyServer and TerraServer. Our choice of these sites was driven by the fact that we were able to obtain access to their request logs, often the major impediment to performing a study similar to ours. SkyServer's web logs are publicly available, while TerraServer's logs were provided to us by Microsoft Research personnel.

The rest of this section describes the structure of the web-traces from the two public services, and is followed by a description of the methodologies we use to analyze the characteristics of the workloads. We then report on the results we find, and conclude with a brief discussion of their implications.

4.1.1 Web-trace Structure

The SkyServer web site provides Internet access to the public Sloan Digital Sky Survey (SDSS) data for both astronomers and for science education. The SDSS is a 5-year survey of the Northern sky (10,000 square degrees) to about 1/2 arcsecond resolution using a modern ground-based telescope. Its goal is to characterize about 200 M objects in 5 optical bands — Ultraviolet, Green, Red, Near Infrared, and Infrared — and measure the spectra of a million objects [76]. The web site has been operating

since June 5, 2001 and as of this writing, receives approximately 3 M requests every month.

The TerraServer web site is one of the world's largest online databases, providing free public access to a vast data store obtained from the US Geological Survey (USGS). The web site contains 3.3 TB of high resolution USGS aerial imagery and USGS topographic maps. The TerraServer web site has been operating since June 1998 and is heavily used: a typical day sees 25 M – 30 M requests.

The underlying architecture of both sites is similar and built on top of Microsoft's IIS and .NET framework offerings: a front-end IIS web server accepts HTTP requests and passes them to corresponding HTTP handlers (e.g., an ASP, ASPX, or ASHX handler) for processing. The HTTP handlers, which implement the service functionality, query against a back-end database server to generate responses by formatting the returned records into HTML pages/SOAP messages. The access logs we work with are recorded by the IIS server, and include requests for static content; we filter out such requests from our analysis as described below. Because of its heavier load, the TerraServer site uses a web server farm, with each server logging its own trace. Our analysis is based on an aggregate trace obtained by merging these individual traces by timestamp order.

Our analysis of the SkyServer site is based on a four month trace from January 1, 2004 to April 30, 2004, containing 11.4 M requests. For the TerraServer site, we could only obtain access to a single day's trace, April 5, 2004, which contained 24 M requests. Table 4.1 shows the entry structure of request logs at the two web sites. Each entry in the request log contains the following information: a timestamp, an IP address of the client that made the request, the name of the requested page/ser-

Table 4.1: Entry structure in the two web traces.

Index	Field
1 – 3	**yy, mm, dd**
4 – 6	**hh, mi, ss**
7	seq
8	logID
9	**clientIP**
10	op
11	**command**
12	error
⋮	⋮

(a) SkyServer Weblog

Index	Field
1	**date**
2	**time**
3	s-computername
4	cs-method
5	**cs-uri-stem**
6	**cs-uri-query**
7	cs-username
8	**c-ip**
⋮	⋮

(b) TerraServer Weblog

vice, and the supplied parameters for the request. For the SkyServer request log, the first 6 fields, “yy”, “mm”, “dd”, “hh”, “mi”, and “ss”, together provide the timestamp information; field “clientIP” provides the client IP address information; field “command” provides both the name of the requested page/service and the supplied parameters for the request. For the TerraServer request log, the “date” and “time” fields together provide the timestamp information; field “c-ip” provides the client IP address information; field “cs-uri-stem” provides information of the name of the requested page/service; and field “cs-uri-query” provides information about parameters supplied for the request.

The logs consist of requests for both static and dynamic content. Our analysis focuses on the latter requests, which involve accesses to a back-end database. We apply a broad interpretation for such requests, including both services that are invoked using SOAP, as well as ASP .NET web applications that are invoked using HTTP

Table 4.2: Fraction of static and dynamic content in the two web-traces.

SkyServer			TerraServer		
Content	hits (M)	%	Content	hits (M)	%
all	*11.402*	*100*	*all*	*24.220*	*100*
static	6.181	54.21	static	5.809	23.99
dynamic	5.221	45.79	dynamic	18.411	76.01

GET/POST. Table 4.2 summarizes the fraction of each category of request.

For each dynamic requests, we parse the logs to identify the request command ("service") and parameters, discarding requests that were either ill-formatted in syntax or incompatible in semantics (this constituted a very small fraction of all requests, fewer than 0.34%). While we looked at a broader set of requests, here we restrict our attention to the most frequently invoked services (out of thousands) that dominate the dynamic traffic at each site. Table 4.3 summarizes the statistics of each kind of request. The client IP address columns correspond to the number of unique addresses requesting that service, and unique groups of addresses that share either their first three octets (corresponding to a 8-bit subnet) or their first two octets (loosely corresponding to a 16-bit subnet).

Services for the SkyServer site include:

- ***x_rect.asp***: which takes as input a rectangle in the sky (specified by the rectangle center, width, and height expressed in terms of the sky coordinates of right ascension and declination) and five optional optical bands, and returns a list of objects found in that rectangle.

- ***x_radial.asp***: which takes as input a circle in the sky (specified by a center point

Table 4.3: Types of services showing request and client IP statistics.

Service	Requests		Client IPs		
	(M)	%	unique	8-bit	16-bit
SkyServer					
all	*5.221*	*100*	-	-	-
x_rect.asp	2.490	47.69	219	188	168
x_radial.asp	0.417	7.99	735	548	432
getjpeg.aspx	0.476	9.12	7410	5223	2860
shownearest.asp	0.339	6.49	9003	7153	3648
x_sql.asp	0.506	9.69	-	-	-
TerraServer					
all	*18.411*	*100*	-	-	-
tile.ashx	16.516	89.71	29029	25699	5466

and a radius expressed in terms of the sky coordinates of right ascension and declination) and five optional optical bands, and returns a list of objects found in that circle.

- ***getjpeg.aspx***: which takes as input a rectangle in the sky, a desired map scale and a set of drawing options, and returns a JPEG image.
- ***shownearest.asp***: which takes as input a circle in the sky, a desired map scale and a set of drawing options, and returns a thumbnail JPEG image.
- ***x_sql.asp***: which takes as input an arbitrary SQL query against the back-end database, and returns the results in required format (HTML, XML or CSV).

For the TerraServer site, one request type dominates:

- ***tile.ashx***: which returns a 200 pixels $\times$ 200 pixels imagery of the "Photo", "Topo", or "Relief" type corresponding to a point on the Earth's surface spec-

ified using the UTM coordinate system [77], on a desired map scale. Requests for the "Photo" type contribute to $\sim 85\%$ of the whole, so we limit our attention to that type.

We characterize locality properties for each of these services, except *x_sql.asp*, whose requests make arbitrary queries again the underlying database. As reported in [76], the relational schema of the SkyServer database is very complicated, thereby making it difficult to analyze data region accesses by requests without additional information about the database internals.

4.1.2 Locality Characterization

To enable the characterization of the three kinds of locality defined at the beginning of this section, we correlate the parameters of each service request with the region of the underlying database used to respond to the request. Ideally, we would like this correlation to be in terms of the physical back-end database. However, for most production services like SkyServer or TerraServer, obtaining the relational schema of the back-end database not only raises difficulties in practice, but even when available may prove overly cumbersome for detailed analysis. As an alternative, for data-centric web services that are our primary interest, we use the information provided in the documentation of the service interface to construct a simpler *logical view* of the database.

Logical view of the back-end database. The logical view of a service's database is constructed by examining the supplied parameters in the service's WSDL interface, and defining a virtual data-space accessed by these parameters, assuming that they are

numeric or alphabetic rangeable. Note that the logical view only approximates usage of the underlying physical database, not its structure.

We are often interested in only a subset of this overall data-space corresponding to parameters that exhibit the most variation across requests. Thus, a view defines a multidimensional data space based on one or more parameter value ranges. Following the concept of *semantic regions* in database caching [71], we can partition such a data space into multiple disjointed regions at some granularity, and model service accesses at the level of these regions.

Our analysis makes use of the logical service views as follows. An individual service request can now be interpreted as an *edge* originating from a particular client IP address and directed to a point in such a multidimensional data space. The reason it is a point (instead of more generally being a region) is because to ensure efficiency, most services usually enforce a range limit on an individual request. As an example, on the SkyServer site, the *x_rect.asp* service only allows a request to search objects within a rectangle which has a maximum size of 0.2 degrees by 0.2 degrees in sky coordinates. By examining clustering of request edges in both the IP address space and the logical view data space, we can quantify the extent to which a request trace exhibits different kinds of locality.

Examples. We take service *x_rect.asp* as an example to show how to construct a "logical view" for a service. As introduced in Section 4.1.1, the input for the service consists of 9 parameters, and hence its logical view might consist of 9 attributes: the coordinates of the center-point, width and height of a rectangle in the sky, and the optional 5 optical bands. An alternative view could consist of the center-point and

Table 4.4: Logical views of the two services.

Service	Input	View
x_rect.asp	rectangle (in sky), 5 optical bands	2D: rectangle center-point
x_radial.asp	circle (in sky), 5 optical bands	2D: circle center-point
getjpeg.aspx	rectangle, scale, drawing options	3D: rectangle center-point, scale
shownearest.asp	circle, scale, drawing options	3D: circle center-point, scale
tile.ashx	scale, point (in UTM), photo type	4D: scale, 3D point

the rectangle dimensions by clustering together the dimensions corresponding to the optional 5 optical bands; service requests in our traces exhibit relatively little variation for these values. Further simplification is possible by observing that each request can refer to a maximum rectangle of a known size, hence it suffices to define the view in terms of the coordinates of the rectangle's center-point.

Table 4.4 lists the logical views that we constructed for each of the five services. Notice that while the first 4 services have a rather simple view which consists of 2 or 3 dimensions, the last service's consists of 4 dimensions. In the rest of this chapter, we use the following abbreviations to refer to these dimensions: for the SkyServer services, the map scale dimension is denoted as "S", the right ascension coordinate dimension is denoted as "X", and the declination coordinate dimension is denoted as "Y"; for the TerraServer service, the map scale is denoted as "S", the scene dimension (the "zone" coordinate in UTM) is denoted as "T", and the "easting" and the "northing" dimensions are denoted as "X" and "Y", respectively. TerraServer uses

these four parameters to identify a point on the earth's surface using the UTM system.

The parameters of each dynamic request in the traces described in Section 4.1.1 are translated to a point in the data space of the logical view for that service. In cases where two services share the same logical view (this happens in the case of *x_rect.asp* and *x_radial.asp* in our case), we pool the requests into a combined group ordering them using the associated timestamp information.

4.1.3 Methodology

As stated earlier, our analysis goals are to characterize to what extent the request logs exhibit *temporal*, *spatial*, and *network* locality. Note however, that the locality structure depends on various parameters: the granularity of data space regions, how addresses are grouped, and the timescales of interest. To systematically examine the effect of these factors, we parameterize the request workload: a particular assignment of parameter values divides up the logical data space into partitions of a certain size, the trace into different time epochs, and the client IP addresses into different address groups.

We then examine locality patterns over a range of values for these parameters, as shown in Table 4.5. The role of the "IP address groups" and "timescale" parameters should be clear: we consider two groupings of client IP addresses, based on whether they share their first 2 or first 3 octets [1], and different time epoch sizes ranging from an

[1] The intention here was to group network addresses to ascertain if locality exists at higher levels of the network, say at the granularity of 8-bit and 16-bit subnets. We realize that the 2-octet grouping is only an approximation for the 16-bit subnet evaluation; a more precise analysis would require inferring the IP address-AS associations.

Table 4.5: Range of parameter values over which locality characteristics were investigated.

Parameter	Values investigated
x_rect.asp	
partitioning policy IP address group timescale	$2^{\{X,Y\}} \backslash \phi \times \{4,7,10\}$ share first {2 | 3} octets {hour, day, weekday, weekend, week, month, all}
getjpeg.aspx, shownearest.asp	
partitioning policy IP address group timescale	$2^{\{S,X,Y\}} \backslash \phi \times \{4,7,10\}$ share first {2 | 3} octets {hour, day, weekday, weekend, week, month, all}
tile.ashx	
partitioning policy IP address group timescale	$2^{\{T,S,X,Y\}} \backslash \phi$ share first {2 | 3} octets {hour, day (all)}

hour to the entire duration of the trace. The "partitioning policy" parameter controls along which dimension(s) and at what granularity the logical data space of the service is partitioned. For example, in the SkyServer *getjpeg.aspx* service, we examine 21 partitioning policies corresponding to seven choices of partition-by-dimension (`S, X, Y, SX, SY, XY, SXY`) , and three granularities for each of these choices (a granularity value of x implies that regions are defined by dividing each dimension of the partition into 2^x equal intervals). To explain the notation, a partitioning policy value of `XY10` corresponds to 2^{20} data space regions; each region spans $1/2^{10}$ of the range of the `X` and `Y` dimensions. For the TerraServer service *tile.ashx*, the granularity of each dimension to be partitioned is set to a fixed value: 11 regions for the "S" dimension, 10 for "T", and 40 for both "X" and "Y".

Characterizing locality

To reason about the three kinds of locality in a unified fashion, we compute a "load fraction" graph. Each point in this graph corresponds to the *maximum* request load (as a fraction of all requests) that is observed between a set of address groups and a set of data space partitions, under the constraint that the cardinality of these two sets is bounded to certain values. Informally, the graph shows the potential benefits of caching a subset of data space view regions at a subset of the network locations from where requests originate. For example, in Figure 4.2, a point at location (30, 10, 0.84) can be interpreted as meaning that "10% of 8-bit IP address groups contribute at most 84% of requests on 30% of the regions".

The computation of the maximum request load is NP-hard with the underlying problem stated as follows. Assume that there are two sets: C and R, where C contains m client IP addresses and R contains n disjoint database regions; $v_{i,j}$ is the number of hits on region r_j that are sent by client c_i; s_j is the size of region r_j. Given two budgets T_C and T_L, the problem can be formulated as:

maximize: $\sum_{i=1}^{m} \sum_{j=1}^{n} v_{i,j} x_i y_j$

subject to: $\sum_{i=1}^{m} x_i \leq T_C$

$\sum_{j=1}^{n} s_j y_j \leq T_L$

where: $x_i \in \{0,1\}$, $i = 1, \ldots, m$, and denotes the inclusion of client i in set C;
$y_j \in \{0,1\}$, $j = 1, \ldots, n$, and denotes the inclusion of region j in set L.

The above problem can be easily reduced to the well-known 0-1 Knapsack Problem [78], which is an NP-hard problem.

Therefore, we use a greedy heuristic (Figure 4.1) to approximate the maximum request load. The heuristic takes as input two budgets (the thresholds on the fraction of IP address groups and of regions), a set of client IP address groups, a set of regions (represented as leaf nodes in the Cell tree) and a set of edges between the two sets where an edge represents one of the addresses in the address group making a request to the stated region; the edge weight denotes the number of such requests. The algorithm then "greedily" selects the maximal weight edges until the two budgets are exhausted. Clearly, the heuristic provides a lower bound on the achievable load fraction.

Inputs:

C: a set of client IP addresses
R: a set of disjoint partitioning regions
E: a set of edges between C and R, $e_{i,j} \in E \Leftrightarrow$ client c_i has requests hit on region r_j;
  the weight of $e_{i,j}$ is the number of requests
B_c, B_l: two budgets

Variables:

C': a set containing the selected elements from C, $|C'| \leq B_c$
R': a set containing the selected elements from R, $|R'| \leq B_l$

Output:

the total weight of edges between C' and R'

Algorithm:

– sort the edges in decreasing weight order
– select the heaviest edges until one of the budgets is exhausted: edge $e_{i,j}$ is selected
  $\Rightarrow C' = C' \cup \{c_i\}$ and $R' = R' \cup \{r_j\}$
– once a budget is exhausted, select the heaviest edges that originate from the nodes in
  that selected set until the other budget is also exhausted
– select all remaining edges whose head and tail are already in C' and R'

Figure 4.1: Greedy computation of "load fraction".

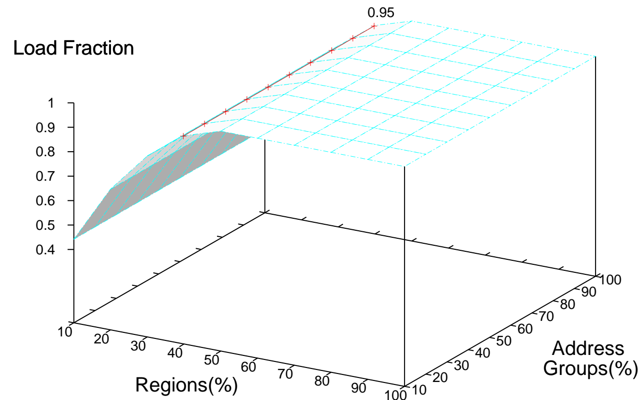


Figure 4.2: IP_3_PP_XY10_TS_All: Spatial locality for requests accessing the *x_rect.asp* service in the SkyServer trace.

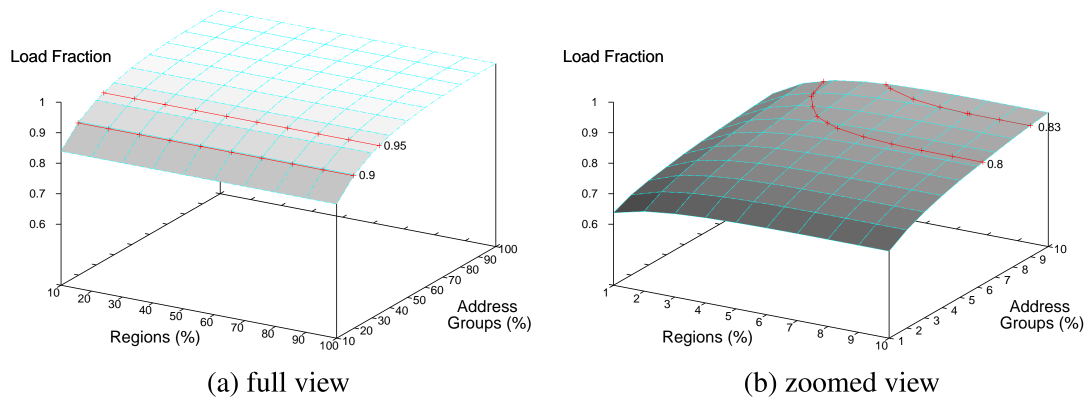
4.1.4 Results

Our results show that there exists high spatial and network locality in both workloads on the SkyServer and TerraServer sites. In this section, our discussion focuses on *x_rect.asp* and *tile.ashx*, the two most representative services at each site.

Each graph presented in this section is indexed with a string of the following form "IP_*n*_PP_*dims*[*l*]_TS_*epoch*[_*val*]", referring to a specific assignment of parameters (IP address grouping, partitioning policy and timescale) as discussed in Table 4.5; parameter "val" indicates that the value is computed as an avg/max/min of request loads from multiple time epochs.

Overall locality characteristics.

For SkyServer's *x_rect.asp* service, corresponding to finest granularity regions (where the data space is divided into 2^{20} regions), addresses grouped into 8-bit subnets, and the largest timescale (4 months spanning the entire trace), Figure 4.2 shows that: (1) 10% of client IP addresses contribute about 99.95% of requests; (2) 84.04% of the re-



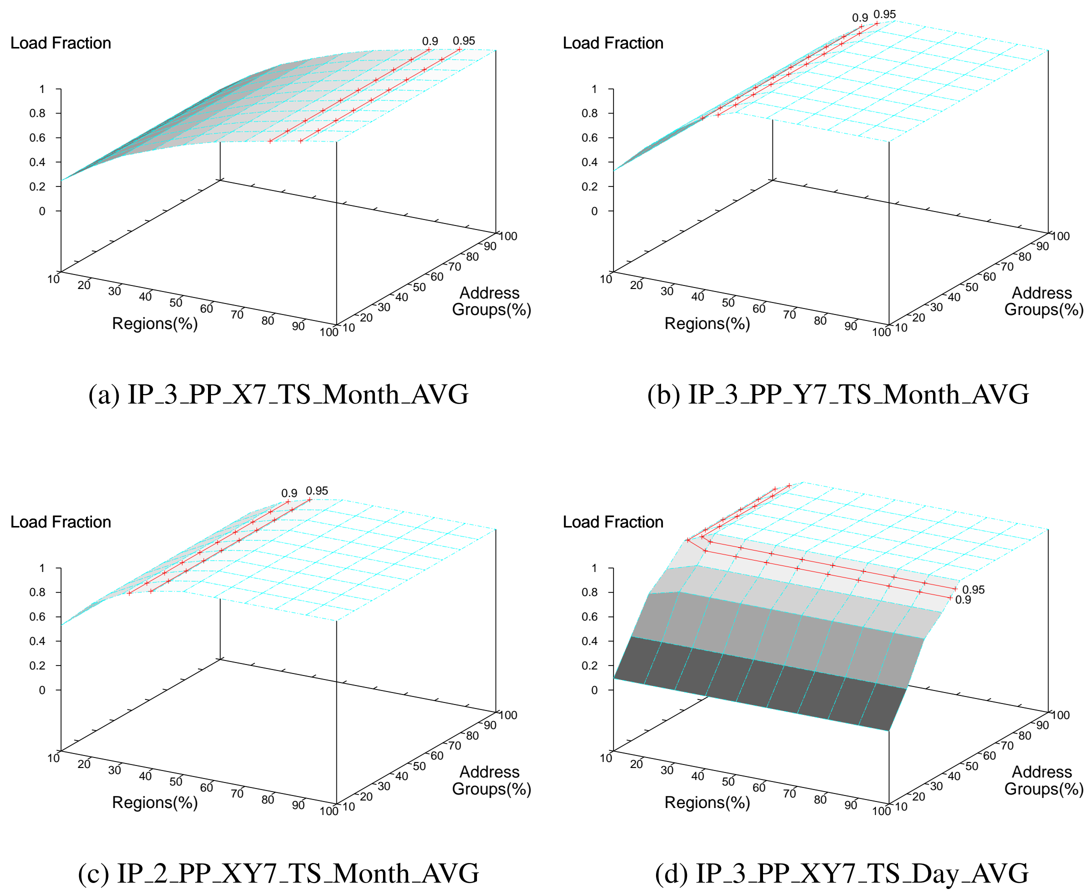
(a) full view (b) zoomed view

Figure 4.3: IP_3_PP_TSXY_TS_All: Spatial locality for requests accessing the *tile.ashx* service in the TerraServer trace.

quests hit on 30% of regions in the data space; and (3) because of (1), for any specific certain fraction of regions, increasing the fraction of IP addresses from 10% to 100% does not affect the computed load fraction too much (the maximum load fraction that can be added is only 0.05%).

Similarly, for TerraServer's *tile.ashx* service, corresponding to finest granularity regions (where the data space is divided into 176,000 regions), addresses grouped into 8-bit subnets, and the largest timescale (1 day, spanning the entire trace), Figure 4.3(a) shows that: (1) 10% of client IP addresses contribute about 83.94% of requests; (2) 99.94% of requests hit on 10% of regions in the data space; and (3) because of (2), for any specific certain fraction of IP addresses, increasing the fraction of regions from 10% to 100% does not affect the computed load fraction too much.

The results imply that for both workloads, caching a small fraction of regions at a small fraction of subnets could potentially reduce a large fraction of service traffic in the network. The results in Figure 4.3(a) indicate that if we could only cache up to 10% of the regions at up to 10% of the subnets, we could still cover (i.e., potentially



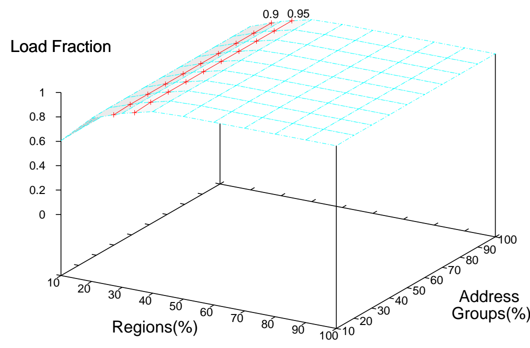
(a) IP_3_PP_X7_TS_Month_AVG (b) IP_3_PP_Y7_TS_Month_AVG

(c) IP_2_PP_XY7_TS_Month_AVG (d) IP_3_PP_XY7_TS_Day_AVG

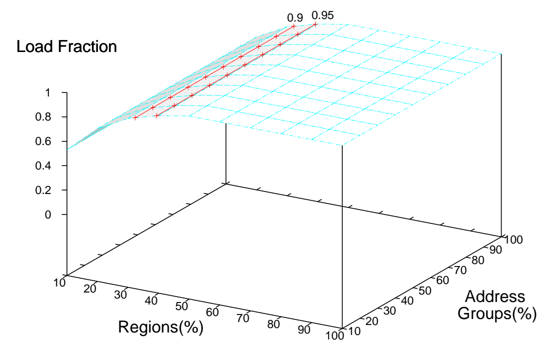
Figure 4.4: Workload locality for SkyServer's *x_rect.asp* service.

improve the performance of) as much as 83% of the overall request traffic. Zooming in on this figure (Figure 4.3(b)) reveals that even if we could only cache at most 1% of the database view regions at no more than 1% of the subnets, we could still cover as much as 63% of the overall request load.

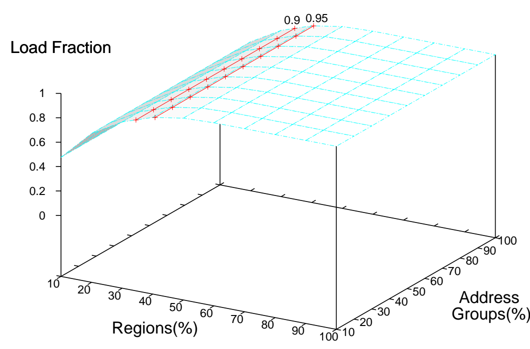
Figures 4.4 – 4.7 present the detailed characteristics of workload locality for the *x_rect.asp* and *tile.ashx* services, over a range of representative values for data space region granularity, addresses grouping and time epoch size. We discuss the salient points of these graphs below.



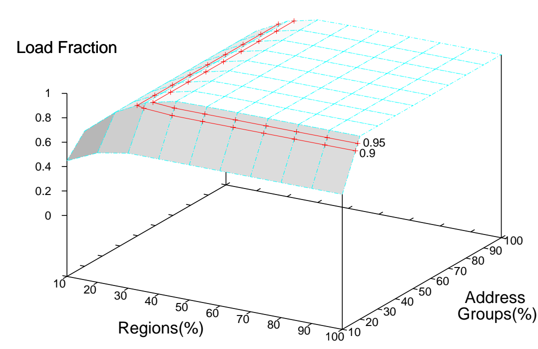
(e) IP_3_PP_XY7_TS_Week_AVG



(f) IP_3_PP_XY7_TS_Month_AVG

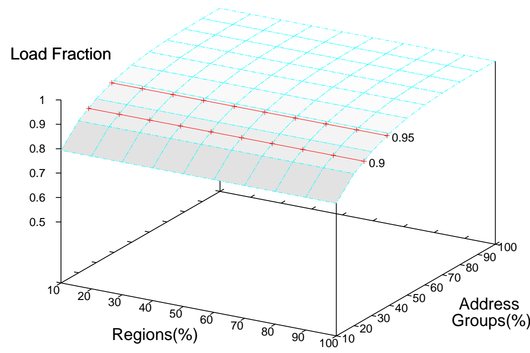


(g) IP_3_PP_XY4_TS_Month_AVG

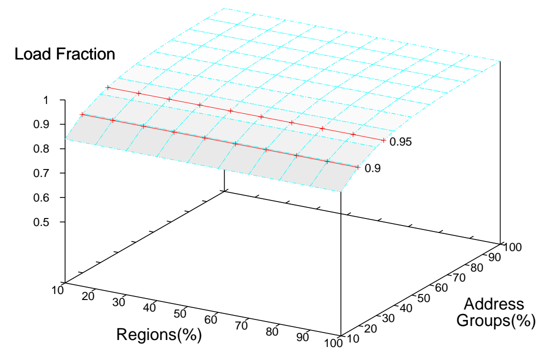


(h) IP_3_PP_XY10_TS_Month_AVG

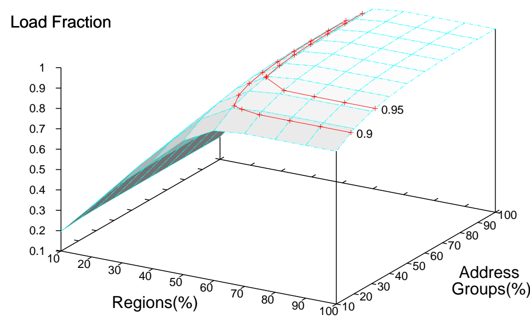
Figure 4.5: Workload locality for SkyServer's *x_rect.asp* service (cont'd).



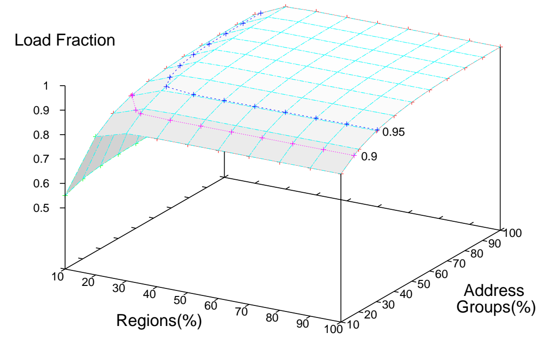
(a) IP_2_PP_TSXY_TS_All_AVG



(b) IP_3_PP_TSXY_TS_All_AVG

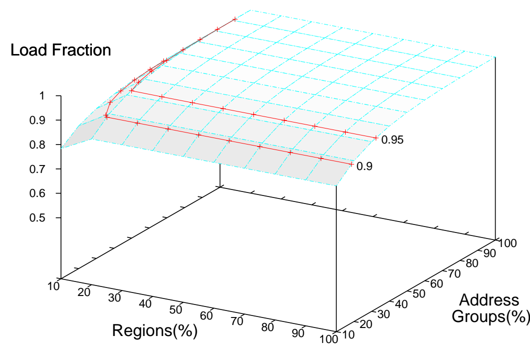


(c) IP_3_PP_X_TS_All_AVG

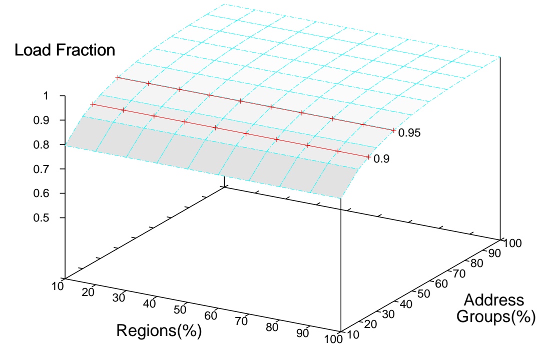


(d) IP_3_PP_Y_TS_All_AVG

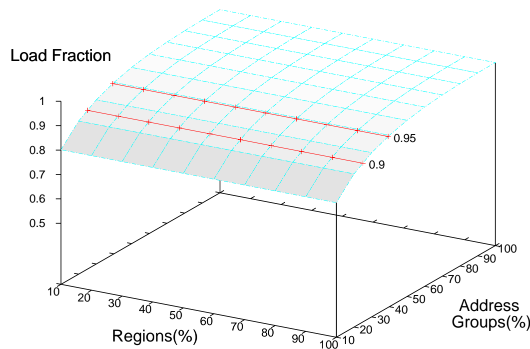
Figure 4.6: Workload locality for TerraServer's *tile.ashx* service.



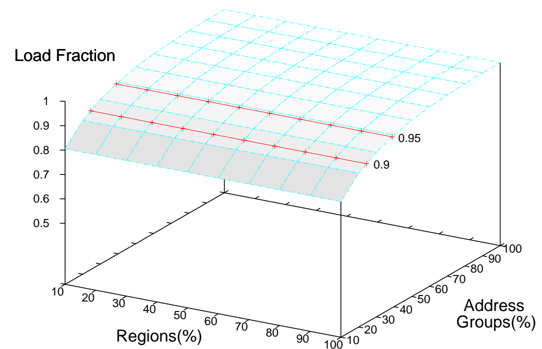
(e) IP_3_PP_XY_TS_All_AVG



(f) IP_3_PP_TSXY_TS_Hour_MIN



(g) IP_3_PP_TSXY_TS_Hour_AVG



(h) IP_3_PP_TSXY_TS_Hour_MAX

Figure 4.7: Workload locality for TerraServer's *tile.ashx* service (cont'd).

Impact of IP address grouping.

To understand if the locality structure is different for higher levels of the network (since any practical replication strategy would need to share a replica among multiple clients), we examined how the load fraction graph changes when all IP addresses that share either their first two octets (loosely corresponding to a 16-bit subnet) or their first three octets (corresponding to a 8-bit subnet) were pooled together into an address group. In the *x_rect.asp* trace, we found that for both grouping scenarios, 10% of the groups contribute to about 99.95% of the requests, suggesting that a large number of requests continue to benefit even if replicas can only be created at higher network levels (Figure 4.4(c) and Figure 4.5(f)). A similar trend was observed for the *tile.ashx* trace. The locality structure stayed the same for the finer grouping, and only reduced slightly for the coarser one where 10% of the groups ended up contributing to 79.2% of the overall requests (Figures 4.6(a) and (b)).

The observations imply that a service replication infrastructure has considerable flexibility in choosing how to improve service performance, and can strike an appropriate tradeoff between prioritizing client-perceived latency and data offloading costs (which would be lower with fewer replicas at higher levels of the network).

Impact of region granularity.

To understand how much of the locality structure is still present for coarser granularity regions in the data space (again, for practical reasons, one would like to replicate large contiguous blocks of the data space as opposed to individual scattered relations), we examined the nature of the load fraction graphs for different granularity values. We find that significant amounts of locality continues to be present even at coarser

granularities.

In the *x_rect.asp* trace, when client IP adddresses are grouped into 8-bit subnets, one still sees a large fraction of client requests targeting a small number of coarser-grained regions. Compared to 93.66% of requests hitting 30% of the regions, when region size is $1/2^{10}$-th of the data space size along each dimension, 88.42% requests hit on 30% of the regions when the region size grows by a factor of 8 along each dimension, and 86.22% of requests continue to hit on the same fraction of regions even when this factor goes up to 64 (Figures 4.5(f), (g) and (h)).

The figures also show that at the finest region granularity, the network locality measured is lower than at coarser granularity when the fraction of address groups is low (10%): 60.67% of requests hitting on 30% of regions, as opposed to 88.38% and 86.19% in the other two figures. This behavior is an artifact of our greedy algorithm, which because it does not prioritize one budget over another, may occasionally mis-select an edge whose weight may be maximal, but whose client IP address group contributes fewer hits than another (e.g., because the second address might be involved in multiple edges).

More interestingly, our analysis reveals different locality behaviors for different region shapes suggesting that, for a given region size, locality can be optimized with additional service-specific knowledge. For example, Figures 4.4(a) and (b) indicate that compared to 54.73% of requests hitting 30% of the regions, when the logical data view is partitioned along the right ascension coordinate (`X`), 77.84% of requests hit on 30% of the regions when the view is partitioned along the declination coordinate (`Y`). A similar observation also holds for the *tile.ashx* trace (Figures 4.6(c) and (d)).

Impact of time epoch size.

To understand whether the locality structures develop over long timescales or are even present over short timescales, we examined how the load fraction graph varies with different time epoch sizes. The *x_rect.asp* trace, which contains few requests for shorter time epochs, shows larger variations in locality patterns for smaller time epochs than the *tile.ashx* trace. For the former, Figures 4.4(d), 4.5(e) and 4.5(f) show that 10% of the address groups contribute only 8.42% of requests for a daily timescale while this fraction goes up to 99.63% for a weekly timescale and 99.95% for a monthly timescale. For TerraServer's *tile.ashx* service, 10% of the address groups continue to contribute to at least 80% of the overall requests even for hour-long epochs (Figure 4.6(b) and Figure 4.7(g)).

The observations imply that locality is in fact affected by the size of the time epoch, with additional service-specific information possibly being required to guide a service replication infrastructure in its choice of an appropriate epoch size at which to detect and optimize locality.

Deviation of measured locality.

To understand the variations in locality over different time epochs, we examined three kinds of statistics for the load fraction, the minimum, average and maximum values, over the entire trace duration. Contributing to these statistics were the load fractions measured at each time epoch. The results indicate that the *x_rect.asp* trace exhibits larger deviations of load fraction compared with the *tile.ashx* trace. This observation likely results from the smaller number of requests seen over smaller time epochs in the SkyServer trace.

4.1.5 Discussion

Locality in general network services. We have classified the locality of usage patterns for data-centric network services across three dimensions and demonstrated the existence of these three kinds of locality in two well-known public services, the SkyServer and TerraServer. We expect that such locality usage patterns exist widely in most maps/imagery services such as Microsoft's MapPoint service, MapQuest service, etc. For general data-centric network services, it is not necessary that all of these three kinds of locality will exist in the service usage patterns. However, one can expect the existence of a subset of the locality. For example, *spatial* and *temporal* locality can be found in online bookstore services like Amazon.com (e.g. the top-selling book list), news services like CNN.com (e.g. the breaking news, live-video), and web searching services like Google Web APIs (e.g. the popular keywords or phrases).

Additionally, although not discussed in the two investigated services due to the required information not presented in the web-traces, one might expect to find other kinds of locality beyond the three ones defined in this section. For example, one can differentiate the service usage patterns according to the client device used for the request or the network bandwidth constraints, e.g., clients with hand-held devices and clients with desktops might demonstrate different service usage patterns for maps/imagery services, or clients with low-bandwidth constraints and clients with high-bandwidth constraints might demonstrate different service usage patterns for multimedia services. Since these kinds of locality usually demonstrate themselves on the application level, we refer to them in the rest of the document as *application-preference locality*.

In the second part of this chapter, we will describe how DataSlicer detects the spatial, network and temporal locality via in-network request inspection. The application-preference locality is accommodated in DataSlicer using a novel overlay construction scheme and we defer its discussion until Chapter 5.

Previous approaches in workload characterization. Our work has focused on characterizing network service requests in terms of what extent of locality they exhibit with respect to the data regions of the back-end database involved in response generation, the network regions where the requests originated, and the associated time epochs.

Pitkow [79] has presented a survey of web characterization studies and there are many other detailed studies that have examined web workloads [80, 81, 82, 83, 84, 85]. However, most of these works have focused on characteristics of static web content, and their results do not directly apply to the case of dynamic web content. Relatively few studies [86, 87, 65, 88] have examined the characteristics of dynamic content: these studies have verified both the need for and likely benefit from caching content at sub-document granularity and moving content generation to the network edge; both these ideas can be thought of as a special case of service replication infrastructures that provide the motivation for the work discussed in this thesis.

Our work extends the above studies to the context of data-centric web services. Given this context, our definitions of temporal, spatial, and network locality differ somewhat from those used in previous studies [81, 83, 84, 85], but retain a similar spirit. From the methodology viewpoint, most notable about our approach is its use of logical views to model accesses against the back-end database and the employment of load fraction graphs as a means to quantitatively describe different kinds of local-

ity. Logical views were influenced by the notion of semantic regions [71] from the database caching literature, where they refer to a range of relation values accessed by a group of queries (requests). The difference in our work is that such regions may only be *virtual*, serving to specify the internal service data required for servicing a group of requests.

4.2 In-network Request Inspection and Locality Detection

Given the existence of locality in usage patterns of data-centric network services, it is desirable that DataSlicer architecture be able to detect such locality information to permit proper actions to be initiated. This section describes how DataSlicer dynamically inspects underlying traffic and discovers service usage locality patterns at various network intermediaries. We refer to the maps service example introduced earlier to illustrate various aspects of this support. The inspection process requires the active involvement of the service providers to provide service-specific information to our infrastructure. In this section, we also introduce a dynamic data structure, called *Cell*, that we use to efficiently aggregate client requests at multiple levels of granularity to infer locality patterns.

4.2.1 Service Registration

The DataSlicer architecture is intended to be used as a distributed hosting platform: services register with DataSlicer, and as part of the registration step supply the required information, which is then made available to all routers. Such information includes service interfaces (e.g., the SOAPAction attribute of a SOAP request to the

service, or the URL information in a HTTP GET/POST request), the underlying logical data space for the service, the mapping scheme that maps a service request into a region in the data space, and the desired performance metrics. This step also identifies (1) the origin servers, who are responsible for organizing the participating routers into an oriented overlay network; and (2) the entry routers, who publish, using UDDI or a similar protocol, their ability to receive service requests.

The service interface information is used by DataSlicer routers to identify the requests for the registered service from the underlying traffic. The logical data space is specified in a straightforward fashion, in terms of its dimensionality, the attributes corresponding to these dimensions, and the value ranges taken by these attributes. Such information is used to construct a *Cell* structure to dynamically infer the service usage locality. To translate arbitrary service requests into regions of this data space, we require the service provider to provide a black-box functional module which serves as a transformer.[2] In generally, these transformer modules are implemented as a dynamic link library that gets loaded at runtime into DataSlicer routers to perform the transformation. For SOAP-based service requests, an alternative solution is possible, which leverages the XML-based nature of SOAP messages: the service specifies XSL stylesheets that are used by XSLT to transform each service request (identified by the corresponding SOAP action). The first approach is very general, but requires the service provider to trust the DataSlicer architecture and additionally, needs knowl-

[2] A service request might contain a sequence of parameters corresponding to a complicated relational schema of the back-end database. The interactions between the service and the database could also involve sophisticated computational logic. Usually, the service provider would not expose such information to the public, which is why we assume that it bundles the required information into a stand-alone module.

edge about the router platform. This latter solution has the advantages of alleviating security concerns because XSLT is a safe language (modulo third-party extensions, whose use can be controlled), and of supporting the transformation in a language and platform-neutral fashion.

We consider some examples of the registration step and associated transformation modules below:

Example 1: MapPoint Service. Our example maps service permits clients to retrieve location-based maps, using a service request called *GetMap*. Figure 4.8 shows the interface information for this service.

According to the service WSDL document, the *GetMap* operation is bound to the following SOAPAction, "http://s.mappoint.net/mappoint-30/GetMap". *GetMap* takes a *MapSpecification* object as an input and creates one or more rendered images of a map. Among the elements of the *MapSpecification* object, the *DataSourceName* defines a string representing the name of the data source, e.g. "MapPoint.NA", which contains the base map of North America that supports finding addresses in Canada, the United States, and Puerto Rico; the *MapOptions.Format* defines the size of the map image; and the *Views* element defines an array of map views (one of the *ViewByScale*, *ViewByHeightWidth*, and *ViewByBoundingRectangle*) to render. Together, the elements define a particular region in the back-end maps database: a specific data source, a map resolution, and a geographic rectangle on earth. Therefore, one can construct a simple logical data space for the *GetMap* service which contains three dimensions: map resolution, latitude, and longitude. The transformer module should be able to translate a *GetMap* service request into a region in such a logical data space.

```
<s:complexType name="MapOptions">
 <s:sequence>
  <s:element minOccurs="1" maxOccurs="1" name="RouteHighlightColor" type="tns:RouteHighlightColor" />
  <s:element minOccurs="1" maxOccurs="1" name="ConstructionDelayHighlightColor" type="tns:RouteHighlightColor" />
  <s:element minOccurs="1" maxOccurs="1" name="ConstructionClosureHighlightColor" type="tns:RouteHighlightColor" />
  <s:element minOccurs="0" maxOccurs="1" default="Smaller" name="FontSize" type="tns:MapFontSize" />
  <s:element minOccurs="0" maxOccurs="1" name="Format" type="tns:ImageFormat" />
  <s:element minOccurs="0" maxOccurs="1" default="false" name="IsOverviewMap" type="s:boolean" />
  <s:element minOccurs="0" maxOccurs="1" default="ReturnImage" name="ReturnType" type="tns:MapReturnType" />
  <s:element minOccurs="1" maxOccurs="1" name="PanHorizontal" type="s:double" />
  <s:element minOccurs="1" maxOccurs="1" name="PanVertical" type="s:double" />
  <s:element minOccurs="1" maxOccurs="1" name="Style" type="tns:MapStyle" />
  <s:element minOccurs="0" maxOccurs="1" default="1" name="Zoom" type="s:double" />
  <s:element minOccurs="0" maxOccurs="1" default="false" name="PreventIconCollisions" type="s:boolean" />
 </s:sequence> </s:complexType>

<s:complexType name="MapViewRepresentations">
 <s:sequence>
  <s:element minOccurs="0" maxOccurs="1" name="ByScale" type="tns:ViewByScale" />
  <s:element minOccurs="0" maxOccurs="1" name="ByHeightWidth" type="tns:ViewByHeightWidth" />
  <s:element minOccurs="0" maxOccurs="1" name="ByBoundingRectangle" type="tns:ViewByBoundingRectangle" />
 </s:sequence> </s:complexType>

<s:complexType name="MapSpecification">
 <s:sequence>
  <s:element minOccurs="0" maxOccurs="1" name="Polygons" type="tns:ArrayOfPolygon" />
  <s:element minOccurs="0" maxOccurs="1" name="DataSourceName" type="s:string" />
  <s:element minOccurs="0" maxOccurs="1" name="HighlightedEntityIDs" type="tns:ArrayOfInt" />
  <s:element minOccurs="0" maxOccurs="1" name="HideEntityTypes" type="tns:ArrayOfString" />
  <s:element minOccurs="0" maxOccurs="1" name="Options" type="tns:MapOptions" />
  <s:element minOccurs="0" maxOccurs="1" name="Pushpins" type="tns:ArrayOfPushpin" />
  <s:element minOccurs="0" maxOccurs="1" name="Route" type="tns:Route" />
  <s:element minOccurs="0" maxOccurs="1" name="Views" type="tns:ArrayOfMapView" />
 </s:sequence> </s:complexType>

<s:element name="GetMap">
 <s:complexType> <s:sequence>
  <s:element minOccurs="0" maxOccurs="1" name="specification" type="tns:MapSpecification" />
 </s:sequence> </s:complexType>
</s:element>

<wsdl:message name="GetMapSoapIn">
 <wsdl:part name="parameters" element="tns:GetMap" />
</wsdl:message>

<wsdl:portType name="RenderServiceSoap">
 <wsdl:operation name="GetMap">
  <documentation xmlns="http://schemas.xmlsoap.org/wsdl/"> ... </documentation>
  <wsdl:input message="tns:GetMapSoapIn" />  <wsdl:output message="tns:GetMapSoapOut" />
 </wsdl:operation> </wsdl:portType>

<wsdl:binding name="RenderServiceSoap" type="tns:RenderServiceSoap">
 <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document" />
  <wsdl:operation name="GetMap">
   <soap:operation soapAction="http://s.mappoint.net/mappoint-30/GetMap" style="document" />
   <wsdl:input> ...  </wsdl:input>   <wsdl:output> ... </wsdl:output>
  </wsdl:operation> </wsdl:binding>
```

Figure 4.8: *GetMap* specification

Types:

```
service_id   :: unsigned integer
service_op   :: SOAPAction | HTTP URL
origin_site  :: host[:port]
origin_sites :: origin_site [;origin_site]
db_attr      :: attr_name
                attr_type
                min_val
                max_val
db_view      :: db_attr[;db_attr]
transformer  :: dll | XSLT program
```

Operations:

```
service_id service_register ( service_op svc, origin_sites sites, db_view view, transformer trans );
service_id service_find ( service_op svc );
void service_unregister ( service_id sid );
```

Figure 4.9: DataSlicer Service Registration Interface

At the service registration stage, one registers the *GetMap* service with DataSlicer using (1) the SOAPAction attribute, “http://s.mappoint.net/mappoint-30/GetMap”; (2) the origin websites that host this service; (3) the above 3-dimensional logical data space including the name, data type and value range of each dimension; and (4) the transformer module. Figure 4.9 shows a portion of this registration interface.

Example 2: SkyServer Service. In the SkyServer site, an intensively used service is *x_rect.asp*, which takes as input a rectangle in the sky (specified by the rectangle center, width, and height expressed in terms of the sky coordinates of right ascension and declination) and five optional optical bands, and returns a list of objects found in that rectangle. Requests to the *x_rect.asp* are HTTP requests and have the following form: “http://cas.sdss.org/dr4/en/tools/search/x_rect.asp?*query_str*”. The former portion (before the question mark) is the URL of the *x_rect.asp* service. The latter portion,

query_str, consists of a collection of parameters for the input described above.

At the service registration stage, one registers the *x_rect* service with DataSlicer using (1) the URL information, “http://cas.sdss.org/dr4/en/tools/search/x_rect.asp”; (2) the origin websites that host this service; (3) a 2-dimensional logical data space that corresponds to the sky coordinate system; and (4) a transformer module which translates the *query_str* into a rectangle in the sky coordinate system.

As with the MapPoint service, the transformer module can be implemented either as a DLL or an XSLT program. An XSLT-based implementation is more preferable due to its language-safety nature and its platform-independency. However, for the SkyServer service, due to the lack of information about the type and structure of the supplied parameters in a query, the transformer module is implemented as a DLL.

Once a service is registered, the service-specific information is made available to all of the DataSlicer routers, who use this information to construct service handlers (as shown in Figure 2.2) to perform functions such as request inspection, locality detection and service replication, etc. In the next section, we introduce an important data structure called *Cell*, which is used by all of these functions.

4.2.2 Cell Structure

To model service usage locality in accessing the back-end database, one needs to maintain the access statistics for requested database content at different levels of data granularity. Traditionally, the maintenance of usage statistics is handled by keeping track of the most popular requests and their responses in a cache-like structure, where the contents are indexed by the request queries. However, such an approach is not desirable in the network services context because (1) it is unlikely that a response

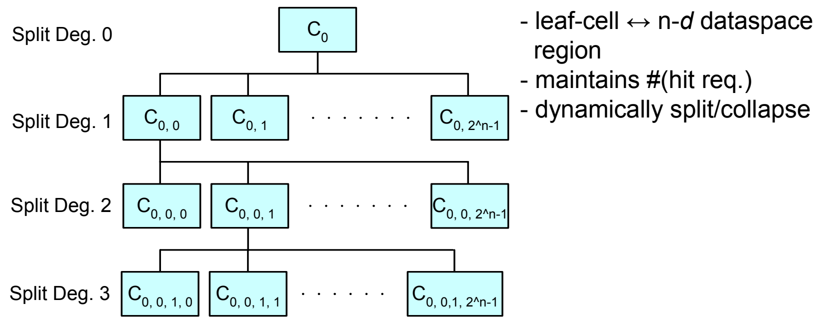


Figure 4.10: A Dynamic Data Structure: *Cell*

to a previous request is reusable due to the variation of values and forms of request parameters; and (2) simple record keeping means that the routers need to store a high volume of requests (e.g. millions of requests per day for the TerraServer service), which leads to storage capacity concerns and results in inefficiency in analyzing the statistics at run-time. To address these issues, we employ a dynamic data structure, called *Cell*, which is shown in Figure 4.10

Informally, a "cell" is a representation of a hyper-region in the logical service's data space, assuming that this data space is multi-attributed, and that the attributes are numerical or alphabetical rangeable. A cell maintains usage statistics about the corresponding region of the service's data space: these include the number of requests hitting the region over a time period, and the average service time seen by these requests (at a particular router).

When the number of requests hitting a cell exceeds a threshold, the cell can be *split* into a set of disjoint sub-cells, each of which covers a smaller region of the data space. Subsequent statistics are only maintained at the level of the sub-cells, each of which starts off with an equal share of the parent cell's hit count. Similarly, when

the hit count of all sub-cells drops below a threshold, the sub-cells can be *collapsed* back into the parent cell. Thus, at any point in time, a router maintains a cell tree, whose leaf nodes divide the service's data space into a set of disjoint regions. The split and collapse operations permit efficient maintenance of statistics for different locality patterns involving coarse as well as fine-granularity regions. These statistics are periodically reset in our current implementation, but can as easily be aggregated to collect information over multiple timescales.

The Cell structure is also used to keep track of service replication information. In DataSlicer, the replication of a service occurs at the leaf-cell level, i.e., the router maintains the service performance for a group of requests that hit a particular leaf-cell, and will request that a region corresponding to the leaf-cell be replicated if the router detects that the performance of that leaf-cell needs to be improved. Once a replica is created at some network location, the affected routers (the ones that have a path leading towards to that replica) will update their cell structure with the replication information such that subsequent requests targeting the replicated cell can be relayed to the replica. Details about the service replication and request redirection procedures are discussed in Chapter 6.

4.2.3 Request Processing

After a service is registered with the DataSlicer infrastructure, clients can access the service by sending requests to the published *entry routers*, from which the requests get relayed through the constructed oriented overlay network across one or more *intermediate routers*, and then forwarded to the origin service servers or replicas. The responses follow the same path in the reverse direction back to the clients. DataSlicer

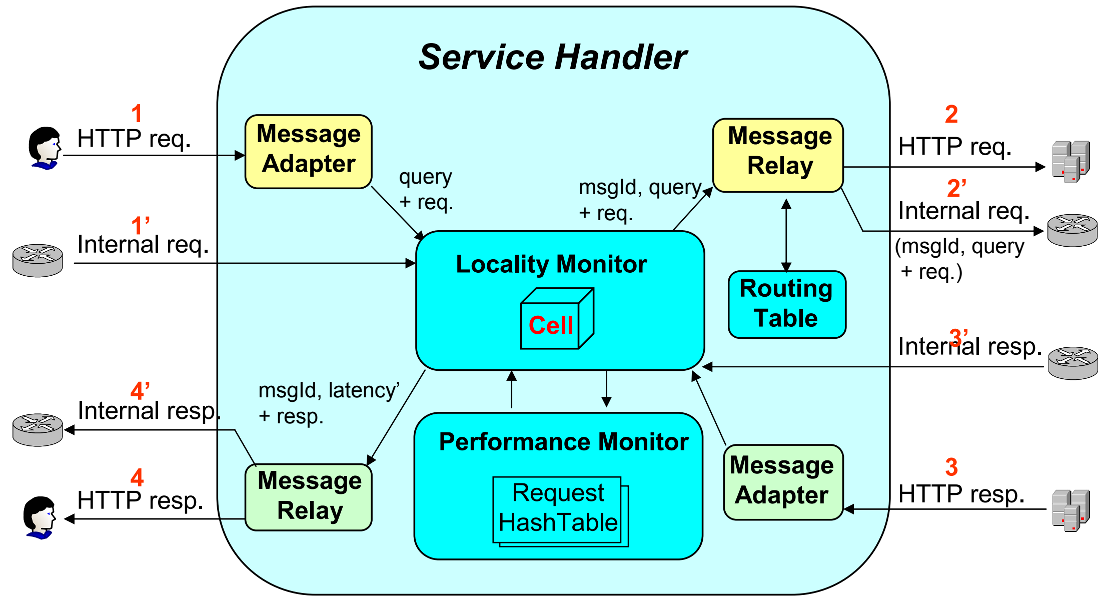


Figure 4.11: Service Request Processing at DataSlicer Routers

routers aggregate information about service requests into the Cell structure and continually monitor the performance seen by a request group.

Each request traversing the router is inspected as shown in Figure 4.11, using multiple functional components. In the figure, the request relaying path from a client to a service replica via a single router is illustrated by the steps "$1 \rightarrow 2 \rightarrow 3 \rightarrow 4$" and the traversal across multiple routers is illustrated by "$1 \rightarrow 2' \rightarrow 3' \rightarrow 4$". We refer to the requests sent in steps 1 and 2 as *origin* requests, and the requests sent during steps $1'$ and $2'$ as *wrapped* requests. Similarly, the responses sent in steps 3 and 4 are called *origin* responses, and the responses sent during steps $3'$ and $4'$ as *wrapped* responses. An incoming origin request is passed by the router to a *Message Adapter* module defined by the corresponding service handler. The adapter extracts mapping informa-

tion from the message body, and generates a wrapped message, which flows through the *Locality Monitor* before being relayed along the path using the *Message Relay* module. Each request kicks off a timer in the *Performance Monitor*, which tracks the response time seen by the request. The Message Relay module interacts with the *Routing Table* module to allocate link resources to the uplink router or replica, and forwards either the wrapped message or the origin request (if the upstream entity is a replica). Incoming responses are similarly handled, with the first Message Adapter module wrapping the response, stopping timing in the Performance Monitor, and relaying the wrapped response or the origin response to a downstream router or end client respectively. A response message is correlated with a request message using a unique ID, which is remembered by the Routing Table module.

4.2.4 Discussion

Our approach for tracking service locality patterns is most influenced by the work of Semantic Caching [71, 72, 73, 74, 75]. Semantic Caching approaches manage the cached data using the concept of Semantic Regions, which consist of a set of relevant tuples that are defined and dynamically adjusted based on the requests originating from clients. Our dynamic Cell data structure represents a multi-dimensional data space defined by the back-end database and views a particular client request as accessing a small region in such a data space: the leaf cells represent a partition of the data space and play the same role as semantic regions in past work.

4.3 Summary

This chapter described our characterization of the workload of data-centric network services in terms of three kinds of locality: *temporal*, *spatial*, and *network*. Our analysis, based on web-traces from the SkyServer and the TerraServer sites, indicates that both workloads exhibit high spatial and network locality across multiple time epochs. The positive findings of locality across multiple dimensions points to the benefits that are likely from DataSlicer-like architectures. In the second part of this chapter, we have described what kind of service-specific information needs to be registered with the DataSlicer infrastructure and discussed how to perform in-network service request inspection and locality detection using the dynamic Cell data structure. In the next two chapters, we will describe how to construct an oriented overlay network to accommodate locality detection and how to leverage such locality information to determine appropriate service replication strategies.

Chapter 5

Oriented Overlays Construction and Maintenance

DataSlicer's ability to dynamically inspect underlying traffic and discover service usage locality patterns at various network intermediaries contingent upon the underlying overlay network being able to yield "good" *clustering* of client requests. This chapter describes the detailed design and implementation of our approach, a "zone-based" oriented overlays construction scheme, which addresses this challenge. We defer a discussion of its performance to Chapter 8.

5.1 "Zone-based" Oriented Overlays

The intermediaries in the DataSlicer architecture define an *oriented overlay* network over which client requests from many sources are routed to the origin server; intermediaries can inspect requests routed through them and cooperate with one another to

ascertain usage locality characteristics. Note that the overlay routing scheme closely affects both whether or not locality is detected and whether such locality can be profitably exploited: a good oriented overlay would offer sufficient *clustering* ability for similar requests (for application defined notions of similarity) without adversely affecting other metrics of interest such as path latencies or effective network bandwidth. The term "clustering" is used differently in our work than in topology-aware overlays: we are not attempting to provide connectivity among grouped nodes, but to provide a merge point in the network where requests from clients that are "close" to each other and share similar application-specific preferences can be grouped together and inspected for service usage locality.

This chapter focuses on the question of how to scalably construct such oriented overlays, whose requirements differ from those targeted by the existing methods for constructing scalable peer-to-peer (P2P) overlay networks. We assume that our overlays will involve on the order of 10^0 — 10^2 origin server(s) and 10^2 — 10^4 participating nodes (note that the latter number refers to the number of intermediaries, not the end-clients which may be much larger). The construction scheme is capable of clustering nodes to form oriented overlays using multiple application metrics. For simplicity, we start by describing our construction scheme using a single metric — network proximity, and then discuss its extensions to support multiple metrics.

5.1.1 Basic Design

To construct the overlay network, each participating node runs a protocol to communicate with others and feeds the collected information into a centralized or distributed algorithm that organizes the nodes into a logical topology based on the desired clus-

tering metric (which we initially assume to be network proximity).

To cluster client requests at various points in the network, without adversely affecting the path latencies, we propose a *zone-based* scheme. A *zone* defines a range of distances (in terms of network latency) from an origin server (which we assume to be reliable and located physically close to the origin website): the higher the level of a zone, the farther away from the origin server it is. According to their distances from the origin server, the participating nodes are partitioned into different zones. Each participating node then selects one or more parents to connect to, forming an overlay. The parent selection has an orientation "bias" towards the origin server: (1) a participating node *A* can only select another node *B* as its parent if *B* resides in a lower level zone; (2) the candidate parents for node *A* come from the nodes which reside in *A*'s next non-empty lower zone; and (3) to avoid adversely introducing additional overhead on latency for the path from *A* to the origin server, *A* usually selects the closest node(s) from the candidates as its parent(s).

Figure 5.1 illustrates a desirable overlay for a data-centric service with a single origin server. In this illustration, nodes 1 and 2 that are close to each other share a common pattern when accessing the service. Similarly, nodes 4 and 5 share another pattern. Node 3, located between these two groups, shares both patterns. Using our zone-based scheme, the participating nodes might be partitioned into three zones according to their distances from the origin server. Ideally, nodes 1 and 2 will be directed to a node like node 6, because 6 provides a shorter path to the origin server, compared to alternatives like node 8. On the other hand, node 3 would not be selected as a parent for nodes 1 and 2 because 3 belongs to the same zone and could potentially adversely increase the path latencies from node 1 (or 2) to the origin server. The example over-

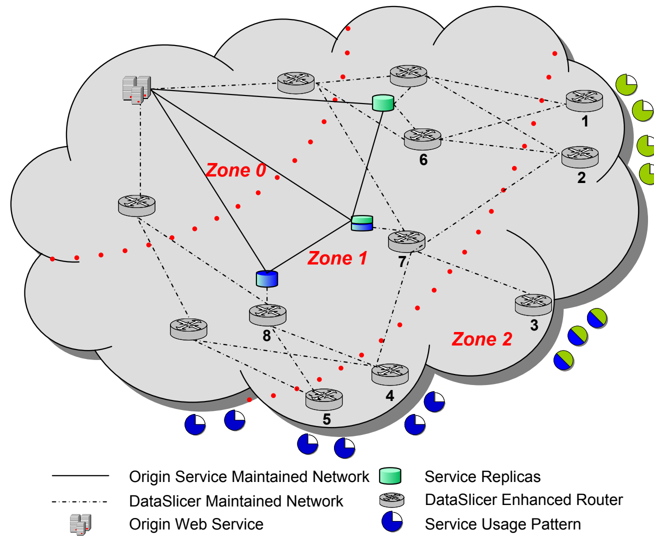


Figure 5.1: Overview of an oriented overlay for data-centric network services. The slim dotted lines show the connections of the constructed overlays.

lay demonstrates an advantage that the intermediate nodes can easily detect locality in client requests and hence allow actions such as service replication to be taken. For example, node 6 is able to detect the similarity in service usage patterns from nodes 1 and 2 and create a replica nearby node 6 which holds only a portion of the data corresponding to that usage pattern.

To support building oriented overlays in situations where there are multiple origin servers, our algorithm allows each origin server to have its own overlay which consists of a disjoint subset of participating nodes.[1] At startup, each node selects the closest origin server and participates in only that origin server's overlay construction, assuming that this overlay potentially provides the best path latency. Since the net-

[1] Note that a single physical node can still participate in multiple overlays by appearing as multiple virtual nodes.

work status changes dynamically, we allow a node to switch to another origin server's overlay from the current one if it detects that the new origin server is closer.

5.1.2 Construction Protocols

A node needs to take two important steps to join in our overlays: *Node Startup* and *Parent Selection*.

Node Startup. A node joins in the system by first registering itself to the closest origin server. To do so, the node probes the latencies between itself and all of the origin servers, selects the one with the smallest latency, and passes this information to that chosen origin server in a *node_join* request. Upon receiving the *node_join* request, an origin server extracts the latency information from the request message, computes the rank of the zone that this node belongs to, and assigns the rank to this node. As its response, the origin server sends the assigned rank, together with an advised candidate parent list, back to that node.

Each origin server maintains a list of the participating nodes and their zone-ranks.

Parent Selection. The parent selection algorithm is designed to ensure that paths are chosen with an orientation "bias" towards an origin server. When the origin server receives a *node_join* request from a node, it responds with an assigned zone rank and a list of advised parent candidates with lower ranks. The node then probes the latencies between itself and these parent candidates and selects K nodes with minimum latencies as its parents, where K, a configuration parameter, is the maximum number of parents that a node can have.

To avoid overload on some intermediate nodes, we also impose a restriction on the maximum number of children that an intermediate node can have. Hence, a node needs to communicate with its selected parent node first to confirm that indeed that node can serve as its parent.

5.1.3 Maintenance Protocols

A good overlay should adapt itself to the dynamic changes of the underlying network conditions, as well as nodes joining and leaving. Key to this adaptation is the ability to effectively detect the changes and propagate such information to the affected nodes.

Origin Server. The origin server receives four kinds of messages for overlay maintenance: *node_join*, *node_update*, *node_leave* and *node_dead*. The first is sent by a new joining node, which registers itself to join in the overlay; the second is sent by a node which is already participating in the overlay and periodically updates the probed latency to the origin server; the third is sent by a node which has determined that it wants to switch to another overlay; and the fourth comes from a node which reports that another node is "dead" because of a probing failure.

The origin server handles the first two types of messages by inserting a new record into its maintained list of participating nodes if the sender does not exist, otherwise, it just updates the information appropriately (e.g., update the assigned rank for the sender). The origin server then sends the rank of the sender and an advised list of candidate parents back to the sender. For the third type of message, the origin server removes the node from the maintained list and notifies the affected nodes (the nodes at zones immediately higher than the node which has left). For the fourth type of

message, the origin server does not eagerly remove the node reported as dead. Instead, it marks that node by setting a timer and increases a counter which keeps track of the number of times that node has been reported as dead. In the case that either the counter exceeds a threshold or the timer expires, the node then is removed and notifications are sent to the affected nodes. The counter and the timer will be reset if either a *node_join* or a *node_update* message is received from the suspected node before the timer expires.

Participating Node. Each node maintains a list of all origin servers and the latencies between itself and these origin servers. At startup, a node selects the closest origin server to participate in its overlay construction. Periodically, a node probes all origin servers to update the latencies and switches to a new overlay if there exists a closer origin server. In the case that a switch happens, a node sends a *node_leave* message to the origin server in its current participating overlay, and then sends a *node_join* message to the newly selected origin server. The node then needs to re-run the parent selection algorithm in the new overlay.

After a node joins in a particular overlay, it maintains a candidate parent list (advised by the origin server) and the latencies between itself and these candidates. Periodically, the node probes the origin server and sends information about the latest observed latency in a *node_update* message. Upon receiving a response from the origin server, the node merges the advised candidate list in the response with its own copy by (1) removing nodes from the current list that are not in the new one; (2) for nodes that are in the new list but not in the old one, probing these nodes and inserting them into the current list.

Inputs:
S: set of origin servers
K: number of parents a node wants to connect to
N: number of children an intermediate node allows
D: threshold on times a node is reported as being dead

n, m: overlay node
$l_{n,m}$: round-trip latency between node n and m
rn: zone rank of node n assigned by an origin server
C_n: node n's candidate parents, advised by an origin server
P_n: parent list selected by node n
L: list of participating nodes maintained at an origin server

Origin Server (s):

upon a join/update request: *(n, $l_{n,s}$)*
compute r_n for node n using $l_{n,s}$
$r := r_n - 1$
while C_n is empty and $r \geq 0$
  add $m \in L$ into C_n for all m where $r_m = r$
  $r := r - 1$
if C_n is empty
  add the origin server into C_n
send (r_n, C_n) to node n

if $n \in L$
  update the rank of n with r_n
  reset *n.counter* and *n.timer*
else
  insert n into L

upon a node_leave request: (n)
  remove n from L
  foreach m where $r_m = r_n - 1$
    notify m that n has left

upon a node_dead message: (n)
  increase *n.counter*, $n \in L$
  set *n.timer* if not set
  if *n.counter* > D or *n.timer* expires
    remove n from L
    foreach m where $r_m = r_n - 1$
      notify m that n is left

Node Startup (*n*, *S*):
probe the round-trip latencies $l_{n,s}$ for all $s \in S$
select the closest s to send a join request (n, $l_{n,s}$)
receive a response (r_n, C_n) from s and run parent selection

Parent Selection (*n*):
probe $m \in C_n$ and sort C_n by $l_{n,m}$
$i := k := 0$
while $k \leq K$ and $i < |C_n|$
  send a *parent_sel* request to $C_n[i]$
  if $C_n[i]$ grants the request
    $P_n := P_n \cap C_n[i]$
    $k := k + 1$
  $i := i + 1$
establish connections to the selected parents

Participating Node (*n*):
upon a parent_sel request from m
if m exists in child list
  grant m's request
else if number of children is less than N
  grant m's request and add n into child list
else
  reject m's request

upon a parent_cancel request from m
remove m from child list

upon a node_dead (m) message from s
remove m from C_n and P_n

Overlay Switching (*n*, *s*, *s'*):
send a leave request (n) to s
send a join request (n, $l_{n,s'}$) to s'
receive a response (r_n, C_n) from s'
re-run parent selection

Node Maintenance (*n*, *s*):
periodically, probe $s' \in S$
if exists $s' \in S$, s.t. $l_{n,s'} < l_{n,s}$
  switch to the overlay oriented towards s'

periodically, randomly select $C_n' \subseteq C_n$
foreach $m \in C_n'$
  probe $l_{n,m}$
  if fail
    remove m from C_n and P_n
    send *node_dead(m)* to s
sort C_n in ascending order by $l_{n,m}$
replace P_n with first K nodes in C_n that can be n's parent
establish connections to the selected parents

Figure 5.2: Distributed algorithm for construction and maintenance of oriented overlays (Independently run for each origin server)

Each node maintains the latencies between itself and its candidate parents by periodically probing a random subset of the candidates, and updates its parent selection if there exists any candidate that can accept new children and has smaller latency than any of its chosen parents. If any of these probes fails, the node reports the failure to the origin server with a *node_dead* message.

There are four types of messages used to exchange information between the participating nodes: *parent_sel*, *parent_cancel*, *parent_grant* and *parent_reject*. A node can only select a candidate as its parent by first sending a *parent_sel* message to and receiving a *parent_grant* message from that candidate. If the contacted candidate finds that its number of children has reached the threshold, it responds to the *parent_sel* request with a *parent_reject* message. The *parent_cancel* message is used in situation where a node changes its parent selection by replacing an already-selected parent with a newly selected, providing the latter has the smaller latency. Upon receiving a *parent_cancel* message, a node removes the sender from its child list.

Figure 5.2 shows the detailed actions taken at the nodes during various stages of the oriented overlays construction scheme.

5.1.4 Overlay Properties

Our zone-based overlay construction scheme is (1) relatively simple — no support from any external measurement infrastructure is needed; (2) efficient — an origin server acts only as a rendezvous point by maintaining the participating nodes and advising about the candidate parent lists, with the result that a node joining in the system need only query the origin server once, following a small number of probes; (3) distributed — the parent selection and maintenance are pair-wise distributed algorithms;

and (4) incurs minimal communication cost — the traffic attributed to our overlay construction and maintenance is substantially lower than that encountered by other unstructured overlays relying on network floods.

Our oriented overlays are also robust in the face of high network-churn because a node that has left the system can be detected quickly with high probability and reported to the origin server, which in turn propagates this information to all of the affected nodes. Node-leaving does not really impact the connectivity of our overlays because: (1) a node cannot reach the origin server only if all of its paths towards the origin server are broken; and (2) a node can select a new parent (if available) to replace the leaving one quickly.

The impact on latency dilations for overlay paths between a participating node and the origin server is minimal because a node's candidate parents always reside in a lower level zone and the parent selection algorithm selects the closest candidates as parents. In this way, we ensure that a path is constructed with a strong orientation "bias" towards the origin server with minimal latency overhead being introduced.

5.1.5 Extensions

Accurate Positioning. Our overlays position the participating nodes in the network using the measured latencies between the nodes. Given that network latency measurements only approximate network proximity, nodes that end up being clustered in the built overlays could in fact be rather far away from each other. Such inaccuracies can affect the goodness of clustering and dependent decisions. Obviously, if additional information such as node coordinates are available (PlanetLab nodes do possess this information), a participating node can provide its position information

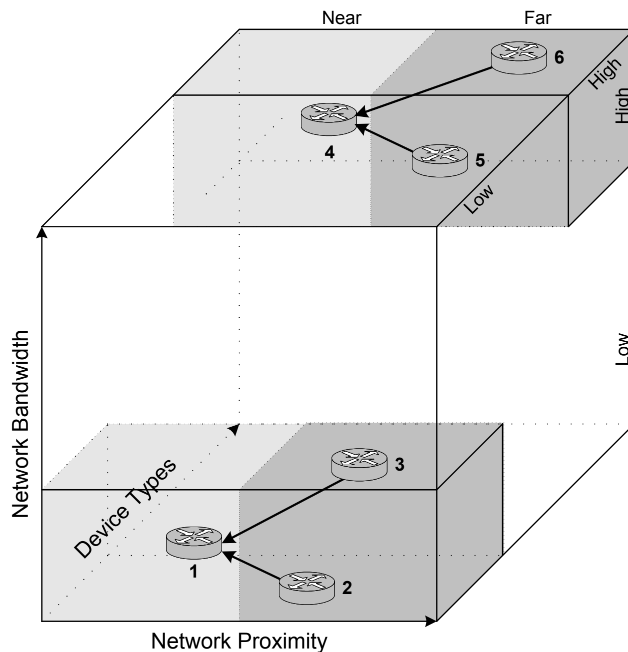


Figure 5.3: View of hyper-zone in oriented overlays with multi-metrics.

to the origin server during the registration step, which can take this information into account when assigning nodes to different zones.

Support for multiple application-specific metrics. The zone-based overlays construction scheme can be easily extended to support more general application-specific metrics, e.g., preferences based on client device types or network bandwidth requirements. The main idea is to allow intermediate nodes to cluster nodes that exhibit similarity in the involved metrics.

We first extend our *zone* structure described earlier to support a multi-dimensional view of the involved metrics: assuming each of the involved metrics is numerically or alphabetically rangeable, we partition the metric-space into *hyper-zones*, each hyper-

zone corresponds to a specific combination of metric-preferences. Similarly, the participating nodes are partitioned into such hyper-zones according to their preferences for various metrics. Figure 5.3 shows an example of an oriented overlay that involves three metrics: network proximity, network bandwidth and client device types. In this example, the two groups (the one that consists of nodes $1, 2, 3$, and the other that consists of nodes $4, 5, 6$) have different preferences in terms of network bandwidth and device types. In each group, one node (node 1 in the first group and node 4 in the second group) is closer to the origin server compared with the other two. Ideally, our overlay scheme would cluster nodes $2, 3$ at node 1 and nodes $5, 6$ at node 4 to facilitate locality detection.

To support multi-dimensional zones, our overlay construction scheme is extended as follows: (1) the origin server maintains a multi-dimensional view of the overlay in terms of the involved metrics; (2) at startup, each participating node can either explicitly report its metric preferences to, or receive a default assignment from the origin server; (3) the origin server sends out a list of advised candidate parents, including their metric-preferences, to the participating nodes; and (4) the parent selection algorithm is extended to evaluate the "goodness" of candidates in such a multi-dimensional metric-space (with respect to clustering related requests).

For the last extension, we use a scoring system which represents overall "goodness" as a linear combination of the "goodness" of each metric: (1) each metric m has its own evaluation rule in terms of the difference between the preference values associated with the participating node and the candidate being evaluated; however, we require that the evaluated score s_m range between 0 and 1; (2) each metric is associated with a weight w_m, $\sum w_m = 1$ for all m; and (3) the overall "goodness" score

is computed as: $S = \sum w_m \times s_m$. The advantage of such a linear scoring system is to allow our overlay construction scheme to easily support a wide variety of scenarios where multiple application-specific metrics are involved.

5.2 Implementation

We have implemented and evaluated our construction scheme for building oriented overlays on the PlanetLab network. The overall implementation effort is relatively modest: the total length of the source code (in C) is about 3,000 lines. The overlay construction scheme is integrated in our DataSlicer infrastructure to cluster client requests across the wide-area network.

On the server side, our server program listens on a public port to receive messages from the participating nodes and processes these messages in an event-driven fashion. There is a tradeoff between notifying the participating nodes with the up-to-date overlay status and reducing the communication cost in overlay maintenance: in the implementation, the server program does not send a response to each *node_update* request; instead, the server periodically (every 300 seconds) sends its advised candidate parent list to each participating node based on the latest information of its overlay. The advantage of doing so is clear: the traffic resulting from overlay maintenance is significantly reduced from $O(N^2)$ to $O(N)$, where N is the number of nodes that participate in the overlay.[2]

On the participating node side, our client program also listens on a public port

[2] Assuming that nodes are evenly partitioned into each zone, for a *node_update* request from a node in an intermediate zone, the origin server needs to sends out $O(N)$ notification messages. The communication cost is therefore $O(N^2)$. In our implementation, an origin server integrates all of the changes that happen to the overlay during a period, and only needs to send out $O(N)$ messages.

to receive messages from the server(s) and other nodes in an event-driven manner. Every 30 seconds, the client program probes all of the origin servers and a randomly selected subset of its candidate parents. Based on the probe results, the client program takes appropriate actions such as switching overlays, reporting liveness of nodes, or changing its parent selection.

To support the client and server programs above, we use a coordinator program running on a reliable node. The coordinator is responsible for starting up the client and server programs (using SSH), periodically checking for liveness, and restarting the programs as required after individual nodes fail and recover.

5.3 Discussion

Most relevant previous work on construction of overlay networks has occurred in the context of supporting data discovery and sharing in a large-scale, heterogeneous network environment, and can be grouped into two main categories: *structured* and *unstructured* overlays.

Structured P2P overlays, like Tapestry [89], CAN [90], Chord [91], Pastry [92] and Coral [40], were designed principally to support data discovery and cooperative data storage using distributed hash tables (DHTs) whereby a data item is identified by a key and nodes are organized into a structured graph topology that maps each key to a responsible node where the data or a pointer to the data is stored. Unstructured P2P overlays, like Gnutella [93], Freenet [94] and Kazaa [95], organize nodes into a random graph topology and use floods or random walks for data discovery and other

queries. To address the potential problem of excessively long random walks or poor use of network bandwidth, researchers have proposed exploiting the network proximity among the participating nodes to build locality-aware overlays, where nodes that are relatively close to each other in the underlying network are clustered/grouped together to ensure that communication between two nodes in a group does not travel outside of this group. The focus of these systems is on supporting an all-to-all flow pattern in the context of data sharing, and unlike our medium-scale focus, the emphasis in them is typically on supporting efficient routing in extremely large-scale systems. Thereby, these systems are not a good match to the requirements of data-centric network services. The latter requires that the participating nodes be organized with an orientation "bias" towards one (or a small number of) origin server(s) such that (1) service usage locality can be detected dynamically by inspecting the underlying traffic flows; and (2) such locality can yield clustering and reuse benefits by replicating a small portion of service data from the origin server(s) at a few locations.

In addition to the work described above, researchers have also examined construction of overlay networks to support multicast flow patterns [96, 97, 98, 99]. The multicast networks address a more related problem, that of delivering a content stream from a single source to multiple locations. Unlike the bandwidth-centric focus of these systems, our work targets more general applications. Additionally, our reason for merging routes in the network has less to do with the elimination of redundant communication, and more to do with the discovering and leveraging service usage locality.

In the context of building overlays with some information about network proximity, [100, 101] has looked at topology-aware clustering of web clients using border

gateway protocol routing information. At the application level, works on topology-aware unstructured overlays have included a landmark clustering scheme [102, 103], which relies upon the existence of a small number of carefully selected landmark nodes that serve as location beacons for the other (usually larger number of) nodes that participate in the overlay. Given the smaller scale of our networks, we have relied upon direct measurements of the latency between the participating nodes and the origin server(s) and likely parent candidates. Recent work on incorporating network locality considerations into structured overlays (e.g., Coral [40]) has also pursued a similar direct measurement approach. Unlike these systems, most of which use network latency as an indicator of proximity, recent work on topology-aware multicast networks [98, 99] has looked into the mechanisms for estimating and optimizing use of network bandwidth. The latter is harder to measure directly, and reasoning about its shared use requires a better model of network utilization.

Finally, a number of recent systems such as IDMaps [104], GNP [105], WNMS [106] and Vivaldi [107] have been proposed to map nodes on the Internet onto locations in a cartesian coordinate system. These systems provide a global distance estimation service at the infrastructure level, and if available and accurate enough, can substitute for some of the measurements our algorithms make currently. Given that there are many applications where accurate geographical location information (as opposed to merely proximity indicators) yields a substantially better model of service usage, the wider availability of such systems will end up further improving the performance of our oriented overlays.

5.4 Summary

In this chapter, we have presented the design and implementation of a *zone-based* scheme for constructing *oriented overlays* to facilitate clustering and inspecting client requests at various points in a wide-area network. We defer the discussion of the evaluation of our approach till Chapter 8.

Chapter 6

Load-balancing and Service Replication

Although the DataSlicer architecture is designed to cope with a variety of service related QoS requirements, we focus on the metric of client perceivable response time in this dissertation. To improve this metric, DataSlicer exploits a combination of request redirection and data replication techniques. The first part of this chapter describes a load-balancing technique, which allows the routers to redirect requests along multiple paths leading to the service replica(s) to obtain better performance; the second part describes the service replica placement problem (SRPP), its complexity, and presents two distributed algorithms for solving the SRPP in tree- and DAG-structured networks respectively.

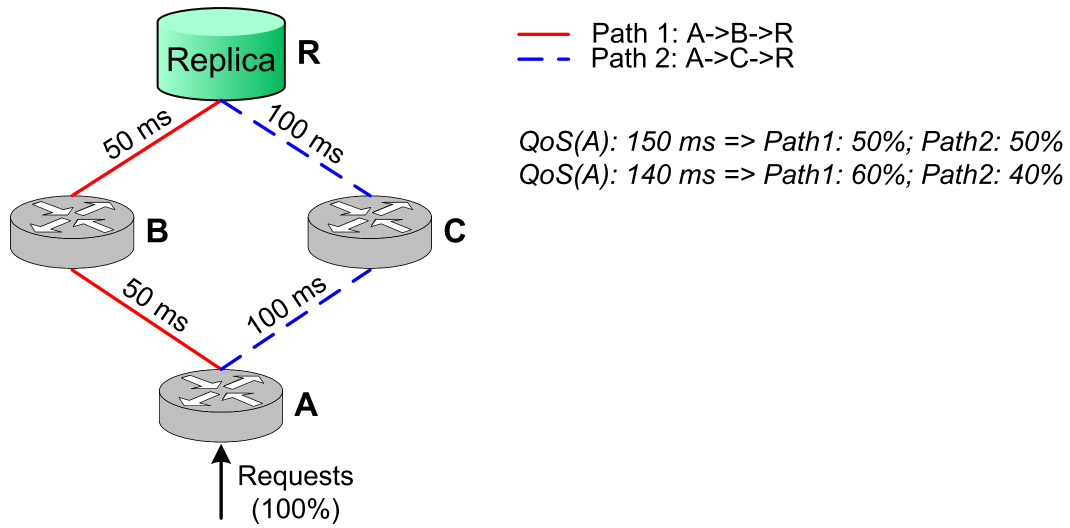
6.1 Load-Balancing

In the previous chapter, we described an oriented overlays construction scheme where a router could potentially have multiple paths lead to the service replica(s). In practice, these paths might have different metrics in terms of network latency, network bandwidth and node CPU utilization, and therefore result in different service response times observed by requests that traverse them. Consequently, one way to ensure that client requests meet desired QoS requirements is by shifting requests from paths that are seeing poor performance onto those paths that are currently experiencing fewer delays.[1]

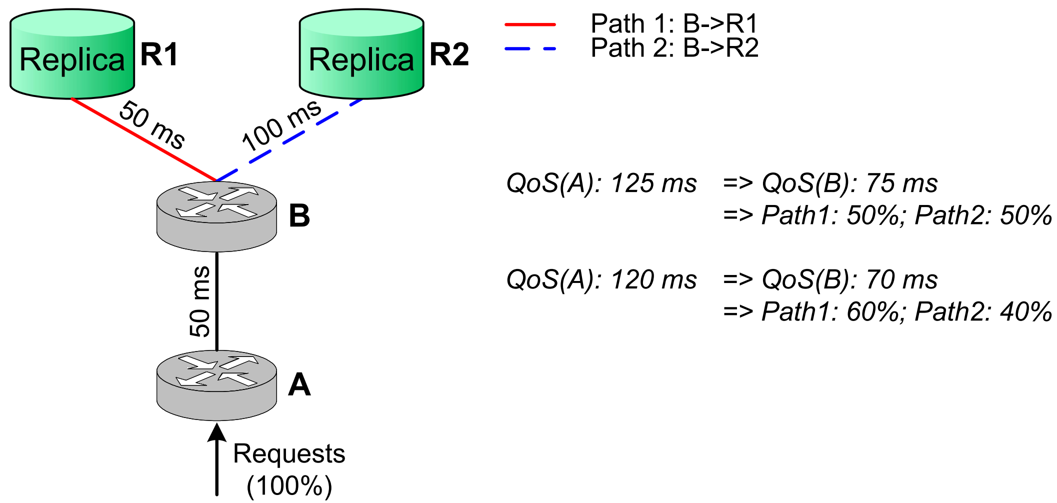
Figure 6.1 shows two scenarios to illustrate how load-balancing can improve performance, and highlight some of the challenges it needs to address. In the first scenario (Figure 6.1(a)), there is a single service replica (R), two intermediate routers (B and C), and one entry router (A) to whom clients send requests. Path $A \rightarrow B \rightarrow R$ has a response time of 100 ms, and the other path, $A \rightarrow C \rightarrow R$, has a response time of 200 ms. To satisfy a client QoS requirement of 150 ms at A, A can distribute 50% of its incoming requests to each out-going link, link (A,B) and (A,C). However, if the QoS requirement at A is 140 ms, A needs to redirect at least 60% of the traffic to link (A,B) and redirect the rest to link (A,C).

In the second scenario (Figure 6.1(b)), there are two service replicas ($R1$ and $R2$), one intermediate router(B), and one entry router (A). A has only one out-going link

[1] Throughout this document, the QoS expectation is defined in terms of the average response time observed by a group of requests, e.g. the requests that hit in the same Cell.



(a) Single-source case



(b) Multiple-sources case

Figure 6.1: Load-Balancing: two illustrative scenarios.

(A,B) and has to relay all incoming requests to this link. On the other hand, B has a link to $R1$ whose response time is 50 ms, and the other link to $R2$ whose response time is 100 ms. B can arbitrarily distribute the incoming requests (from A) to its out-going links to achieve a performance in the range between 50 ms to 100 ms. Therefore, although A can not really redistribute the incoming requests to multiple paths by itself, it can propagate its QoS requirements to B such that B can perform load-balancing for A to satisfy client QoS requirements at A.

These scenarios highlight the following challenges:

- How to collect information about path performance?
- How to deal with potential "thrashing" situations, where one needs to continually re-balance load between two or more paths?
- How to factor in competition for shared resources such as node CPU utilization and network bandwidth, which affect the overall performance seen by affected requests?

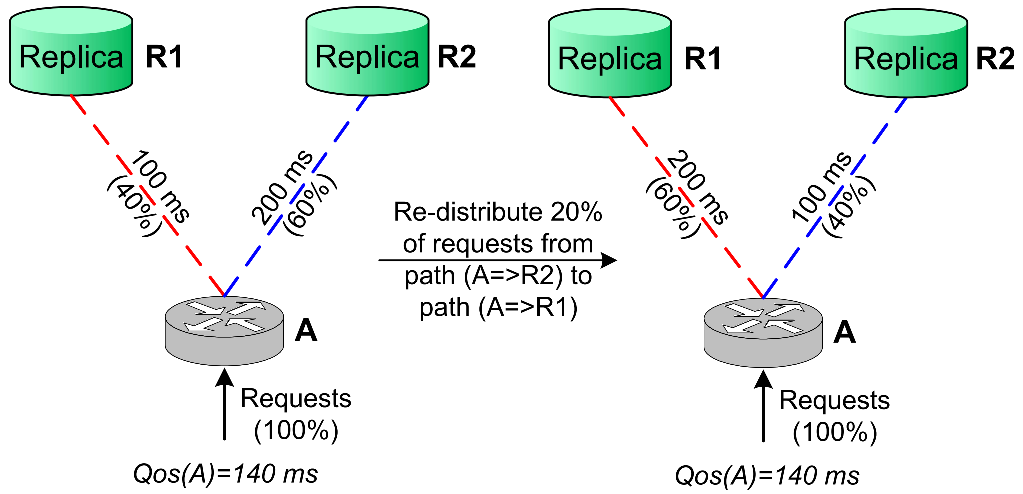
6.1.1 Problem Formulation

Collecting path information. The above scenarios implicitly assumed that the load-balancing scheme has knowledge of the performance metrics seen along multiple paths leading to the service replica(s). In practice, however, this requirement is usually very hard to satisfy because paths could consist of many links and the metrics of each link could be very dynamic in a real-world network, especially taking the effect of network bandwidth constraints and node CPU capacity into consideration. Ideally, one would prefer a scheme where each router can make load-balancing decisions

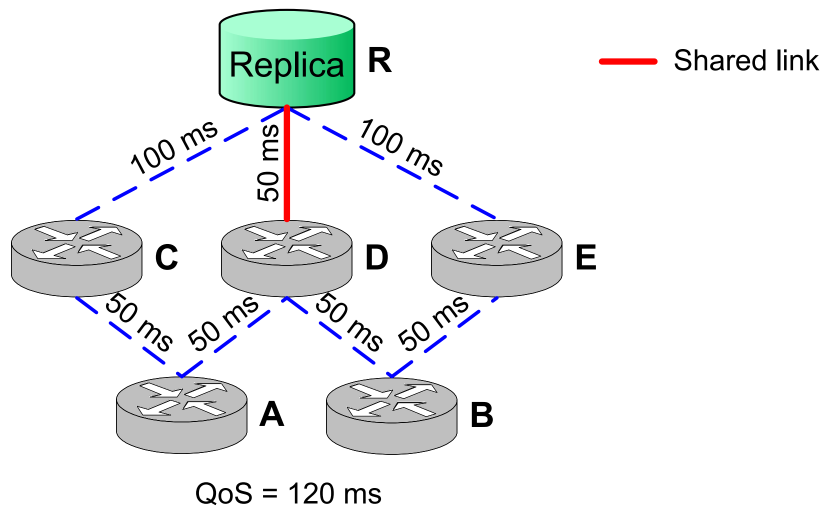
based only on its local view of the path information.

Thrashing Situations. When one considers the impact of network bandwidth constraints and node CPU capacity, the performance of a path changes as one varies the amount of traffic assigned to that path. This could lead to a "thrashing" situation, where traffic repeatedly ping-pongs between a path that sees good performance and one that sees poor performance. Figure 6.2(a) shows a concrete example of such a situation: the QoS requirement at router A is 140 ms, and the current load distribution at A has 40% of the traffic assigned to path $A \rightarrow R1$ and 60% to the path $A \rightarrow R2$, resulting in an average performance of 160 ms. Since path $A \rightarrow R1$ has a better performance (100 ms), A decides to shift 20% of traffic from path $A \rightarrow R2$ to path $A \rightarrow R1$ in order to satisfy the QoS requirement. However, the traffic shifting results in a degraded performance for path $A \rightarrow R1$ (response time increases from 100 ms to 200 ms), and an improved performance for path $A \rightarrow R2$ (response time drops from 200 ms down to 100 ms). Therefore, router A needs to do further load-balancing by shifting 20% of traffic from path $A \rightarrow R1$ back to path $A \rightarrow R2$, leading to the thrashing situation. The challenge for an effective load-balancing scheme is how to prevent or resolve the occurrence of the thrashing situation.

Resource competition. For efficiency, one would like a distributed load-balancing scheme where each router is making autonomous decisions based on its local information. However, in a real-world network, there exist situations where resources (e.g. nodes or links) are shared by multiple paths originating from different routers. Hence,



(a) Trashing situation



(b) Shared-link situation

Figure 6.2: Load-Balancing: challenges.

the autonomous decisions on traffic re-distribution made at individual routers could conflict with each other and could lead to competition in how the shared resources are utilized. Figure 6.2(b) shows an example where link (D,R) is shared by path $A \rightarrow D \rightarrow R$ and path $B \rightarrow D \rightarrow R$. Since both A and B see a poor performance on the other available path ($A \rightarrow C \rightarrow R$ for router A and $B \rightarrow E \rightarrow R$ for router B), they could shift traffic to the path that has better performance ($A \rightarrow D \rightarrow R$ for router A and $B \rightarrow D \rightarrow R$ for router B). This results in competition for utilizing the shared link (D,R), and hence could lead to a performance downgrade for link (D,R). The challenge here is how to achieve the global stability of traffic distribution even though each router is independently making its own decisions.

6.1.2 Approaches

The first issue of collecting path performance information is addressed by aggregating the affected factors (round-trip latency, network bandwidth and node CPU capacity, etc.) into one measureable metric: the response time observed by the routers. The routers approximate a centralized scheme for tracking performance of different paths by working as follows. Each router maintains information about its outgoing links on a per-cell basis; instead of monitoring individual metrics such as network latency, bandwidth, and node CPU utilization, our approach utilizes the request response times measured at the routers for past requests traversing a link to estimate the performance that would be seen by future requests. More specifically, in our *Cell* structure, besides maintaining the average response time observed by the group of requests, each individual cell at leaf level also maintains the performance information about each of the outgoing links, including the average response time observed for requests relayed

to a particular link, and the average traversal time for a request/response transmitted through that link.

Figure 6.3 shows a concrete example of the measurement of response time at our routers: consider a request relayed along the path $R1 \rightarrow R2 \rightarrow R3 \rightarrow replica$, with the response sent back along the same path but in the opposite direction. $R1$ kicks off a timer (at time $t1$) for the request when it arrives at $R1$ and relays the request to $R2$ after doing necessary inspection as described in Chapter 4. Similarly, $R2$ and $R3$ kick off a timer (at time $t2$ for $R2$ and $t3$ for $R3$) when they receive the request. When the response returns from the service replica, $R3$ stops its timer (at time $t4$) and computes the observed response time ($ST_3 = t4 - t3$). $R3$ then piggybacks this observed response time ST_3 in the wrapped response and sends it back to $R2$. When the wrapped response arrives at $R2$, $R2$ stops its timer and computes the observed response time ($ST_2 = t5 - t2$). $R2$ also computes the time for the request/response messages to traverse link $(R2, R3)$ as $LT_{(R2,R3)} = ST_2 - ST_3$. By aggregating similar information across the requests relayed to link $(R2, R3)$, $R2$ can estimate the average network traversal time and the average response time provided by link $(R2, R3)$. Similarly, $R1$ can compute the average traverse time and the average response time provided by link $(R1, R2)$. If requests have been relayed along the other path $R1 \rightarrow R4 \rightarrow R3 \rightarrow replica$, then $R1$ can also compute the similar performance metrics for link $(R1, R4)$, and $R4$ can compute the performance metrics for link $(R4, R3)$.

Given the above link-level information, the router continually manages the fraction of its requests that are relayed along a particular link, using the measured performance seen by previous requests as a feedback mechanism. A link is *satisfied* if the response times seen for that link are below the required threshold, otherwise, it is *unsatisfied*.

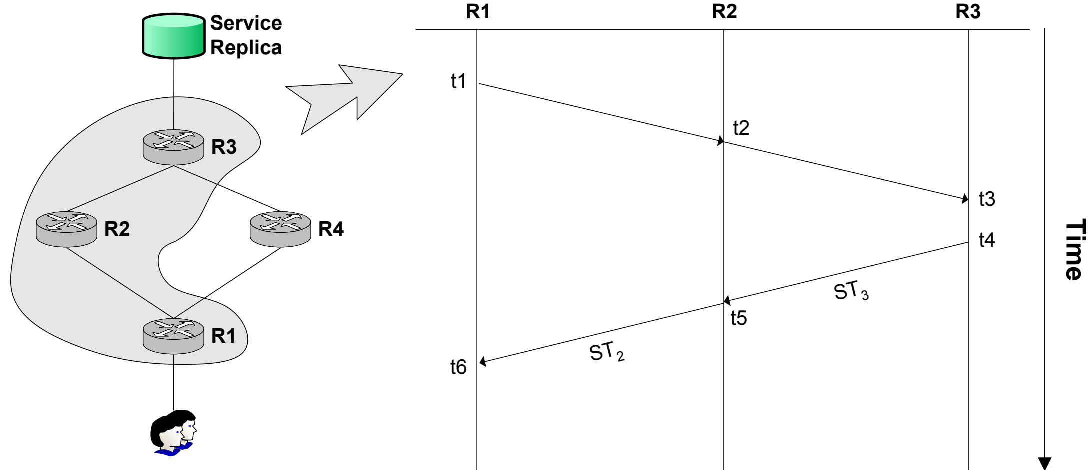


Figure 6.3: Response time measurement.

Using the same terminology, a router is *satisfied* if the combination of all of its links provides an average response time below the threshold and is *unsatisfied* otherwise. Similar to the window adjustment technique applied in TCP congestion control, an unsatisfied router tries to re-balance among its links in a "gradual" fashion, i.e., it initially moves a small fraction of the requests from an unsatisfied link to a satisfied link to avoid overwhelming the latter, and gradually ramps up this amount as it finds the other link continuing to behave properly. The only exception to this gradual shifting is when a new replica is created, at which time the associated routers eagerly redirect a larger fraction of their traffic to the new replica. The load balancing procedure terminates if either the router regains satisfaction or no satisfied links exist. In the latter case, a service replication action has to be taken.

To address the thrashing situations described earlier, where traffic repeatedly ping-pongs between an unsatisfied link and a satisfied link, we exploit a prediction-based

technique. Specifically, we predict the change in response time one would likely see *assuming* the redistribution, and permit the redistribution to take place only if this response time is below the threshold. The prediction assumes a linear relationship between the traffic change ΔF and the change of the response time ΔT, i.e., $\lambda = \Delta T / \Delta F$, with per-link λ values computed as a running average based on past measurements of request response times seen by traffic redistribution involving that link.

Note that the load balancing strategy does not directly prevent two routers from directing traffic onto the same (lightly loaded) shared path. The premise, substantiated by our experiments, is that such sharing will eventually reflect in the performance seen by the corresponding outgoing links at each router.

6.1.3 Discussion

Much current research has investigated use of request distribution techniques to improve web performance. These approaches can be categorized into two main trends: cluster-based network servers with centralized front-ends and loosely-coupled distributed servers employing DNS-based redirection or some other similar scheme.

Research in the former trend [108, 109, 110] usually exploits some variation of a weighted round-robin strategy on the front-end nodes to distribute the incoming requests to the servers in the back-end cluster. As example, the Dispatch product by Resonate Inc. [111] supports limited content-based request distribution, but restricts itself from supporting content-based dynamic distribution policies. In [112], the researchers address traditional shortcoming of such approaches by proposing locality-aware request distribution to achieve high locality in the backends' main memory cache as well as good balance. Such locality-aware techniques are beginning to get

incorporated into commercial products.

The second trend employs various techniques including DNS round-robin or HTTP client re-direction to distribute requests to a collection of distributed server surrogates, which cache the content from the original servers. Many existing CDN systems fall into this category. Among them, the commercial product by Akamai Inc. [48] is a good representative system. The Akamai CDN is employed mainly by content providers, and redirects client requests to surrogate servers using a DNS-based load balancing system, which continuously monitors the state of hosted services, and the server surrogates and networks. Each of the server surrogates frequently reports its load to a monitoring application, which aggregates and publishes load reports to the local DNS server such that it can then determine which IP addresses to return when resolving DNS names. Unlike the Akamai CDN, CoDeeN [113] is an academic testbed CDN which engages clients instead of content providers: clients need to specify a CoDeeN proxy in their browser settings when requesting the content. Each of the proxy servers in CoDeeN continuously monitors its local as well as its peers' information about the CoDeeN instance's state and the host environment and periodically reports such information to a central controller. The proxy servers first try to serve the incoming requests from their caches. In case the requests are not cached, the proxy server uses a redirector which considers request locality, system load, reliability, and proximity when selecting another CoDeeN node to forward the requests to.

Our load-balancing technique falls into the second trend above and is similar to the CoDeeN approach where the participating nodes act as forward/reverse proxy servers in request/response relaying. The differences between our load-balancing technique and the one of CoDeeN are: (1) our technique approximates the global information

about the replicas via local metric measurements; and (2) the goal of request distribution in our architecture is to maintain QoS requirements while in CoDeeN, there are no explicit QoS guarantees and the proxy servers simply select the "best" peer for request forwarding according to the criteria described above.

6.2 Service Replication

In situations where load balancing is insufficient to meet client QoS expectations, DataSlicer generates one or more service replication requests. The underlying problem here is one of *minimum cost replica placement*, i.e., deciding *which regions* of the data space to replicate at *which locations* (given a set of replica sites) to minimize overall replication cost, while satisfying a predefined QoS expectation on average client response times. The first problem, determining which regions needed to be replicated, has been addressed in Chapter 4, where we used the *Cell* structure to detect the regions that represent locality of client access patterns. Here, we focus on discussing the second problem: given the locality information and client QoS expectations, how to determine a service replica placement strategy with minimum cost.

We first describe the formulation of the above service replica placement problem, and then present algorithms to solve this problem on both tree- and DAG-topology networks.

6.2.1 Problem Formulation

In Chapter 5, we described how to organize the routers into an oriented overlay network to cluster client requests. The formed overlay has a directed acyclic graph

(DAG) topology. In addition to this router network, the DataSlicer architecture assumes the existence of a replica network maintained by the service provider. A router associates itself to a nearby service replica node (e.g. within 10ms in terms of network latency), and views the service replica as its local data repository where it can request a service replications. A router that does not associate itself with any service replicas forwards the service replication requests to its parent routers and has them initiate service replication on its behalf.

To simplify the formulation of the service replica placement problem, we combine these two networks together, and assume that (1) a subset of routers in the routing network can also serve as service replica nodes for service replica creation, called *replica router*s (requests relayed to a replica router can be satisfied directly if the corresponding data region has been replicated at this router); and (2) entry routers and intermediate routers are separate such that no intermediate routers can receive requests directly from clients. Additionally, we assume that (1) the network is static such that round-trip latencies of the links are fixed; and (2) the network bandwidth and node CPU capacity are unlimited such that the request response time is dominant by the network latency metric. These additional assumptions allow each entry router to redirect all of the client requests that hit in a service region to the *closest* replica router that contains a replica of this region to achieve the best performance.

Given our problem setting which involves creating replicas to host significant amounts of data in a wide-area network, we assume a fairly general cost function where the cost of a replication is monotonically determined by the volume of involved data, the distance of the service replica from the origin server, and the cumulative cost of consistency maintenance. The replication cost function can be expressed as the

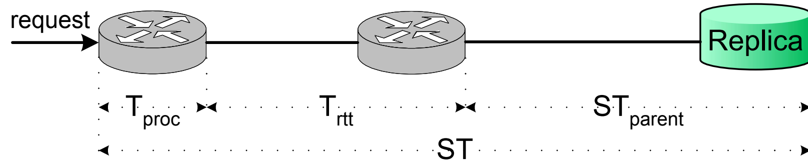


Figure 6.4: Computation of the response time for a request at a router.

following equation:

$$\text{COST} = c_1 \times \text{VOLUME} + c_2 \times \text{DISTANCE} + c_3 \times \text{UPDATE} \qquad (6.1)$$

where c_1, c_2, c_3 are some constants, VOLUME is determined by the amount of data that needs to be replicated, DISTANCE is determined by the distance between a replica and the origin server and is approximated by the round-trip latency of the path leading from the replica to the origin server, UPDATE is determined by the frequency/amount of data updates seen by the replica due to data consistency issues. For read-mostly workload, UPDATE typically represent a very small fraction of the overall data traffic and hence can be ignored.

The response time (ST) of a router for a single request can be broken down into three parts: request processing time (T_{proc}) at the router, round-trip network delay between the router and its parent (T_{rtt}), and the response time of the parent ($\text{ST}_{\text{parent}}$), as shown in Figure 6.4 and formulated as below:

$$\text{ST} = T_{\text{proc}} + T_{\text{rtt}} + \text{ST}_{\text{parent}}$$

Assume that the processing time (T_{proc}) at the router is negligible ($T_{\text{proc}} \ll T_{\text{rtt}} + \text{ST}_{\text{parent}}$), then:

$$\text{ST} \simeq T_{\text{rtt}} + \text{ST}_{\text{parent}} \tag{6.2}$$

The average response time of a router is computed by taking the average of the response times for all requests processed by this router. Here, we distinguish two cases, corresponding to whether or not the router is a replica router. If the router is a plain router, assuming that the number of requests is N, and the response time for request i at the router is ST_i:

$$\text{AST} = \sum_{i=1}^{N} \text{ST}_i / N \tag{6.3}$$

If the router is a replica router and contains a service data region in its local repository, for those requests that hit in this region, T_{rtt} and ST_{parent} become 0. Assuming the probability of a request hitting in the region is P_{hit}, we have:

$$\begin{aligned}
\text{ST}' &= P_{\text{hit}} \times T_{\text{proc}} + (1 - P_{\text{hit}}) \times (T_{\text{rtt}} + \text{ST}_{\text{parent}}) \\
&\simeq (1 - P_{\text{hit}}) \times (T_{\text{rtt}} + \text{ST}_{\text{parent}}) \qquad \text{/* } T_{\text{proc}} \text{ is negligible */} \\
&= (1 - P_{\text{hit}}) \times \text{ST} \qquad (6.4)
\end{aligned}$$

Thus, we can define the average response time for a replica router which contains a service replica:

$$\text{AST}' = (1 - P_{\text{hit}}) \times \text{AST} \tag{6.5}$$

Using the terminology above, the service replica placement problem can be stated as:

Service Replica Placement Problem (SRPP)

Given a DAG-based routing network which consists of a subset of replica routers ($\{\mathrm{RR}_i\}$) and a subset of entry routers ($\{\mathrm{ER}_j\}$), a database that has been partitioned into multiple disjoint regions ($\{\mathrm{P}_k\}$), a distribution of client accesses against the database at the entry routers, and a QoS threshold (T) on the average response time, find a min cost replica placement strategy such that the overall infrastructure satisfies the QoS requirement, i.e., the average response time of each entry router falls below the QoS threshold: $\mathrm{AST}_r \leq \mathrm{T}, r \in \{\mathrm{ER}_j\}$.

The SRPP is NP-hard.

We simplify the SRPP by considering a special DAG-topology network: a chain network (Figure 6.5), where each intermediate router has only one child, and we also assume all of the routers are replica routers. We show that the SRPP is NP-hard even for this simplified network.

Suppose that the chain consists of one origin service server S and n routers where RR_1 is the entry router and RR_n is the exit router. The database has been partitioned into K disjoint regions $\{P_k\}$. We define the distance between a router RR_i and the origin server as the cumulative length of the path in the chain which starts at RR_i and ends at S. Consequently, we denote the cost for replicating data region P_j on router RR_i as $c_{i,j}$, where $c_{i,j}$ is determined by Equation 6.1. Without losing generality, we can view the origin server S as a special router RR_{n+1} whose cost of replication is free, i.e., $c_{n+1,j}$ equals to 0 for all $j \in [1,K]$. Consequently, we denote the round-trip latency between RR_1 and RR_i as $l_{1,i}$, where $i \in [1,n+1]$. Obviously, $l_{1,1}$ equals 0 and $l_{1,n+1}$ is the cumulative round-trip latencies of the chain. Given a client request distribution

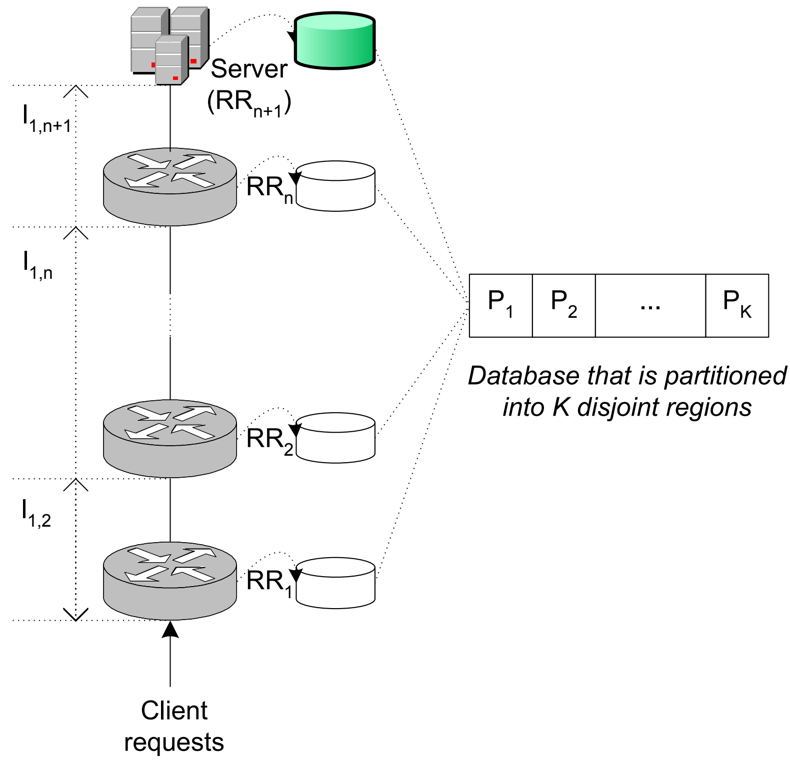


Figure 6.5: Chain-based hierarchical routing network.

over the K database regions, $r_1, r_2, \ldots, r_K$, a QoS threshold T of the average response time, and using the following terms: reqs $= \sum_{j=1}^{K} r_j$ and $T' = T \times$ reqs, the simplified SRPP can be formulated as:

$$\text{minimize: } \sum_{i=1}^{n+1} \sum_{j=1}^{K} c_{i,j} x_{i,j}$$

$$\text{subject to: } \sum_{j=1}^{K} r_j \sum_{i=1}^{n+1} l_{1,i} x_{i,j} \leq T'$$

$$\sum_{i=1}^{n+1} x_{i,j} = 1, \ \ j = 1, \ldots, K$$

$$\text{where: } x_{i,j} = \begin{cases} 1, & \text{if database region P}_j \text{ is replicated at router RR}_i \\ 0, & \text{otherwise} \end{cases}$$

Let the average response time of RR_1 before replica placement be AST_{RR_1}, denote $C = \text{AST}_{\text{RR}_1} \times reqs - T'$ as the total amount of time that needs to be reduced to satisfy the QoS requirement. Let $l'_{1,i} = l_{1,n+1} - l_{1,i}$ be the amount of time that could be reduced if a replica is placed at router RR_i. The simplified SRPP is reduced to finding a min-cost replica placement in the chain such that the amount of saved time at entry router RR_1 is at least C:

$$\text{minimize: } \sum_{i=1}^{n+1} \sum_{j=1}^{K} c_{i,j} x_{i,j}$$

$$\text{subject to: } \sum_{j=1}^{K} r_j \sum_{i=1}^{n+1} l'_{1,i} x_{i,j} \geq C$$

$$\sum_{i=1}^{n+1} x_{i,j} = 1, \; j \in [1, K]$$

This is the minimized version of Multi Choice Knapsack Problem (MCKP) [78] , which is known to be NP-hard.

To understand whether there exists a different formulation of the service replica placement problem amenable to an efficient solution, we enforce a restriction on the SRPP resulting in the *Restricted Service Replica Placement Problem (RSRPP)*. RSRPP differs from SRPP by imposing a per-region QoS-assured requirement, which may increase the cost over that required by SRPP. However, since the database is partitioned into multiple disjoint regions, RSRPP can be broken into a set of independent subproblems, each of which works on only one region.

Restricted Service Replica Placement Problem (RSRPP) Statement

Given a DAG-based routing network which consists of a subset of replica routers ($\{\mathrm{RR}_i\}$) and a subset of entry routers ($\{\mathrm{ER}_j\}$), a database that has been partitioned into multiple disjoint regions ($\{\mathrm{P}_k\}$), a distribution of client accesses against the database at the entry routers, and a QoS threshold (T) on the average response time, find a min cost replica placement strategy such that for any entry router, ***the average response time of requests hitting in each database region, considered separately****, remains within the QoS threshold.*

Claim: *A min-cost replica placement strategy for RSRPP is equal to the union of min-cost replica placement strategies for RSRPP's subproblems, where each subproblem works on an individual database region.*

Proof:

First, by assumption, the database regions are disjoint and therefore independent from one another. So a replication created for a region will not affect the performance of others.

Secondly, the union of the min-cost replica placement strategies for subproblems provides a solution for RSRPP because it ensures that the average response time of the entry router for each database region remains within the QoS threshold.

Finally, the union of the min-cost replica placement strategies for subproblems is a min-cost replica placement strategy for the RSRPP. Assume the union of min-cost replica placement for subproblems is $S = \bigcup s_i$, where

each s_i is a min-cost replica placement strategy for an individual subproblem. Suppose there exists another replica placement strategy *S'* which has less cost. Obviously, *S'* can be divided into a set of disjoint replica placements $\{s'_i\}$, each of which contains only the replica placement for one database region. In other words, each s'_i is a solution for an individual subproblem. Then, there exists at least one s'_k, $\text{Cost}(s'_k) < \text{Cost}(s_k)$. This contradicts the definition of s_k. ■

In the rest of this section, we focus on discussing the algorithms for the subproblem of RSRPP. Because the subproblems are independent from each other, the algorithms can be run concurrently for each subproblem.

6.2.2 Algorithms

We first study the RSRPP on a tree-topology network and prove it to be a polynomial problem by providing both centralized and distributed algorithms. However, the RSRPP remains NP-hard on a DAG, for which we provide a heuristic algorithm.

Algorithms for service replication on a tree-topology network

To simplify our description of the algorithms, we assume, without loss of generality, that all of the entry routers are unsatisfied, i.e., for a specific database region, the average response time of each entry router exceeds the QoS threshold. This assumption does not affect replica placement because of the observation that already satisfied routers do not require any replica creation in the router tree in order to reduce their average response times. Thus, we can reduce a general router tree to a tree where the

leaf nodes corresponding to unsatisfied routers by:

- applying a depth-first search on the router tree
- marking a node as “unsatisfied” if: (1) the node is an entry router and its average response time exceeds the QoS threshold; or (2) any of its children are marked as “unsatisfied”
- removing all of the unmarked nodes from the router tree

We also assume that all of the routers are replica routers, i.e., they are able to hold replicated regions. Notice that a router tree can be reduced to a tree where all of the intermediate routers are replica routers by the following transformation (without loss of generality, assume that the root router is a replica router):

- given a non-replica intermediate router **r**, let **R** be **r**’s parent
- insert **r**’s children into **R**’s child list and update their link information (especially the response time) appropriately
- remove **r** from the router tree

After the transformation, we can safely remove the entry routers if (1) the entry router is not a replica router, and (2) the round-trip latency of the link between the entry router and its parent exceeds the QoS threshold. Obviously, no replica placement solution exists which can satisfy the latter kind of entry routers.

Given such a reduced router tree, the next question we need to answer is how to find a min-cost replica placement strategy for a subproblem in polynomial time.

(I). Centralized algorithm on tree topology

Figure 6.6 shows our centralized algorithm which solves the subproblem on a tree-topology network in polynomial time.

Claim: *The replica placement strategy generated by the centralized algorithm for a tree-topology network is optimal.*

Proof:

The algorithm is a greedy algorithm.

Assume the entry routers are $r_1, \ldots, r_m$, listed in descending rank order based on how unsatisfied they are. Suppose the router chosen by the algorithm to create a replica to satisfy r_1 is **R**, we claim that any min-cost replica placement strategy should include **R** as one of the nodes on which a replica is created. The proof follows immediately from the following observations:

1. By the definition of replication cost (Equation 6.1), since the size of a database region is fixed and we can also assume a fixed rate of data updating, the only variant that affects the cost is the distance.
2. A replica created on a node residing in **R**'s subtree has a cost at least as much as the one placed at **R**.
3. A replica created on a node outside of **R**'s subtree can not satisfy r_1.

By induction, the algorithm generates an optimal solution for the subproblem of RSRPP. ■

Inputs:*(maintained per-router for a particular region)*

T: a router tree whose leaf routers (entry routers) are unsatisfied

Q: client QoS threshold (approximated by response time at entry routers)

AST_r: average response time observed by router **r**

$t_{r,R}$: the round trip time of path between router **r** and its ancestor **R**

Centralized Algorithm:

1. Sort the entry routers **r** by their average response times in the descendent order.
2. Starting with the most unsatisfied entry router **r**, traverse along the path from **r** to the root to find the highest intermediate router **R** in the tree where the replica can be created in order to satisfy $t_{r,R} \leq Q$.
3. Remove all the routers in the subtree rooted at **R** (except **R**).
4. Repeat steps (2) and (3) until there exist no more unsatisfied entry routers.

Figure 6.6: Centralized algorithm for service replication on a tree-topology network.

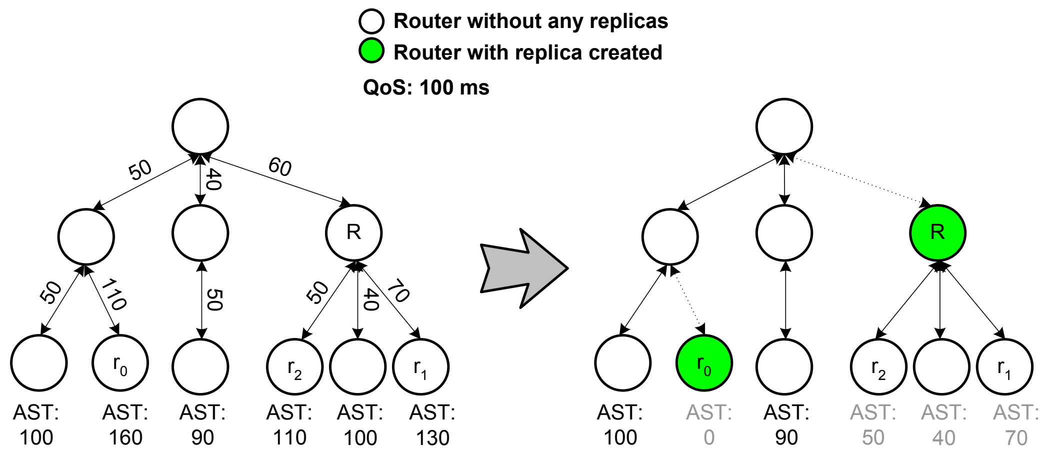


Figure 6.7: An illustrative example of applying the centralized algorithm for service replication on a tree-topology network

The algorithm runs in time $O(D \times E)$, where D is the depth of the router tree and E is the number of unsatisfied entry routers.

Figure 6.7 shows a simple example of applying the centralized algorithm to create service replicas in a tree-topology network. In this example, the client QoS requirement is 100 ms, and the tree configuration and the round-trip latencies of the links are as shown in the figure. The algorithm identifies that the unsatisfied entry routers are r_0, r_1 and r_2, and sorts these entry routers according to their perceived response times. The algorithm then first looks for the proper replica site to satisfy r_0, which ends up being r_0 itself. Since r_1 and r_2 can not get benefits from the created replica at r_0, the algorithm then finds the replica site for r_1, which ends up being the intermediate router R. Because the replica created at R can satisfy both r_1 and r_2, the algorithm terminates.

(II). Distributed algorithm on tree topology

Figure 6.8 shows our distributed algorithm to solve the subproblem, which for simplicity is described in terms of the actions taken by routers in a particular round of the protocol. The basic idea is to use the round-trip time estimates available at each router to determine if creating an upstream replica for the data space region can in fact satisfy the client QoS expectations. The first router in the path from clients to the origin service that determines this in the negative is the one that ends up requesting the replica creation.

Claim: *The replica placement strategy generated by the distributed algorithm for a tree-topology network is optimal.*

Proof:

The claim is proven if any optimal replica placement strategy should include all of the nodes that issue replication requests to the origin server in the algorithm. We prove it using induction.

Since our algorithm works in bottom-up order where the lower level routers send satisfaction messages to their parents, starting at the leaf level, we refer as a *round* a step of the algorithm where routers at a particular level of the tree send their satisfaction messages. Therefore, round 0 indicates that the algorithm works at the entry routers. Suppose the depth of the router tree is D, we have:

(1) At round 0, the algorithm checks the entry routers and issues replication request(s) if and only if the t_{lat} of any of the entry routers is greater than t_{dec}, its required reduction in response time. These entry routers that have requested replica creation should be present in the optimal replica placement strategy.

(2) Suppose at round k, $k < D$, the nodes that issued replication requests according to the algorithm are in the optimal replica placement strategy. At round *k+1*, if there is an intermediate router **R** which is chosen by the algorithm to create a replica, then there exists an unsatisfied entry router **r**, which has the following properties:

- the round-trip latency between **r** and **R** is not greater than the reduction in response time required by **r**
- the round-trip latency between **r** and **R'**s parent is greater than this requested reduction

Inputs:*(maintained per-router for a particular region)*
T: a router tree whose leaf routers (entry routers) are unsatisfied
Q: client QoS threshold (approximated by response time at entry routers)
AST_r: average response time observed by router **r**
$t_{r,R}$: the round trip time of path between router **r** and its parent **R**
C_r: the child routers of router **r**

Entry router (r):
set $t_{\text{dec}} = \text{AST}_r - Q$
if $t_{\text{dec}} < 0$
  send "Satisfied" message to **r**'s parent **R**
else if $t_{r,R} > t_{\text{dec}}$
  request replication
  send "Satisfied" message to **r**'s parent **R**
else
  send ["Unsatisfied", t_{dec}] message to **r**'s parent **R**

Intermediate router (r):
collect messages from children in C_r
if messages are all "Satisfied"
  send "Satisfied" message to **r**'s parent **R**
else
  set t_{dec} = minimum t_{dec} of all unsatisfied messages sent from **r**'s children
  follow steps taken by the Entry router

Root router (r):
collect messages from children in C_r
if there exist any "Unsatisfied" messages
  request replication

Figure 6.8: Distributed algorithm for service replication on a tree-topology network.

- no replica has been created on any node on the path from **r** to **R**
- any node residing in **R**'s subtree, which has been chosen in an earlier round to create a replica, is in the optimal replica placement strategy (follows immediately from the induction assumption)

To satisfy entry router **r**, the optimal replica placement strategy has to include **R** as one of the nodes holding the replica, which means that the nodes chosen to create the replica up to the tree level $D-k-1$ should be included in the optimal replica placement.

By induction, the algorithm generates an optimal strategy. ■

The algorithm will terminate after round D, where D is the depth of the router tree. The number of messages that need to be sent for communication between routers is at most $2N$ where N is the number of nodes in the router tree.

Algorithms for service replication in a DAG-topology network

Our network intermediary architecture addresses the client requests clustering problem by exploiting an oriented overlay construction technique. In the formed overlay, the routers are organized into a DAG-topology network. Ideally, we would like to solve the min-cost replica placement problem on such a DAG-based topology. However, the subproblem of RSRPP in a DAG-topology network remains NP-hard. To prove this claim, assume that there are n entry routers $\{r_i\}$ and m replica routers $\{R_j\}$ in the DAG. Let c_j denote the replication cost of creating a replica on replica router R_j; C denote the cost of creating replicas on all of the replica routers, $C = \sum_{j=1}^{m} c_j$; $t_{i,j}$ denote the latency of the shortest path from entry router r_i to replica router R_j (if no

path exists between r_i and R_j, $t_{i,j} = \infty$); and T denote the QoS threshold on the entry routers. Without loss of generality, let R_0 denote the origin server such that c_0 equals 0. The subproblem of RSRPP on DAG-topology can then be formulated as below:

$$\text{maximize: } C - \sum_{j=0}^{m} c_j x_j$$

$$\text{subject to: } \sum_{j=0}^{m} f_{i,j}\, t_{i,j}\, x_j \leq T,\ i = 1,\ldots,n$$
$$\sum_{j=0}^{m} f_{i,j} = 1,\ i = 1,\ldots,n$$
$$\sum_{j=0}^{m} f_{i,j}\, x_j = 1,\ i = 1,\ldots,n$$

$$\text{where: } x_j = \begin{cases} 1, & \text{if the database region is replicated at router } R_j \\ 0, & \text{otherwise} \end{cases}$$

$$f_{i,j} = \begin{cases} 1, & \text{if } r_i \text{ redirects its requests to } R_j \\ 0, & \text{otherwise} \end{cases}$$

It is easy to prove that this problem is at least as hard as the Knapsack problem, which is known to be NP-hard. Therefore, we use a heuristic approach to solve the problem.

(I). Heuristic algorithm on DAG topology

The heuristic algorithm on DAG topology is similar to the centralized algorithm on tree topology except that (1) the latter always selects the most unsatisfied entry router to find a replica site, but the former *randomly* selects one from the set of unsatisfied entry routers; (2) a router in the latter can only find a replica site on its unique path leading towards to the root, but a router in the former can *randomly* find a replica site

on any of its multiple paths leading towards to the root; and (3) once a replica site is created, the latter removes all of the entry routers in the subtree rooted at this replica, but the former removes only these entry routers that have at least one path leading to this replica and where the round-trip time of this path does not exceed the QoS threshold.

It is possible for this heuristic algorithm to implement a bias in its selection of unsatisfied entry routers based on the degree of discontent and similarly, drive the selection of the replica router based on the minimum or maximum of the round-trip latencies of the shortest paths leading to the replica routers. However, it is unclear whether such a biased version could outperform the randomized version. Obviously, the algorithm runs in time $O(E^2 \times M)$ where E is the number of entry routers and M is the number of replica routers.

(II). Distributed algorithm on a DAG topology network.

The heuristic algorithm requires that (1) each router maintains information about the round-trip latencies of the shortest paths between itself and its reachable replica routers; and (2) an entry router relays all of its incoming requests to only one replica router and uses only a particular path leading to that replica router. As discussed at the beginning of this chapter, the maintenance of the global view of the network is costly and hard to accomplish in practice where the network is very dynamic. Moreover, in a DAG-based network, a router is potentially able to connect to multiple replica routers, and it is more desirable to distribute the load among these paths leading to replica routers compared to using only one path to relay all of the requests. More importantly, our algorithm by assumption ignored the effects of the network bandwidth

and node CPU capacity metrics. In reality, these two metrics would prevent a router from relaying all of its requests to a single path due to utilization constraints.

In the rest of this section, we present a distributed algorithm to approximate the optimal solution for the subproblem of RSRPP on a realistic DAG-topology network, which also allows load distribution among the multiple paths leading to replica routers by exploiting the load balancing technique described earlier in the chapter.

As described before, we use the request response time to approximate the effect of the network metrics including latency, bandwidth and node CPU capacity. Figure 6.9 shows our algorithm for solving the subproblem, which permits distributed implementation and for simplicity is described in terms of the actions taken by routers. The algorithm combines information about the replica associated with the node with the per-outgoing link round-trip response time estimates available at each router to determine whether creating an upstream replica for the database region can in fact satisfy the client QoS requirements. The main observation used by the algorithm is that an unsatisfied router that has to initiate a replication action cannot get much benefit from replica creation at a replica site whose round-trip service time is larger than the QoS expectation associated with this router (which is not necessarily an entry router). To obtain information about round-trip times between a router and its closest replica, we augment the way in which a router piggybacks its observed service time in the wrapped response: a router sends the downstream router not only its observed service time, but also the service time observed by its closest ancestor that has an associated replica (if the router itself has an associated replica, both values are the same).

The algorithm only indirectly minimizes cost of replication (by pushing replica creations as close to the origin servers as possible), and because of its distributed na-

***Inputs**: (maintained per-router per-region)*

Q: client QoS threshold (approximated by response time at entry router)
L: set of outgoing links
f_l : fraction of requests relayed to link l
t_l : round-trip time of link l
$t_{l,R}$: round-trip time to the closest replica **R** observed by link l
AST_l : average response time observed by link l
AST: average response time observed by the router

Actions:

load_balance(*r*,*q*):
re-balance load on router **r** using threshold q

replication_request(*r*,*q*):
set $L' \subseteq L$, s.t. $t_{l,R} < q$ for all $l \in L'$
if $|L'| > 0$
randomly select $l \in L'$
send replication request $[q\text{-}t_l]$ to link l
else if **r** has associated replica
request the replica to be created at **r**

Entry router(*r*,*Q*):
if AST > Q
if $\exists\, l \in L$, s.t. $\text{AST}_l < Q$
load_balance(r,Q)
else
replication_request(r,Q)

Intermediate router(*r*,*q*):
if AST > q
if $\exists\, l \in L$, s.t. $\text{AST}_l < q$
load_balance(r,q)
else
replication_request(r,q)

Figure 6.9: Distributed algorithm for service replication on a DAG-topology network.

ture cannot claim to be optimal, in particular to avoid redundant replica creation. However, as our experiments show, in practice, the combination of the load balancing strategy and the replication algorithm appear to significantly reduce this possibility.

6.2.3 Discussion

Data replication in wide-area networks has been widely applied both in database replication systems and web content delivery networks to reduce client-perceivable latency, better utilize the fixed bandwidth of the upstream links, and alleviate the server load.

The replica placement problem has been well studied and shown to be an NP-complete problem for general network graph topologies. Systems have traditionally employed relatively simple heuristics such as demand-driven caching of frequently accessed (usually all) data at the network edge. More advanced approaches have also included some reasoning of data access patterns across multiple clients to determine where to place a replica. Example approaches in this category include the "best-client", "cascading replication" and "fast spread" mechanisms discussed in [114], which locate new replicas near clients that generate the most traffic, near other related replicas or along shared paths from clients to the origin service. In addition to such best-effort mechanisms, several researchers have also looked at formulations of the replica placement problem where the goal is to optimize some global metric, usually average client access costs. A representative formulation models the placement problem of placing M proxies on N nodes as a k-Median problem [115]: for tree topologies, the latter problem admits an optimal solution based on dynamic programming, albeit with high complexity ($O(N^3M^2)$) [116], but approximations need to

be employed for more general topologies [117, 58]. Researchers have also examined optimal strategies for the partial replication problem, when one needs to determine both the subset of replica objects and their placement [118, 59]. Most known results in this category have restricted themselves to hierarchical network topologies.

Our network intermediary architecture can react to unsatisfactory performance by employing any of a number of algorithms, including the ones mentioned above. However, our problem context of data-centric network services precludes most of these algorithms from being directly applicable. First, given the volumes of data that such services involve, it may not be sufficient to cache accessed data only at storage-constrained edge servers. Second, and perhaps more importantly, even in situations where only a subset of the service data is being replicated, the costs of replica creation and maintenance cannot be totally ignored. Thus, our placement problem is closer to the optimal placement formulations described above. The main differences stem from the specific nature of the problem we target, and the requirement for a distributed implementation.

6.3 Summary

In this chapter, we have described how our network intermediary architecture exploits load balancing and service replication techniques to improve the performance and scalability of data-centric network services in wide-area network environments. We have proved that the service replication problem (SRPP) is an NP-hard problem and studied the restricted version of SRPP on both tree- and DAG-topology networks. The evaluation of these algorithms is deferred until Chapter 8.

Chapter 7

System Robustness

In a wide-area network, one might experience many kinds of network outages including the failure of participating nodes, connection breaking for the links, and packet drops in transmission, etc. Additionally, because the network is very dynamic, one might also experience errors in measurement of network metrics such as network latency and bandwidth. DataSlicer needs to be able to recover from and adapt itself to these network outages and inaccurate network measurements to provide stable functionality. This chapter describes four techniques applied in our architecture to ensure system robustness: (1) liveness monitoring and repair; (2) link-level flow control; (3) non-blocking/asynchronous communication; and (4) smoothing out statistical fluctuations in network measurements. The development of these techniques was driven by our experiments and repeated refinements of the architecture, where DataSlicer was running on a real-world network for extended periods of time. Together these techniques proved to be sufficient to provide a high level of robustness.

7.1 Liveness Monitoring and Repair

In a wide-area network where nodes are geographically distributed across the world, the network inevitably suffers from various faults such as link failures, node crashes and packet losses. This requires the DataSlicer architecture to be able to continuously monitor the system, and upon any faults being detected, trigger corresponding recovery actions. In this section, we focus on discussing the monitoring and repair schemes for ensuring three different kinds of liveness: node liveness, link liveness and message liveness.

Node Liveness. There are many reasons that can cause a node to cease to function, such as the electrical outages, application or operating system crashes, or the exhaustion of available resources. It is important for DataSlicer to be able to detect such "dead" nodes and remove them from the constructed overlay network to provide good performance, e.g., to avoid messages being relayed to a "dead" node or for recycling the resources occupied by a "dead" node.

Many schemes have been proposed for node liveness monitoring. For any monitoring scheme, there is a tradeoff between providing instant detection of node liveness and reducing the communication cost. In our problem context, desirable properties include the ability to run in a distributed fashion on the participating nodes and require minimum communication cost. To achieve this goal, our node liveness monitoring scheme exploits two important mechanisms: "lease-based tracking" and "heart-beat probing". The "lease-based tracking" mechanism allows a node to keep track of the liveness of other nodes by setting a "lease" for each of these nodes and having them

revalidate the leases before expiration; the "heart-beat probing" mechanism allows a node to actively report its liveness to and inquire about the liveness of another node by sending a heart-beat message to the latter (which will revalidate the lease maintained by the latter) and expecting a response within a certain time period.

The node liveness monitoring scheme runs per overlay network constructed by our oriented overlay construction scheme, and is described in terms of the actions taken on the origin server and the participating nodes:

- The origin server maintains a global view of the liveness of the participating nodes. For each node, the server sets a *lease* and a *counter* (initialized to zero) when the node joins the overlay. The *lease* keeps track of how long the server believes that a node is alive and the *counter* keeps track of how many times a node has been reported as "dead" by other participating nodes. A participating node is considered as "dead" by the server if either it fails to renew its lease with the server before it expires, or the *counter* exceeds a pre-defined threshold (notice that whenever a node renews its lease, its corresponding *counter* will be cleared). For a "dead" node, the server removes it from the overlay and multicasts this information to other affected nodes.

- Each participating node maintains its local view of the liveness of its parents, candidate parents and children: (1) for each child node, the node sets a *lease* when it accepts the child and requires the child to renew the lease before it expires. If any child fails to renew its lease, the node removes the child from its children list and recycles any occupied resources as necessary; (2) for the selected parent nodes and all candidate parent nodes, the node periodically

chooses a subset of these nodes by random and sends heart-beats to the chosen nodes. If any probe fails, the node will reports the death of the probed node to the server, removes it from the parent or candidate parent list and recycles any occupied resources as necessary. The probe may trigger the parent-selection procedure described in Section 5.1 if any parent node is detected as being dead. Notice that the time epoch size of the period for heart-beat probing is usually far less than the one for lease renewal.

Our node liveness monitoring scheme is distributed by allowing each participating node to maintain its local view of the liveness of its nearby nodes, and uses the server to act as a rendezvous point which aggregates the local views of the participating nodes to maintain a global view of node liveness in the network. The randomized selection of candidate parents for heart-beat probing significantly reduces the amount of communication while preserving the ability to detect a "dead" node quickly with a high probability.

The node liveness monitoring scheme is implemented as a part of our overlay maintenance protocols.

Link Liveness. In our oriented overlay network, a router usually needs to maintain multiple links between itself and its parents or children. It is important for a router to be able to detect the broken or idle links so that messages can be relayed through the overlay network properly and the utilization of node resources can be optimized.

Broken links can be easily detected because the links are established on top of sockets using the TCP protocol: for each socket, a router listens on a port and aggressively retrieves any exceptional signals associated with that socket. When an

exception occurs, for an outgoing link, the router will take actions such as reporting an exception for the requests that were relayed to this link, and trying to re-establish the link if possible; for an incoming link, the router will take actions such as reporting an exception for the requests that arrived at this router using this link, recycling the resources assigned to this link (such as queue buffers as discussed in the next section), and closing the socket.

Idle links refer to the links which do no see any activity for a long time. Since a parent router usually has to accommodate multiple children which compete for the utilization of its resources, disconnecting idle links and recycling their occupied resources will improve the utilization of the parent node's resources. Similar to node liveness monitoring, a parent router sets a "lease" for each incoming link. Any activity (data transmission) on an incoming link renews the link "lease". An incoming link is considered as idle if its lease expires, in which case the parent router can disconnect that link and free its occupied resources.

Message Liveness. A client request has to be transmitted through our overlay network across one or more routers, each of which queues/buffers the messages and can potentially drop it to free up resources (as we describe in the next section, this is a rare event). The way we ensure end-to-end liveness of messages is to apply a time-out mechanism such that proper actions including exception reporting or message re-transmission can be taken in time.

The liveness of a request is maintained by a *time-to-live* (TTL) value set by the entry routers. This information is piggybacked in the wrapped request relayed through the overlay network. Each router keeps track of all of the active requests (i.e., re-

quests that are waiting for responses) and periodically checks the validity of their TTL values. The first router that detects an expiration of the TTL value for an active request can (1) report a *time-out* exception for this request, which will then be recursively transmitted back to the client; or (2) re-transmit the request (using a different outgoing link). The latter might be less desirable because it introduces extra storage overhead at the routers, and more importantly does not bring much benefit: with the TTL usually set to a rather large value, e.g. 180 seconds (which is also the default timeout value for a TCP connection), few clients will wait any additional time for the response.

7.2 Link-level Flow Control.

In DataSlicer, an intermediate router usually has multiple children, which compete against each other for the utilization of the intermediate router's resources such as network bandwidth, CPU and especially *buffers*. The intermediate router collects incoming requests into a queue and then processes them in order, which raises the possibility of the queue backing up such that the router has to tail-drop the subsequent requests. Such a strategy is not desirable because it causes message loss within the network. We instead use a better solution which is able to address the following two issues: (1) prevent a router from relaying requests to its parent until the parent has spare slots in its request queue; and (2) optimize the utilization of the request queue.

To realize the desirable strategy described above, we employ a *token-based flow control* algorithm (shown in Figure 7.1). The algorithm views the available slots in the request queue as *tokens* and tries to pre-allocate a certain number of tokens for

each incoming link[1] such that concurrent requests relayed from an incoming link will not exceed the slots allocated to it. Whenever a child router relays a request to an outgoing link, it *consumes* a token on that link. The consumed token will be *regenerated* when the response is returned to the child router. Similarly, whenever a parent router receives a request from an incoming link, a token associated with that link is consumed. The consumed token will be regenerated if the parent router sends the response back on that link. Both the parent router and the child router keep track of the information of token usage for the link between them. Such information includes the total number of tokens assigned to this link by the parent router and the number of the currently *consumed* tokens. A child router will suspend relaying requests on an outgoing link if its available tokens have run out. Similarly, a parent router will stop receiving requests from an incoming link if there are no available tokens for that link.

The above scheme addresses the first challenge described earlier. However, it does not optimize the utilization of the request queue of the intermediate router because each incoming link is assigned a fixed number of tokens (computed as S/M, where S is the size of the request queue and M is the maximum number of children this router can accept). Such a rigid assignment of tokens leads to inefficient usage of the request queue in case the actual number of child routers is lower than the threshold or when some of the incoming links are inactive.

To improve the efficiency of queue utilization, we allow a dynamic assignment of tokens for the links which can promote an active link with extra tokens (if available)

[1] We assume that for each registered service, there is a single router-router link. However, our algorithm is easily extendable to support multiple such links.

Inputs: (*maintained per-router*)

- S: size of request queue
- P: pool of spare tokens, initially $|P| = S$
- M: maximum number of incoming links (assuming a single router-router connection for each child router)
- L: set of incoming links
- P_l: set of tokens assigned to link l
- c_l: current number of tokens consumed by link l
- u_l: utilization status of tokens assigned to link l
- r_l: child router which created the link l
- B: base number of tokens assigned to a new created link l, $B = S / M$
- b: unit of number of tokens to promote/degrade a link, $b < B$

Actions:

collect_tokens(R)
while $|P| < B$ do
  foreach $l \in L$ where $|P_l| \geq B$ do
    transfer b tokens from P_l to P
notify r_l with the updated $|P_l|$, $l \in L$

adjust_tokens(R)
foreach $l \in L$ do
  if u_l is less than a lower-bound threshold
    transfer b tokens from P_l to P
foreach $l \in L$ do
  if u_l is greater than a higher-bound threshold
    if $|P| \geq B$
      transfer b tokens from P to P_l
    else
      find $l' \in L$ s.t. $|P_{l'}| > B$
      transfer b tokens from $P_{l'}$ to P_l

Intermediate router (R)

upon a new link l is created
if $|P| < B$, then collect_tokens(R)
transfer b tokens from P to P_l
add l into L and notify r_l with $|P_l|$

upon a request is arriving at R from link l
if $c_l < |P_l|$
  receive the request;
  c_l += 1; // *consume a token for* P_l

upon a response is returning to R for link l
if $c_l > |P_l|$
  return a token to P; // regenerate a token for P
else
  return a token to P_l; // regenerate a token for P_l
  c_l -= 1;

periodically
compute *ul* for each link
adjust_tokens(R)

Child router (*rl*, *l*)

upon receiving an token assignment of link l
  update $|P_l|$ for link l

upon relaying a request to link l
  if $c_l < |P_l|$
    do request relaying;
    c_l += 1; // consume a token
  else
    suspend request relaying until a token is regenerated

upon a response returned from link l
  return a token to P_l; // regenerate a token
  c_l -= 1;

Figure 7.1: Token-based flow control algorithm

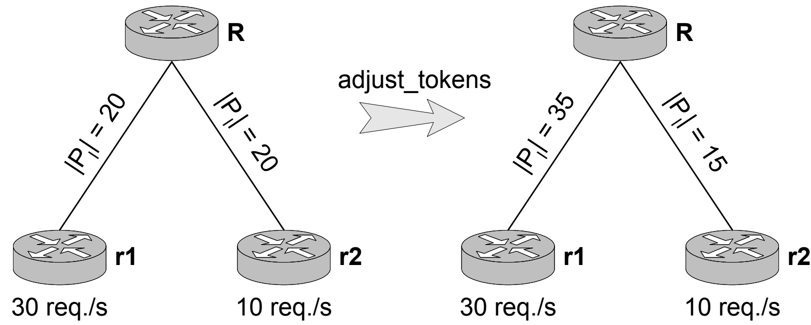


Figure 7.2: An illustrative example for token-based flow control algorithm

or degrade an inactive link by decreasing its assigned tokens. To support such a dynamic token assignment, the intermediate router needs to maintain statistics about the token utilization of each incoming link and periodically promote/degrade a link depending on its token utilization.

Figure 7.2 shows an illustrative example of our token-based flow control scheme. In this example, the intermediate router **R** has two children **r1** and **r2**. Router **R** has a request queue which consists of 500 request slots, and can accept up to 25 child routers. The number of tokens used to promote/degrade a link is set to 5. The utilization of tokens assigned to a link is defined as the ratio of the maximum number of concurrent requests to the total assigned tokens for a link, and the lower-bound and higher-bound of the utilization threshold are set to 0.5 and 1, respectively. The figure shows that initially, both incoming links were assigned a base number of tokens (500 / 25); however, router **R** is able to detect that the token utilization of the link between

R and **r1** is 1 and therefore promotes this link until the total number of the assigned tokens is 35; similarly, router **R** is able to detect that the token utilization of the link between **R** and **r2** is only 0.5 and so decreases the number of tokens assigned to the link to 15.

7.3 Non-blocking/Asynchronous Communication

DataSlicer routers are application-level routers and communication between routers follows these steps: (1) a router accepts an incoming TCP/IP connection, or creates a connection to another router; (2) once connections are connected, the routers exchange various commands via TCP/IP; (3) these commands cause various activities to happen. In general, the performance issues for such application-level routers are to: (1) accomplish many concurrent tasks as quickly as possible, (2) efficiently cope with a great deal of waiting (caused by TCP/IP slowness, or for the other end to send the next command), and (3) perform TCP/IP operations efficiently.

To improve the performance of the routers, we address the above three criteria by: (1) creating persistent connections between routers; (2) applying an event-driven request processing model; and (3) using non-blocking request processing for router-router communication. The first technique allows a pair of routers to communicate with each other via a single persistent connection, which reduces the impact of "slow startup" of TCP/IP protocols and alleviates competition for resources (e.g. sockets). The second technique allows a router to process the incoming commands in a one-thread-multiple-tasks manner, thereby reducing the context-switching overhead incurring in a multi-thread or multi-process model. The last technique allows a router

to initially process an incoming request (inspect the request and relay it to an upstream router), and then continue on to other activities; when the corresponding response returns or an event occurs (such as a time-out event), the router can be notified and react accordingly.

To support the non-blocking request processing model for router-router communication, each router keeps track of the active requests using a hash table which stores a mapping between the message ID and a structure that contains a variety of information regarding the message. Some fields of this structure include:

- TTL value
- Incoming link (socket) information
- Outgoing link (socket) information

Upon receiving an incoming request, a router extracts the message ID and TTL value from the request (if the request is not a wrapped request, the entry router will generate a unique message ID and set the TTL value for this request), and identifies the incoming link and the outgoing link for this request. The router then creates a record for this request which contains the information described above, stores the record into its hash table, and then relays this request (wrapping it if necessary) to its parent. On receiving a response, the router uses the message ID information paggypacked in the response to identify the incoming link information, and relays the response to that link (after necessary processing, e.g. computing the response time of this request, updating the load of the cell at which this request hit, etc.).

The above model assumes that the response contains the corresponding request message ID. This assumption is true for router-router communication since this in-

formation is piggybacked in the wrapped messages. However, once a request leaves our router network, we have no control about how a service replica responds to the request. In other words, an exit router can not identify a request using the message ID information. For this reason, the communication between an exit router and a service replica follows a synchronous model to allow the exit router to identify the request for a response sent from a service replica using the outgoing link information. To improve the performance of router-replica communication, we allow an exit router to establish multiple connections between itself and a service replica (unlike the single connection for router-router communication).

7.4 Smoothing out Statistical Fluctuations in Network Measurements

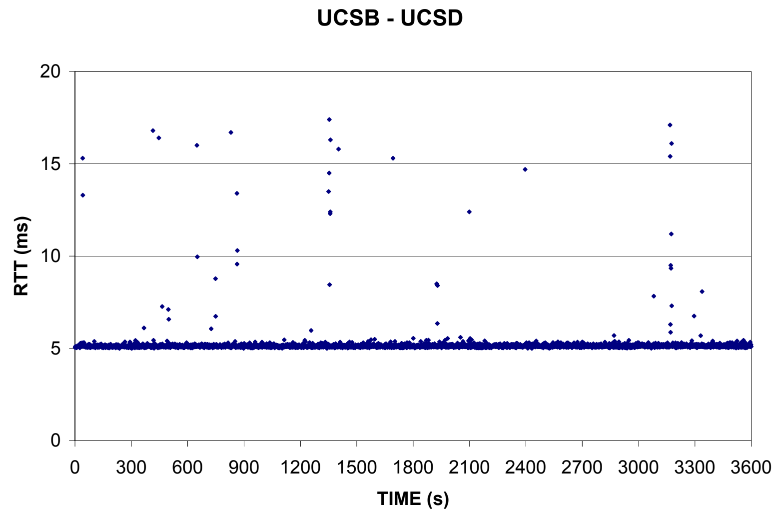
In Chapter 5 and 6, we have described the techniques exploited in the DataSlicer architecture to organize the routers into an oriented overlay network, distribute the requests in the network, and replicate the data regions on-demand. These actions rely on the measurement of some application-specific metrics, such as the network latencies of the links or the request-response times observed by the routers. The application-specific metrics are in turn influenced by the underlying network metrics, such as latency, bandwidth, and node CPU utilization, etc.

A problem that one encounters in a real-world network is that values of network metrics tend to fluctuate dynamically, which complicates the measurements taken by the routers. For example, Figure 7.3 shows the measured round-trip latencies over a one hour period for the link between UCSB and UCSD nodes and the link between

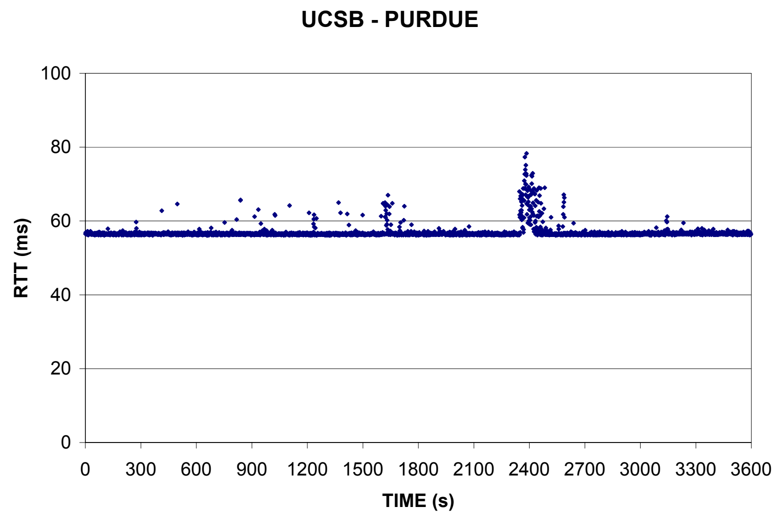
UCSB and Purdue nodes (all of the nodes are PlanetLab nodes). The average latency of the link between UCSB and UCSD is approximately 5 ms; however, there exist scattered points during which the measured latencies are dilated up to 18 ms. On the other hand, the average latency of the link between UCSB and Purdue is approximately 57 ms, and the latency fluctuation of this link is not as frequent; however, there is a sudden increase in latency to about 80 ms at time 2400 s which lasts for minutes.

Consequently, such fluctuating measurements could lead to an unstable overlay network where routers might frequently jump up and down between zones because of the dynamic change in network latencies. Similarly, the fluctuating measurements could cause unnecessary service replications or avoidable fault recovery actions since a sudden increase in the request response time observed by a router could trigger actions such as load-balancing or service replication. It might appear that taking the average of a number of measurements should solve this problem; however, this approach does not quite work as expected when fluctuations although short-lived, produce wide variations in measured values. Taking the average over a long sequence of measurements could smooth out these effects, but at the cost of responsiveness. Additionally, the statistical mean works only for true normal (gaussian) distributions and will be tugged from the central tendency in response to a small number of outliers, or to skewing of the tails of a distribution. The statistical median can address these problems, but is costly in terms of storage and computation.

To ensure both stable measurements and not sacrifice responsiveness, we employ a two-level measurement strategy which uses a hybrid of these two statistical values to smooth out the fluctuations in metric-measurements, as described below:



(a)



(b)

Figure 7.3: Network latency measurements on the PlanetLab network.

- Compute the statistical mean value of a measurement over a (short) time period.
- Compute the statistical median value of these mean measurements (over a longer time period, which is viewed as a collection of multiple smaller periods).
- Depending on the requirements in practice, one can either use the mean values to stand for the measurement of a metric over the smaller time granularity, or use the median of means values to stand for the measurement of a metric over the longer time granularity.

The median of means computation proves to be very robust over the fluctuations we have observed in a real-world network, and is what DataSlicer uses for deciding on potentially expensive decisions such as the assignment of router *zone ranks* in the oriented overlay construction and maintenance, or determination of the service replication actions. For decisions that are not as expensive, e.g., load balancing traffic across the outgoing links, we work with the mean measurements over a smaller time granularity.

Our two-level metric-measurement strategy provides a good tradeoff between stability and responsiveness where one has the flexibility to select an appropriate statistical value and time granularity to measure the performance metrics which will then be used as an input for different algorithms to make decisions. Additionally, the hybrid of the two statistical values reduces the cost of storing original measurements needed for a median computation, and therefore improves the efficiency of the computation.

7.5 Summary

This chapter has described four techniques to maintain system robustness in our DataSlicer architecture, which are motivated by our experiments running on a real-world network for extended periods. We do not really invent novel solutions, but adopt or integrate existing approaches to address these challenges. The selection of a particular approach over many other alternatives follows the principles of simplicity, efficiency and scalability in wide-area networks.

Chapter 8

Evaluation

This chapter evaluates the benefits and the costs of using the DataSlicer architecture to host data-centric network services in wide-area network environments. The main goals of this evaluation are to answer the following questions:

- Does the oriented overlay construction bring benefits for clustering client requests according to various application-specific metrics?
- Can locality information, if it exists, be detected dynamically from the underlying traffic at different intermediate routers?
- How well do the load-balancing and service replication techniques cooperate with each other to provide QoS-assured services to clients? Are the actions of service replication "reasonable", i.e., without incurring redundant or wasteful replications?
- Does the architecture deliver robust performance in a real wide-area network?

The first part of this chapter describes the implementations of two prototypes of the DataSlicer architecture, and the testbeds on which we conduct our experiments. The second part presents the results for the evaluation of the questions listed above.

8.1 Experimental Environment

We have built two prototypes of the DataSlicer architecture. The first prototype was built using C#, SOAP, XML/XSLT and demonstrates the utility of the DataSlicer architecture in the context of application-level SOAP routers and standard Web Services based client-server interactions. This prototype has the advantage that most of the interactions between the service providers and the infrastructure can be expressed at a high level (e.g. as XSLT programs) and can leverage existing protocols such as WS-Routing and WS-Referral. However, because of the cost incurred in the design, this infrastructure suffers from scaling issues. The second prototype was built from the ground up to support scaling and evaluation on a real-world network. Given our choice of the real-world network, the PlanetLab network which is shared by many research institutes, we ended up with a low-overhead C-based implementation, where the interactions between the service providers and the infrastructure are a little more involved with DLL issues, but the infrastructure allows us to experiment with scaling and robustness issues much better. Depending on the size of the network/level of complexity, one can imagine either of these prototypes being useful for a real-world deployment.

To permit exploration of a range of behaviors, our experiments use a synthetic service that reflects characteristics common to data-intensive map- and imagery-services

such as MapPoint, SkyServer, or TerraServer. In defining the service data space and other request parameters, we use as a guide the real MapPoint service: our service supports queries for maps in North America and Europe using geographic longitude and latitude information. Each request has a size of 4KB, and the map is rendered into a 400×400 pixels image box, resulting in a response size of $\sim$34KB. Given the fixed image size, the map requests are solely determined by the location of the map center point and the scale of the map. Therefore, the logical data space can be defined by three attributes: latitude, longitude and map scale, with an individual map request viewed as a point in this logical data space. Our cell structure supports splitting up to 6 levels, partitioning the logical data space into a maximum of 2^{18} regions. The smallest region, for a resolution of 50000:1, corresponds to map information at the city level and involves $\sim$11MB of data.

8.1.1 C# Prototype

The C# implementation of the DataSlicer architecture contains approximately 5,000 lines of C# code and is built on top of Microsoft's ASP.NET Framework 1.1 and Microsoft Web Services Enhancement Package (WSE), v2.0. The router implements the ASP.NET *IHTTPHandler* interface with extensions to support WS-Routing and WS-Referral protocols in order to provide SOAP routing functionality, and is registered as a custom HTTP handler with a web server running Microsoft Internet Information Services (IIS) 6.0, which is responsible for receiving/forwarding the HTTP messages.

To evaluate the performance of this prototype, we conducted experiments on an emulated WAN. The emulated network used Click, a modular software router developed at MIT, to emulate a WAN environment using a LAN cluster which consisted of

8 routers (512MB memory, 1GHz CPU) and 32 clients. The emulated network consisted of 8 network domains, each with 1 router and 4 clients, with the inter- and intra-network configurations (latencies, bandwidths) set based on representative measurements from the PlanetLab network. These 8 networks were manually organized into a tree topology with the router in the root domain serving as the origin server hosting the synthetic service. Clients request the service via the routers that sit within their local network domain.

8.1.2 C Prototype

The C implementation of the DataSlicer architecture contains approximately 16,000 lines of C code, of which 3,500 lines are in an overlay construction module structured for standalone use. The service handlers running at each router are implemented using two ports. One is a public port accessible from end-clients, the other is a private port used by the routers to relay service requests/responses through the overlay network. All of the message relaying and the internal communication between the routers is processed on an event-driven basis using non-blocking network connections to efficiently utilize the CPU power of a node.

The synthetic service was hosted on an origin server residing at New York University. The networks (including the DataSlicer router network and the service replica network) were constructed from about 150 – 190 PlanetLab nodes, distributed across North America and Europe. For the nodes chosen to act as routers, each node hosted a router application and a varying number of clients (from 1 to 20), and was configured to be able to accept up to 20 children and select up to 3 parents. The origin server and the replica sites hosted the service on top of Apache HTTP Server Version 2.0,

using an open-source module *mod_fastcgi* 2.4.0 from FastCGI.com. The origin server (512MB memory, 1GHz CPU) was capable of serving approximately 425 requests per second with 1KB response size, and about 215 requests per second with 10KB response size. However, the throughput of a replica node could be far less than that seen by the origin server because of resource sharing among multiple PlanetLab users.

In the next section, we first evaluate the DataSlicer architecture on a tree-based emulated network, focusing on the investigation of the effectiveness of the locality detection technique and the tree-based service replication algorithms. The following sections describe the investigation of the DataSlicer architecture on the PlanetLab network, and present the results of our oriented overlay construction scheme, and the effectiveness of our load-balancing and service replication techniques on a DAG-topology network.

8.2 Evaluation of C#/.NET Prototype

We start our investigation of the performance of DataSlicer on a tree-based emulated network, which provides a controllable testbed in terms of network latency and bandwidth, as well as node CPU utilization, to evaluate: (1) the ability of DataSlicer to detect the locality patterns in the underlying traffic; and (2) the ability of DataSlicer to replicate portions of the service with minimum cost.

8.2.1 Configuration of the Emulated Network

Figure 8.1 shows an overview of the network configuration we use for the experiments on the emulated WAN. The configuration consists of eight network domains, each

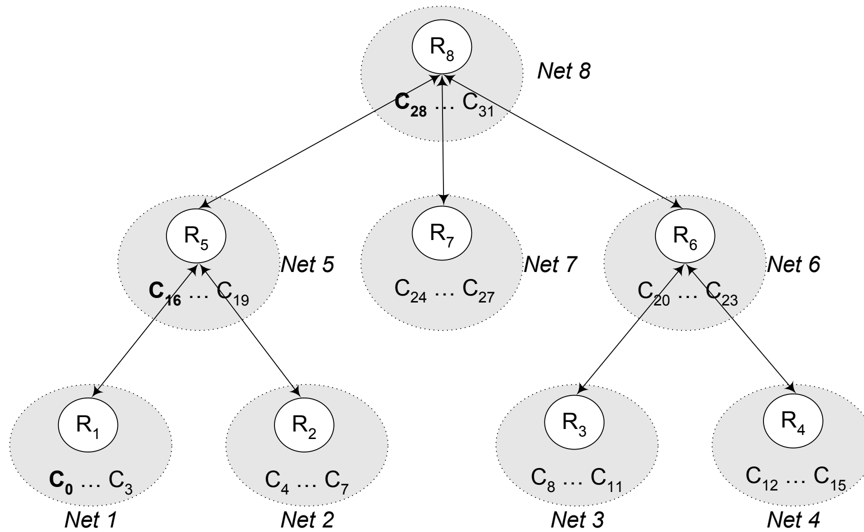


Figure 8.1: Configuration of the emulated network.

with a router and four clients that generate the service requests. The 8 router nodes ($R_1, \ldots, R_8$) are organized into a tree as shown in the figure, and serve as entry routers for the four clients in the same domain. R_8 acts both as a root router and as the location of the origin service. All of the routers are replica routers, i.e., a service replica can be created on any of the routers.

The configuration of the emulated network, i.e., network latencies and network bandwidth, come from measurements we took between pairs of PlanetLab nodes over an extended period (the bandwidth values were scaled down by a factor of 3 so as to accommodate the hardware limitation of a 100 Mb/s switch in our emulated system). Table 8.1 shows the close correspondence between the metrics measured on the two systems.

Table 8.1: Network metrics on the PlanetLab Network and the Click-based emulated WAN.

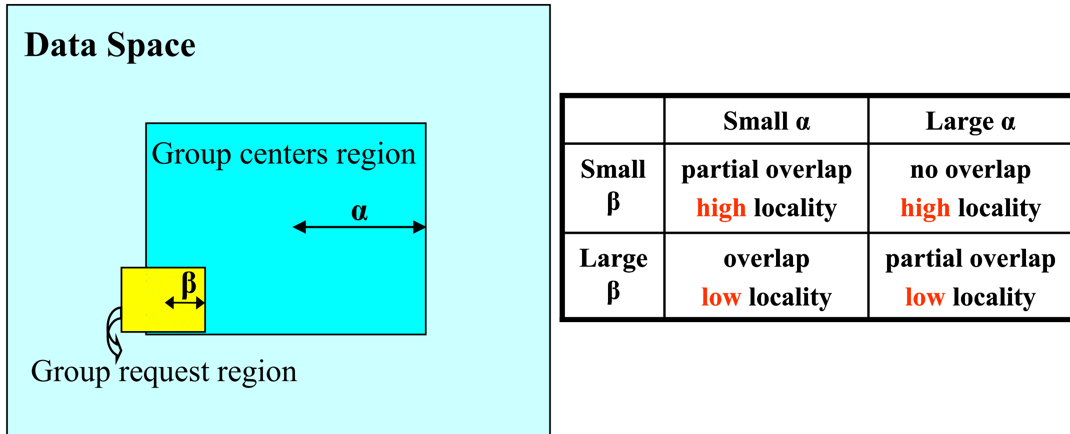
PlanetLab			
Path	RTT (ms)	b/w (Mb/s)	
		UDP	TCP
umich-caltech	72.75	38.24	6.79
umich-washington	66.49	44.68	6.0
columbia-cmu	73.79	5.68	5.36
columbia-princeton	13.89	45.76	17.28
nyu-umich	46.25	37.84	7.52
nyu-columbia	9.31	45.63	10.34

Emulated Network			
Link	RTT (ms)	b/w (Mb/s)	
		UDP	TCP
net1-net5	72.75	12.76	2.28
net2-net5	66.49	14.89	2.0
net3-net6	73.78	1.90	1.5
net4-net6	13.89	15.29	5.77
net5-net8	46.25	12.62	2.51
net6,7-net8	9.31	15.21	3.45

8.2.2 Client Workload

Clients repeatedly send requests to the service, waiting for a response before sending the next request. The size of messages came from the measurement of the real MapPoint service, i.e., 4KB for a request and 34KB for a response. To prevent saturating our underlying emulation system, each client is restricted to generating at most 5 requests every second (the actual rate might be lower because of congestion).

The workload generated by the clients reflects the results described in Chapter 4, which showed that real workloads exhibit locality in both the regions of the service's data space they access and at the network level. To understand how our architecture and algorithms behave for these kinds of locality, our clients send requests according to the following pattern: they first select a rectangular region in the data space (*global region*) such that all clients within the same domain first agree upon a group center point within the global region, and then randomly request a point within a new rectangular region (*domain region*) surrounding the group center point. Thus, by controlling how close the group centers of different domains are to each other (denoted as



	Small α	Large α
Small β	partial overlap high locality	no overlap high locality
Large β	overlap low locality	partial overlap low locality

Figure 8.2: Usage of parameter α and β in generation of client workloads.

the parameter α, whose value is the ratio of a side of the global region to the range of the corresponding dimension in the origin data space), and how large the rectangular region is for each group (denoted as the parameter β, defined similar to α above but for the domain region), we can generate workloads that exhibit either spatial locality, or network locality, or both (see Figure 8.2).

For each of the experiments, all 32 clients generate requests against the service simultaneously. The tree-based service replication algorithm is given as input a maximum client response time threshold of 500 ms, the usage statistics and the response time estimates for cells, and generates as its output a service replication strategy. To prevent overwhelming the network, we restricted that concurrent replica creations at a router had to happen in sequence.

In the rest of this section, we show and discuss only results for representative clients along one of the longest paths in our network configuration: net1 (c_0) - net5 (c_{16}) - net8 (c_{28}). Since the network configuration and client behavior is symmetric,

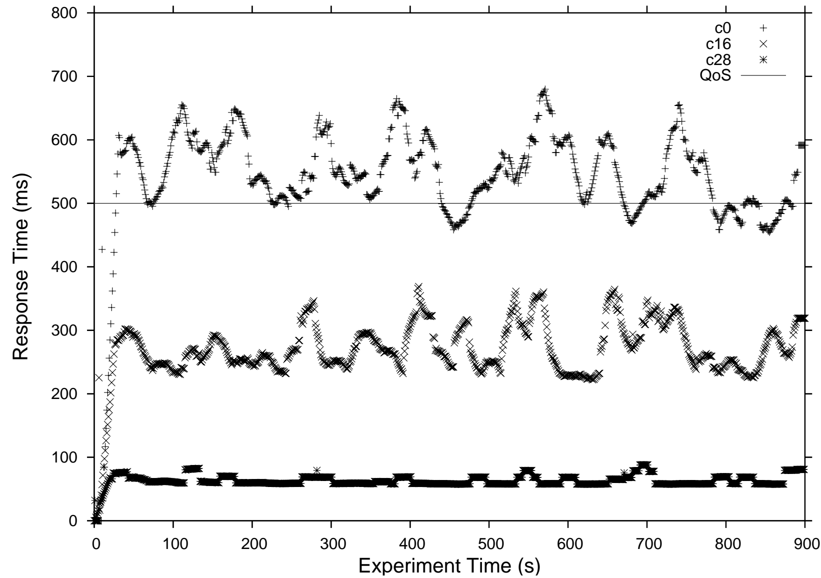


Figure 8.3: Performance on an unloaded network.

the performance of the other clients tracks the ones reported. The graphs presented below show the moving average of the response time observed by the last 20 requests received at a client, and is computed every second.

Before discussing how our architecture performs in the complicated scenarios where the workloads exhibit different kinds of locality, we first present the baseline response time seen by clients in an unloaded network. Figure 8.3 shows this data for our three clients. The fluctuation in the response times stems from the (emulated) behavior of the network paths and typifies the same characteristics as our PlanetLab measurements. The response time seen by an individual request has two components: round-trip time of the network path and the (non-overlapped) latency introduced by the computation at each router. The first component dominates: each router adds

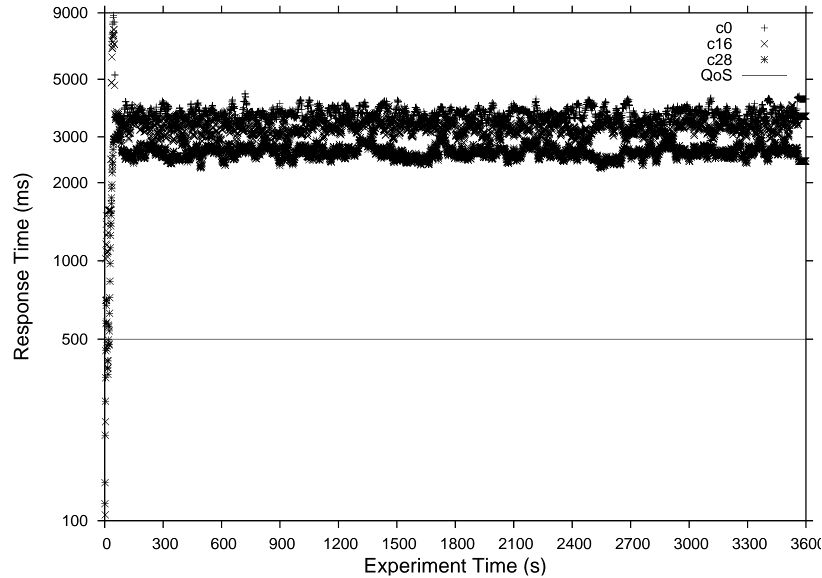


Figure 8.4: Performance seen for a workload that exhibits distributed grouping and low spatial locality.

$\sim$ 20ms to the overall response time, with only 5 ms attributable to our code (the rest is caused by the .NET framework implementation).

Note that the response time threshold was not satisfied at c_0 even in the unloaded network. Once the network is loaded, without service replication, none of the clients can be satisfied (even clients in the same domain as the origin service who get affected by the fact that the service needs to handle a large number of requests).

Distributed grouping, low spatial locality.

In the first experiment, each client randomly requested a region in the data space, producing overall low spatial locality in the workload. Since the network was con-

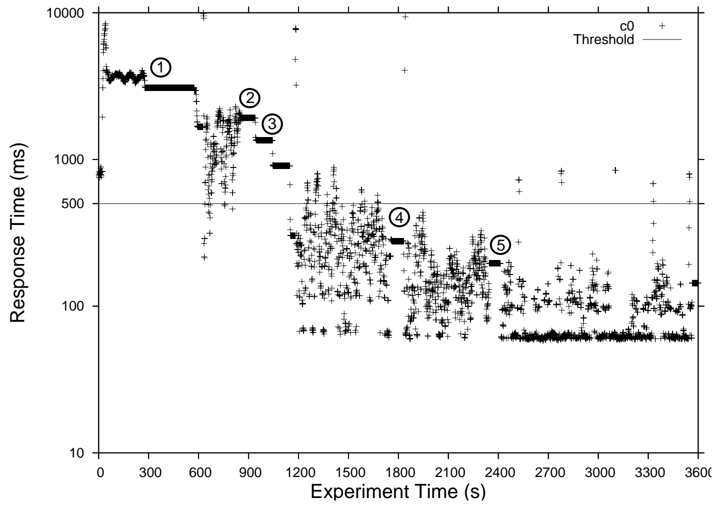
gested, Figure 8.4 shows that all three of our clients observed response times far in excess of the desired threshold. Our architecture performed no replication because the usage in all regions of the data space stayed below the configured threshold of 500 requests per 30 seconds.

Distributed grouping, high spatial locality.

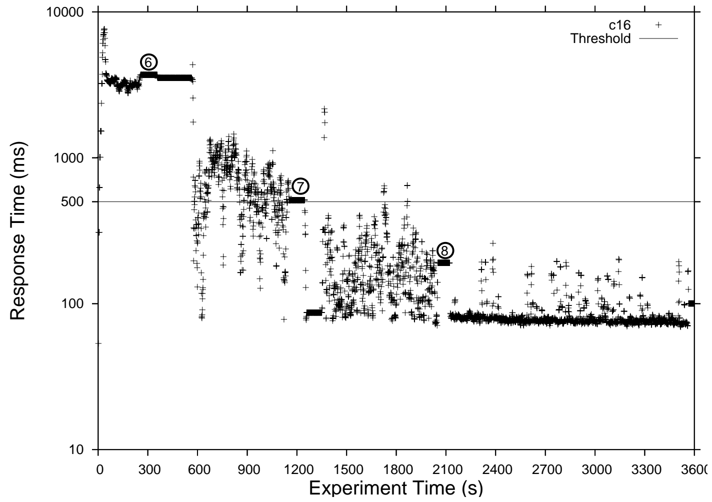
In the second experiment, each group of clients within the same domain targeted their requests to a small region in the data space. However, because group centers might be far away, one would be unlikely to find much overlap between the regions targeted by clients from different domains.

Figure 8.5 shows that our architecture was able to dynamically detect such locality and replicate portions of the service properly to satisfy client QoS thresholds on response time. Given the locality structure, service regions were replicated at the router node in a domain to satisfy that domain's clients. In this case, 3 regions with high access rates, 0030223, 0030222, and 0032000,[1] were replicated on R_1 starting at times 300s, 900s and 2400s, respectively. The first replication event did not happen until at time 300s in the experiment because of the router configuration. Similarly, the 3 regions accessed by the clients in Net5, 0212333, 0213220, and 0212331, were replicated at the domain router, R_5, starting at time 300s, 1200s, and 2100s, respectively. Note that each replication request resulted in a 11MB data transfer across a congested network path: each such transfer took approximately 300 seconds, and had the effect

[1] The region ID corresponds to the path in the cell tree taken to reach this region. For a 2-dimensional space, each split produces four subcells that are labelled 0–3. The ith digit in the region ID corresponds to the parent subcell of the current region at level i.



(a) c_0



(b) c_{16}

Event	Region	Router	Replica lifetime	Event	Region	Router	Replica lifetime
1	0030223	R_1	[300, -]	5	0032000	R_1	[2400, -]
2	0030222	R_5	[895, 1620]	6	0212333	R_5	[300, -]
3	0030222	R_1	[900, -]	7	0213220	R_5	[1200, -]
4	0032000	R_5	[1800, 2640]	8	0212331	R_5	[2100, -]

(c) Replica placement

Figure 8.5: Performance seen for a workload that exhibits distributed grouping and high spatial locality workload.

of serializing the replication requests from different routers. Consequently, it was only at time 2100s and 2400s that the clients of Net5 and Net1 saw response times below their required thresholds. The spikes in response times seen at various points in the graphs (e.g., at time 600s and 1200s in the c_0 graph) can be explained as follows. Since a router queues up client requests for a region that is being replicated (the intuition here was to not have new requests compete with replica creation traffic for scarce network bandwidth), once the replication completes and these requests are serviced, their response times reflect the queuing time as well.

Figure 8.5 also shows two other interesting points. First, note that some regions were replicated in a redundant fashion: regions 0030222 and 0032000 were first replicated on R_5 and then re-replicated on R_1. There are two explanations for this, which are addressed by the C-based implementation we describe latter in this chapter:

- Our tree-based replication algorithm looks at the current round-trip time estimate between an intermediary router and its parent to decide where best to perform the replication. However, once a replica is created at the parent, it is possible that the parent can service more client requests per unit time, which in turn increase queueing delays and hence the round-trip times seen by requests coming from the child router. What is required is a better way of estimating the round-trip time that would result after replication. Region 0032000 falls into this category.

- Each router operates asynchronously, with a thread waking up every 300 s to participate in the distributed replication algorithm. The following situation is thus possible: in one round, a router might find the request load for a region

to be below the threshold required to request replication and consequently send an "Unsatisfied" message to its parent, while in the next round, the threshold may get crossed causing the router to initiate replication on its own. If the parent processes the first message late, it might end up seeing a request load that exceeds the configured threshold, and thus request replica creation. In our experiment, region 0030222 falls into this category. Better synchronization between the routers would fix this problem.

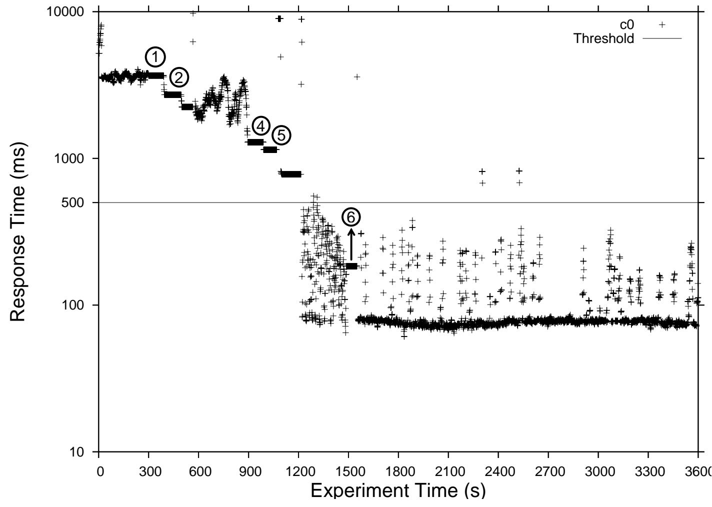
These missteps were corrected in subsequent timesteps, with the replicas at R_5 getting reclaimed at times 1620s and 2640s respectively because of inadequate use.

What is interesting, and this is the second point, is that before the replicas get reclaimed, they have an unexpected benefit: of reducing the latency for the replication request for region 0030222 from region R_1 at time 2400s, which was now satisfied by R_5 instead of going all the way to the origin service. This short-circuit manifested itself in the fact that response time seen by c_0 improved fairly quickly after the replication was requested, unlike the behavior observed for the earlier requests.

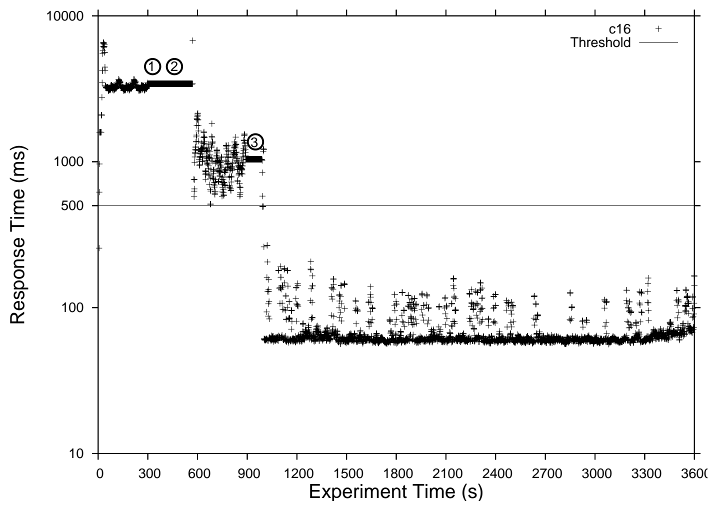
Centralized grouping, high spatial locality.

In the third experiment, the workload pattern presented high spatial locality over the entire network, with all clients requesting data from a small shared region.

Figure 8.6 shows that the DataSlicer architecture was able to detect this kind of locality and replicate service regions properly. Regions 0211111 and 0122222 were two of the commonly requested regions and hence were replicated at both R_1 and R_5. In this case, the redundant replication was warranted: clients in Net5 needed to have



(a) c_0



(b) c_{16}

Event	Region	Router	Replica lifetime	Event	Region	Router	Replica lifetime
1	0211111	R_5	[300, -]	4	0033333	R_1	[900, -]
2	0122222	R_5	[300, -]	5	0211111	R_1	[900, -]
3	0300000	R_5	[900, -]	6	0122222	R_1	[1500, -]

(c) Replica placement

Figure 8.6: Performance seen for a workload that exhibits centralized grouping and high spatial locality.

the region replicated in R_5 to satisfy their response time threshold requirement, while clients in Net1 could not have their response time requirements satisfied with a replica at R_5 and hence, needed a closer replica. Rerunning the experiment with the response time threshold raised to a higher value, 1500 ms, highlights this point: in this case, replicas at R_5 sufficed for clients in both Net5 and Net1.

Centralized grouping, high spatial locality, high network locality.

In the fourth experiment, the workload was the same as the previous one, except that clients in Net7 contributed more than 90% of requests in the overall workload. This scenario matched the locality patterns we encountered in our study of the SkyServer and TerraServer traces and was properly handled by our architecture: replica creation only happened on R_7, the router node of the domain whose clients made most of the requests.

8.3 Evaluation of C Prototype

As mentioned earlier, the difficulty that prevented us from evaluating the C# prototype on a real-world network like the PlanetLab network is that the open-source implementation of the ASP.NET framework for UNIX environments is computationally costly, and therefore does not scale well. As a result, we ended up re-implementing a low-overhead C-based prototype of our architecture which allows us to investigate the impacts of techniques such as overlay construction, load-balancing and service replication on a large-scale DAG-topology network, and experiment with system robustness issues.

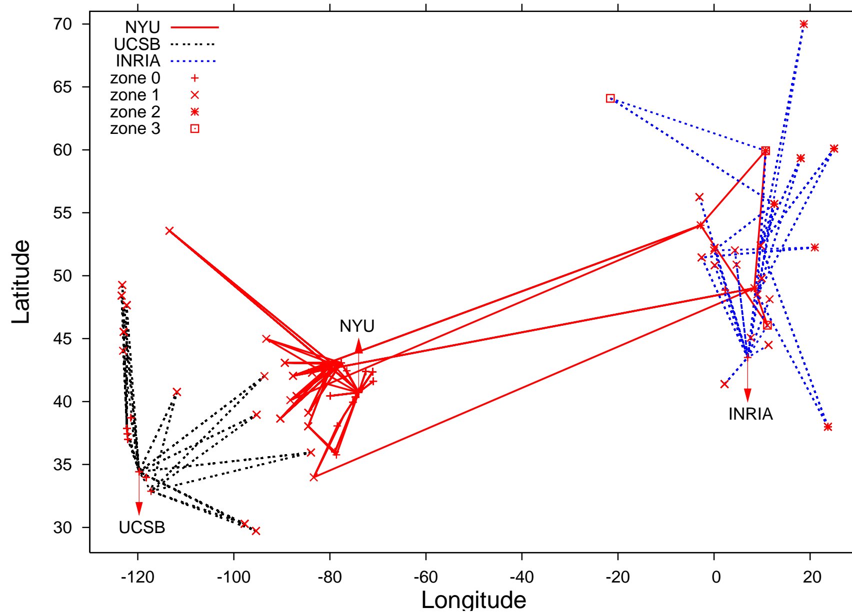


Figure 8.7: An oriented overlay with 3 origin servers constructed on the PlanetLab network.

8.3.1 Oriented Overlays Characteristics

We start by describing the characteristics of the underlying overlay networks that got built in our deployments. Our zone-based scheme was configured to partition nodes into five zones with the following criteria: *Zone0* corresponds to a round-trip time from the origin server of $0 \sim 20$ ms, *Zone1* to $20 \sim 60$ ms, *Zone2* to $60 \sim 100$ ms, *Zone3* to $100 \sim 200$ ms, and *Zone4* to more than 200 ms. For an overlay that is built using the PlanetLab nodes spanning the United States and Europe and an origin server residing at New York, we intuitively expect nodes in north-east United States to fall into Zone0, nodes in the central portions of the United States to fall into Zone1, nodes in the west United States to fall into Zone2, and nodes in Europe to fall into Zone3 or Zone4.

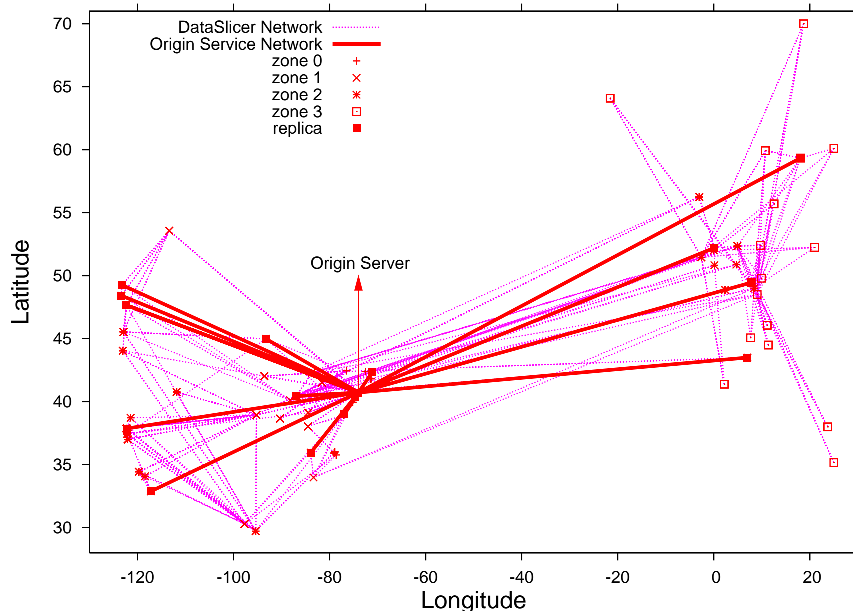


Figure 8.8: An oriented overlay with a single origin server constructed on the PlanetLab network.

To illustrate the generality of our oriented overlay construction scheme, Figure 8.7 shows the oriented overlay networks built from the 150 nodes in a system with three origin servers, residing respectively in the east coast of the United States (NYU), the west coast of the United States (UCSB) and in France (INRIA). The results show that most of the nodes participate in the overlay oriented towards an origin server which was geographically closest. However, a few nodes violated this geographical proximity rule: four nodes in Europe participated in the overlay associated with the NYU server instead of the INRIA server because of smaller round-trip latency. Since the number of such violations is very small, it does not affect the metrics of our constructed overlays.

In the rest of this subsection, we characterize the properties of these overlays,

restricting our attention to overlays oriented towards a single origin server at NYU. Figure 8.8 shows this overlay, which involved 149 participating nodes (16 service replicas and 133 routers) distributed across North America and Europe. The origin service maintained network is shown by the solid lines, and the oriented overlay network is shown by the dashed lines. Although a node only took about 1 second to become a member of the formed overlay, the construction algorithm took approximately 600 seconds to stabilize the overlay because the origin server updates the nodes with advised parent candidates using the longer timing granularity of 600 seconds.

We look at three aspects of the overlays: (1) the nature of the overlay (how nodes are partitioned into zones, and what kind of connectivity the overlay provides), (2) the performance of the overlay (to what extent is network latency improved/impaired), and (3) the ability of the overlay to cluster client requests.

Table 8.2 shows that 30 (20.14%) nodes were partitioned into Zone0, 29 (19.46%) nodes into Zone1, 61 (40.94%) nodes into Zone2, and 29 (19.46%) nodes into Zone3. For the nodes in Zone0, the out-degree was always 1 since these nodes could only select the origin server as their parent. Most of the nodes in other zones were able to connect to 3 parent routers, the maximum allowed in our configuration of experiments. Some nodes also established an additional connection to a replica, resulting in an average out-degree of 3.22.

To understand what kind of impact our overlay has on a node's network latencies, we compare the average latency seen by a node (computed by taking the average of the latencies of all paths from the node to the origin server) in the constructed overlay with that experienced by a direct connection between the node and the origin server. The ratio of these values is called $Latency_Dilation$. Table 8.3 summarizes the latency

Table 8.2: Node distribution in the constructed oriented overlay with a single origin server

Zone level	Nodes	InDegree	OutDegree
0	30	2.67	1.00
1	29	6.00	3.10
2	61	1.77	3.31
3	29	1.03	3.17

Table 8.3: Latency dilation in the constructed oriented overlay with a single origin server.

Dilation	Nodes	Zone level			
		0	*1*	*2*	*3*
$0 \sim .5$	17	4	4	7	2
$.5 \sim .8$	34	0	6	28	0
$.8 \sim .95$	24	7	10	7	0
$.95 \sim 1.05$	35	16	8	7	4
$1.05 \sim 1.2$	19	0	1	8	10
$1.2 \sim 1.5$	17	0	0	4	13

dilations for participating nodes. As expected, for nodes in Zone0, the latency was the same as would be seen by a direct connection to the origin.[2] For nodes in the other zones, the overlay did not overly dilate the path latencies, with the worst dilation seen by nodes in Zone3 which found their performance impaired by a factor of up to 1.5. Somewhat surprisingly, a significant number of the nodes in Zone1 (10 out of 29), in Zone2 (35 out of 61), and in Zone3 (2 out of 29) achieved a better latency by a factor of 20%.

The primary use of the oriented overlays in our architecture is to cluster nodes that have correlated service usage patterns, and as stated earlier, we assume that for

[2] The fact that some Zone0 nodes are shown with latency dilation values smaller than 1 is attributable to small (expected) measurement perturbations because of dynamic network conditions. Given the low absolute values of latencies in these cases, these perturbations sometimes result in large variations in the latency dilation value.

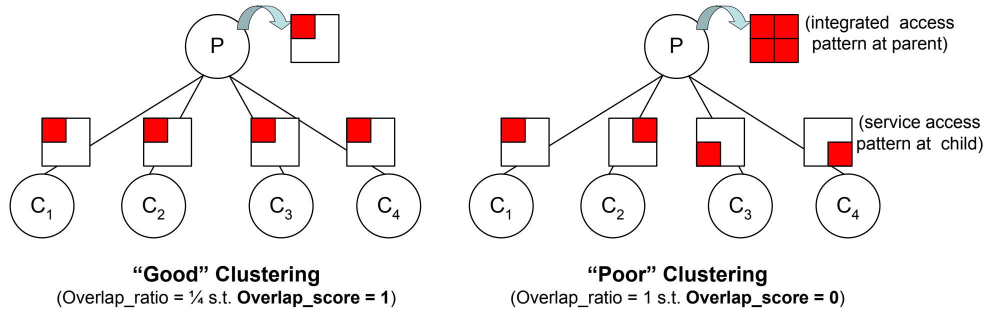


Figure 8.9: Examples of clustering in oriented overlays, evaluated using overlap score.

our service, geographical proximity is an indicator of request similarity. To quantify the clustering that our overlay network produces, we model service usage as follows. Each node is associated with a geographic region where it itself sits at the center. This region represents the portion of the service data accessed by client requests originating at that node. A measure of request clustering on an intermediate node is the overlap between the geographic regions of its child nodes. We define the overlap ratio (Ratio) to be the ratio of the area of the union of the child regions to the sum of the areas of these regions. The goodness of clustering is measured by an overlap score that compares this overlap ratio to the ideal case — where all of the child nodes reside at the same location (and hence the overlap ratio is $1/\#(\text{children})$):

$$\text{Score} = (1 - \text{Ratio})/(1 - (1/\#(\text{children})) \tag{8.1}$$

Obviously, the closer the overlap score value to 1, the better the clustering. Figure 8.9 shows two examples of clustering: the left figure shows an ideal clustering where all of the children of P share the same pattern, resulting in an overlap score of 1; the right figure shows a poor clustering situation where for node P, none of its children share a common access pattern, resulting in an overlap score of 0.

Table 8.4: Clustering in the constructed oriented overlay with a single origin server.

Region size: 3° long. × 3° lat.					
Score	Nodes	Zone level			
		0	*1*	*2*	*3*
0 ∼ .2	2	0	2	0	0
.2 ∼ .4	3	0	1	2	0
.4 ∼ .6	18	6	5	6	1
.6 ∼ .8	10	6	4	0	0
.8 ∼ 1	5	3	2	0	0

Region size: 5° long. × 5° lat.					
Score	Nodes	Zone level			
		0	*1*	*2*	*3*
0 ∼ .2	1	0	1	0	0
.2 ∼ .4	3	0	1	2	0
.4 ∼ .6	5	1	1	3	0
.6 ∼ .8	21	10	7	3	1
.8 ∼ 1	8	4	4	0	0

Table 8.4 shows the overlap scores computed on intermediate nodes (the entry nodes have an overlap score value of 1). With the size of the associated region set as 3.0° longitude by 3.0° latitude, 47.37% of intermediate nodes (18 out of 38) score between 0.4 ∼ 0.6, and 39.47% of intermediate nodes score higher than 0.6. As we increase the size of region to 5.0° by 5.0°, the percentages change to 17.24% and 76.32%, respectively. Given the fact that nodes are rather geographically diverse, our overlay construction algorithm demonstrates the ability to cluster nodes that are geographically close together, thereby permitting service usage locality to be easily detected on intermediate nodes.

8.3.2 Performance on Small-scale Networks

To investigate how well the DataSlicer architecture improve performance via on-demand load-balancing and service replication, we conducted multiple experiments on a small-scale network.

Configuration.

This small-scale network consisted of a single origin server residing in NYU, and 3

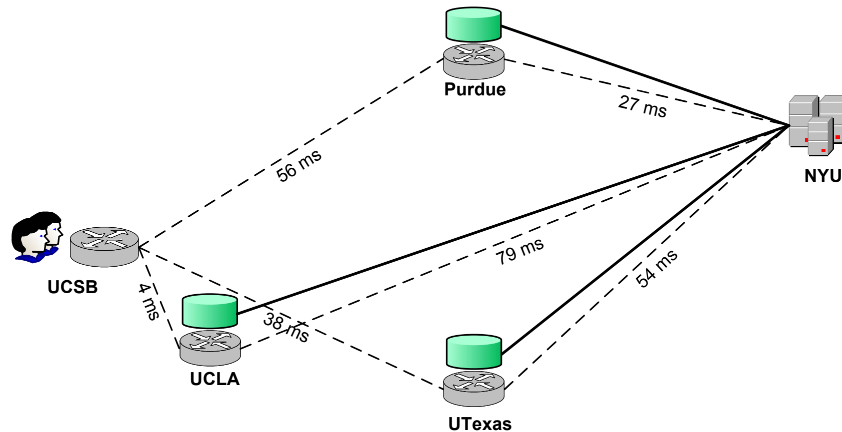


Figure 8.10: Configuration of the small-scale network on the PlanetLab network.

service replicas and 3 intermediate routers that resided at Purdue University (Purdue), University of Texas at Austin (UTexas), and University of California at Los Angeles (UCLA), respectively. Each intermediate router associated itself with the co-located replica and was connected to the origin server. The network contained a single entry router, which resided at the University of California at Santa Barbara (UCSB), and was connected to all of the intermediate routers. Additionally, the network included five clients, which resided on the same host as the entry router.

The configuration of this small-scale network and the round-trip latency of each link is shown in Figure 8.10. To achieve our desired topology, we augmented the network latency range definition for Zone1 to be between 20 $\sim$ 80 ms for our overlay construction algorithm. The statistical median request response times measured at the entry router, were about 94 ms for path UCSB - UCLA - NYU, 93 ms for path UCSB - Purdue - NYU, and 102 ms for path UCSB - UTexas - NYU. Given that a request involves approximately 1 to 2 ms of server processing (on a 256 MB, 1 GHz standalone host and likely more on a shared PlanetLab host), a client quality of service

requirement of 90 ms would make the entry router unsatisfied and require a replica to be created on one of the intermediate routers.

The two timing granularities (for the two-level measurement scheme described in Chapter 7) were configured to be 60 seconds and 600 seconds respectively. Service replication was only invoked for cells that (1) were unsatisfied; (2) received more than 500 requests over a period of 600 seconds; and (3) were at a splitting level deeper than 3 (roughly corresponding to map information at the state level or smaller at resolution 50000:1).

Clients sent service requests at a rate of 5 per second to their co-located router. The sizes of request and response messages were 0.2KB and 1KB, respectively. For simplicity, our clients repeatedly requested the maps in a small geographic region, resulting in only one leaf-level cell being detected as presenting access locality at the entry router.

Results.

Figure 8.11 shows the statistical mean response time measured at the clients over a timing granularity of 30 seconds during the run of the experiment. The results show that at the beginning of the experiment, the clients saw a mean response time in the range 95 $\sim$ 100 ms and were unsatisfied. At time 600s, a replica of the cell that was being accessed was created at UCLA, reducing the mean response time to about 72 ms. At time 990s, we injected a node failure into the architecture, which shut down the intermediate router at UTexas. This failure was detected quickly because the UCSB router found its upstream link to the UTexas router had been broken. Consequently, the UCSB router redistributed the fraction of requests that were assigned to that bro-

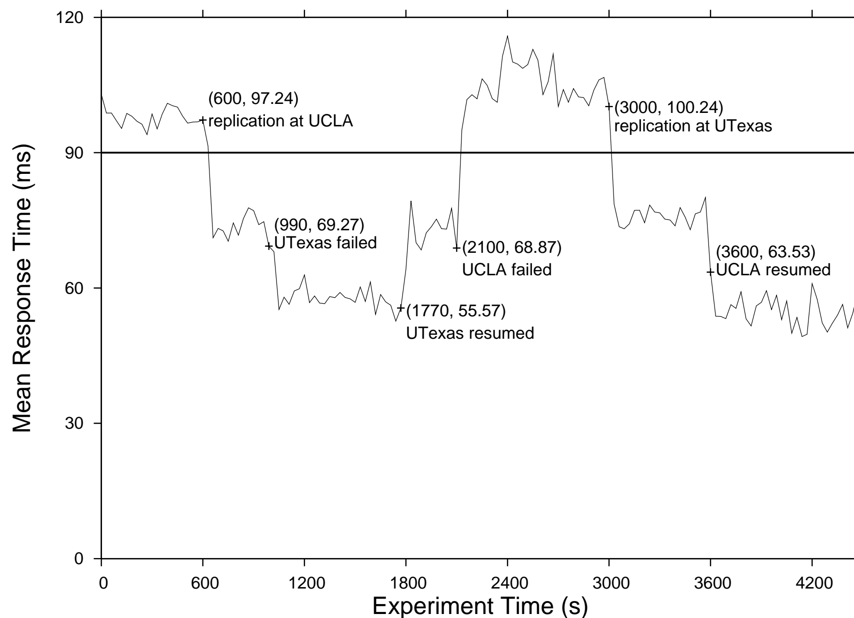


Figure 8.11: Performance observed by a client at UCSB in the small-scale network experiment.

ken link to the others, resulting in the mean response time further decreasing because of increased traffic now being served by the UCLA replica. At time 1770s, the UTexas router was resumed. As a consequence, the UCSB router redistributed the traffic to all of the three links, causing the mean response time to approach its pre-failure value. At time 2100s, we injected another node failure at the UCLA router, which increased the mean response time sharply to above 100 ms because the UCLA replica was no longer reachable. This resulted in the DataSlicer architecture requesting that another replica be created at the UTexas site, which dropped the response time below the desired threshold. At time 3600s, the UCLA router was resumed and further reduced the mean response time because now requests were being sent to two replicas near the intermediate routers.

Table 8.5: Load-balancing on the small-scale network.

QoS threshold	Flow fraction			Statistic Mean
	UCLA (11.5)	UTexas (50.5)	Purdue (99.5)	
50	*33*	*34*	*33*	54
	39	*34*	*27*	49
	43	*34*	*23*	45
30	*33*	*33*	*34*	55
	55	*33*	*12*	35
	61	*33*	*6*	30
	63	*33*	*4*	28

To investigate the effectiveness of the load-balancing technique, we conducted two additional experiments on the same network using the replicas that were created at UCLA and UTexas. However, we set the client response time threshold at the entry router to be 50 and 30 ms respectively. The link connected to the UCLA router, provided a median response time of 11.5 ms, the link connected to the UTexas router provided 50.5 ms, and the link connected to the Purdue router provided 99.5 ms. Table 8.5 shows that, in the experiment where the quality requirement was set to 50 ms, the entry router first shifted 6% of requests from the link UCSB→Purdue to the link UCSB→UCLA, which reduced the mean response time observed at the entry router from 54 ms to 49 ms. Since the resulting performance was very close to the quality requirement, at a later point, the entry router saw the mean response time (51 ms) exceed the threshold and shifted an additional load fraction to further reduce the response time to 45 ms. From then on, the entry router was able to maintain its performance. Similar behavior was also observed in the other experiment when the quality threshold was set to 30 ms.

Our results verify that the DataSlicer architecture can leverage the detected locality

information to determine appropriate strategies including both load balancing and service replication to satisfy the client QoS requirements. Note that although the specific example we discussed here ended up creating a replica at the UCLA site, other runs created replicas at the other sites as would be expected from our random selection scheme described in Chapter 6. Our results also show that the architecture is robust to network outages and connection failures, and can adapt itself to achieve stable behaviors.

8.3.3 Performance on Large-scale Networks

The purpose of the experiments discussed in this subsection is to understand how well an integration of the techniques described in Chapters 4 – 7 performs against the objective of achieving service performance and scalability in a WAN environment.

Configurations.

The configurations of the large-scale networks we used in our experiments were similar to the configuration of the small-scale network except for the number of the involved nodes: a typical the large-scale network consisted of about 150 PlanetLab nodes, which represented one origin server residing at NYU, 16 service replicas in the United States, 2 replicas in Canada, 4 replicas in Europe, and approximately 140 routers spanning North America and Europe. The constructed oriented overlay network was similar to the one shown in Figure 8.8.

Clients, resident on each of the nodes, requested the service via their co-located router at certain rate, e.g., 5 requests per second. However, if the response time ended up being longer than the inter-request period, the actual rate might be lower. At

Table 8.6: Parameters of experiments running on the large-scale network.

Parameters	Values
#(client)/node	**5**, 10, 20
Request rate	1/s, **2/s**, 4/s
Region center	**Fixed**, **Varied**
Region size	$0.1^\circ \times 0.1^\circ$, $1^\circ \times 1^\circ$, $3^\circ \times 3^\circ$, $5^\circ \times 5^\circ$
QoS threshold	250ms, **500ms**, 1000ms
Timing granularities	**600s/60s**, 1200s/60s, 1800s/60s
Response size	**1KB**, 4KB, 10KB

(a) Parameter setting

Region Values	Overlap & Locality
$0.1^\circ \times 0.1^\circ$, Fixed	Overlap, **high** locality
$0.1^\circ \times 0.1^\circ$, Varied	Partial overlap, **high** locality
$1^\circ \times 1^\circ$, Varied	Partial overlap, **medium** locality
$3^\circ \times 3^\circ$, Varied	Partial overlap, **low** locality
$5^\circ \times 5^\circ$, Varied	Partial overlap, **very low** locality

(b) Region values vs. Implicated locality

startup, each client selected a rectangular region (the size of this region was determined by a passed parameter) on the earth's surface with the center point sitting at the client's own geographic coordinates. The client then randomly requested a small map within the chosen region. By varying the size of this rectangular region, we were able to model workloads that exhibit either low or high locality (larger regions correspond to individual requests that are more spread out, hence exhibit less locality). Note that this workload is capable of exhibiting geographical proximity-based request locality at both the service data and network levels.

Table 8.6(a) shows the parameters we used in our experiment configurations. Similar to the table in Figure 8.2, Table 8.6(b) shows the degree of locality in client workload using the combination of the region parameters. Notice that even when the region

center parameter is set to *varied* and the size of requesting region for clients is very small, there still exists partially overlap among client requesting regions since some of the PlanetLab nodes physically reside at the same locations (as opposed to "no overlap" when α is set to a large value in Figure 8.2).

Each experiment started with clean replicas (no pre-existing replicas) and lasted approximately 2 hours. In the rest of this section, we focus on discussing a few illustrative experiment configurations shown highlighted in bold in Table 8.6.

System Robustness.

The DataSlicer architecture continued running and behaved as expected during experiments running over extended periods on the large-scale network. This stable behavior happened despite changes in network metrics such as latency and bandwidth (too frequent to list), node outages (on average, 37 nodes were found inaccessible at least once in a 2-hour period) and application terminations by the PlanetLab software (as example, in one of the experiments, 37 instances of terminated applications were found on 19 different nodes in a 2-hour period). In each experiment, DataSlicer responded to these situations by updating parent assignments in its overlay network (on average, about 640 times in a 2-hour period), per-router link re-balancing decisions (too frequent to list) and replica creations (ranged from 3 to 10 replicas in the different experiments).

Meeting objectives w.r.t. client-perceived response time.

To investigate whether end-clients in fact see benefits from our infrastructure, we measured the mean response times observed by clients over 60-second intervals, and

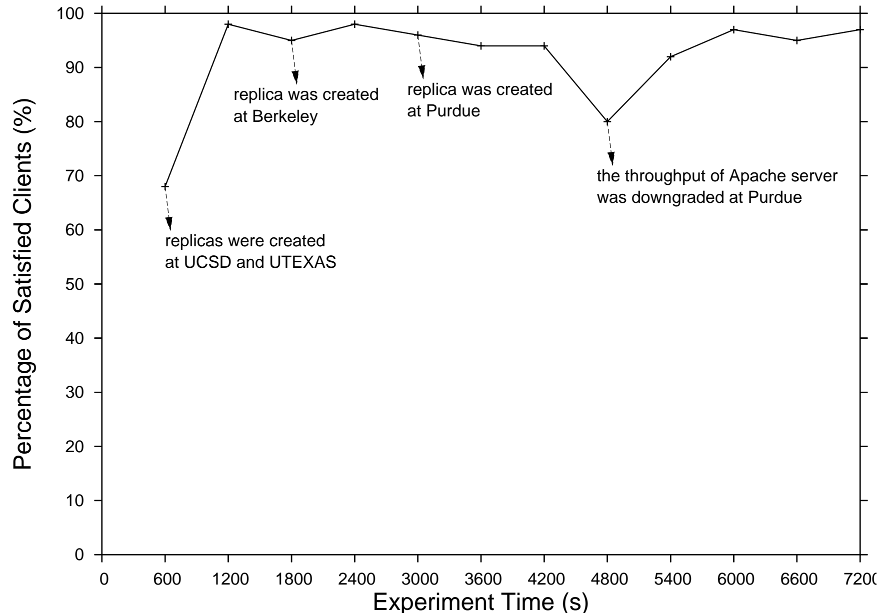


Figure 8.12: Percentage of satisfaction of clients vs. Experiment time

compared the statistical median of these times over a 600-second timing granularity with the desired quality thresholds. In one of our representative configurations, where the number of clients per node was 5 and the request rate for each client was 2/s, our results show that about 32% of clients were unsatisfied at the end of the first 600 seconds. Most of these clients resided in the west coast of the U.S. and Europe. To satisfy these clients, DataSlicer created replicas as necessary to satisfy up to 98% of clients by 1200s, the time the next measurement was recorded (see Figure 8.12), and continued doing so as required to ensure that more than 90% of the clients saw satisfactory performance. Notice that the percentage of satisfied clients drops down to 80% at time 4800s, largely due to a sudden decrease of throughput from ∼150/s to ∼40/s on the Purdue replica due to resource sharing in the PlanetLab network.

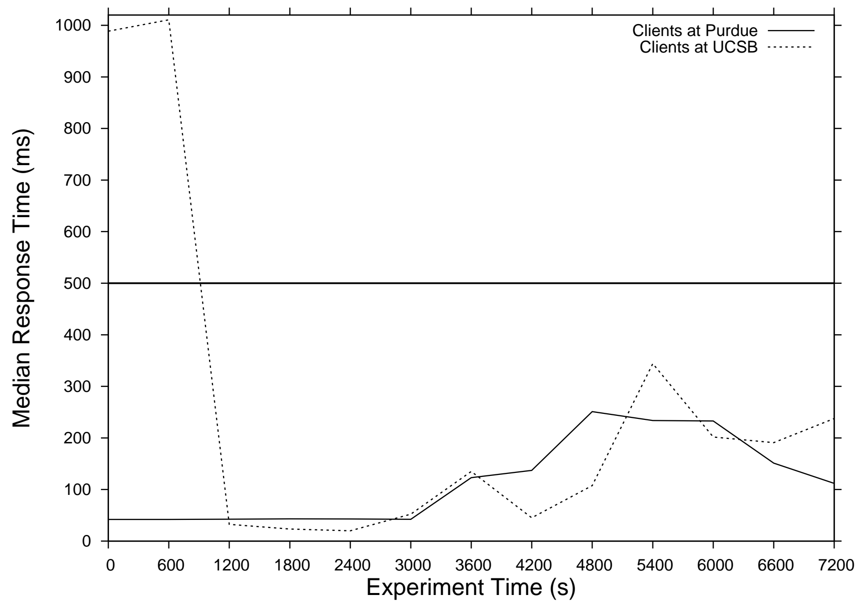


Figure 8.13: Statistical median response times observed by clients at Purdue and UCSB.

DataSlicer was able to re-balance the traffic on the affected routers to increase the percentage of satisfied clients back to its original levels.

Figure 8.13 shows the median response times seen by clients at the Purdue and UCSB routers over the experiment lifetime. For clients at UCSB, the median value dropped after time 600s because of the replica creation at UCSD. It bumped back up at time 1800s due to an update in parent selection occurring at the UCSB router, resulting in traffic re-balancing among the links. The figure also shows that clients at Purdue observed an increased median response time after time 3000s, due to an increasing amount of traffic being redirected to this node because of the replica created there.

“Goodness” of replica creation.

To evaluate the “goodness” of replica creation, we look more closely at the number of created replicas and the nature of the sub-networks benefiting from each created replica.

First, we investigate how DataSlicer responds to spatial locality in the workload by considering an extreme situation where all of the clients request the same region, resulting in only one cell being accessed. DataSlicer responds by replicating this cell at 4 replica sites at UTexas, UCSD, Berkeley, and Purdue. Our results show that with the first 2 replica sites created at UTexas and UCSD, most of the clients were satisfied. However, because of dynamic changes of network characteristics and overload on the created replicas, two other replicas were created as necessary at later points. Figure 8.14 shows the subnetworks rooted at each replica site. The replicas at UTexas, UCSD, and Berkeley have 40, 5, and 21 descendants respectively. The replica at Purdue has a large sub-network, comprising of 89 descendants including most of the nodes in Europe.[3] This validates our expectation of clustering requests based on geographical proximity. Additionally, this also explains why the percentage of satisfied clients dropped a little bit after the replica was created at the Purdue site: a large fraction of traffic in the overlay had been redirected to the Purdue replica which had a lower capacity of service throughput, compared with our origin server, and thereby resulted in a few more unsatisfied clients.

[3] There were a very few Zone1 nodes residing between our origin server and the Europe nodes (Zone2). As a consequence, the Zone2 nodes in Europe ended up with selecting their parents who were close to and thereby associated the replica at the Purdue site.

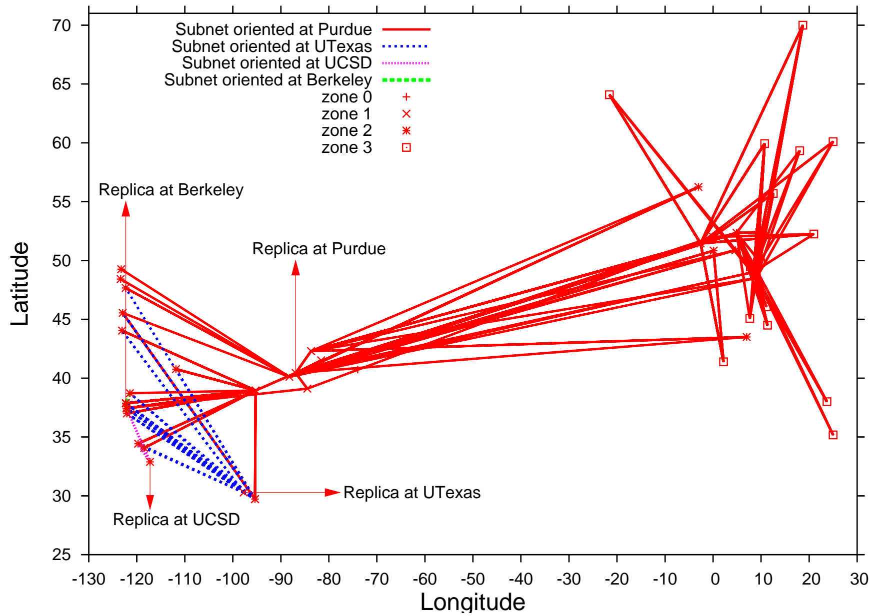


Figure 8.14: Sub-networks rooted at multiple replica sites in a large-scale experiment.

Our next investigation focuses on how DataSlicer responds to more realistic spatial and network locality in the workload. Recall that each client was configured to request a region centered around its own geographical coordinates. Therefore, by choosing different sizes of this region — $0.1^\circ \times 0.1^\circ$, $1^\circ \times 1^\circ$, $3^\circ \times 3^\circ$, and $5^\circ \times 5^\circ$ — we can control the amount of spatial and network-level locality that would be seen. Smaller regions result in clients making more clustered requests, with locality patterns getting detected at network locations that cluster requests from geographically close clients. Larger regions result in more dispersed requests and therefore lower locality. To understand the impact of region size, we discuss as a representative example, the cells replicated at the UTexas replica, which resides in a network region where its descendants are in the west U.S.. 10 cells corresponding to these regions were repli-

cated at UTexas when the size of client request region was $0.1^\circ \times 0.1^\circ$. This number was 9 when the size of the request region was $1^\circ \times 1^\circ$, and increased to 22 when the size of request region was $3^\circ \times 3^\circ$. On the other hand, we found that several cells corresponding to the regions in Europe were created at the Purdue replica, because the replication requests originated from the descendants in Europe.

Since not all of the descendants in a network region could get benefits from a replicated cell at a replica, the numbers of created replicas in these experiments were a little larger than the previous one: 5 replicas were created in the experiment where clients requested a $0.1^\circ \times 0.1^\circ$ region, and 6 in the experiment where the region size was $1^\circ \times 1^\circ$. In both experiments, the percentages of satisfied clients were up to 97%. On the other hand, 6 replicas were created when clients requested a $3^\circ \times 3^\circ$ region, and only one (at Purdue) was created for $5^\circ \times 5^\circ$ region, with both experiments seeing only 62% of satisfied clients. This is because the routers found that client requests were scattered over too many leaf-level cells, with the result that very few routers could observe a cell whose load exceeded our preset threshold (500 requests within a 600-second period) to trigger its replication.

Our results show that the DataSlicer architecture can detect both spatial and network-region locality in the workload and use such locality as a guide to appropriately create replicas and redistribute traffic towards to these replicas. The fact that replicas were created at the Purdue site in response to requests from the Europe nodes suggests that our oriented overlay construction can be enhanced with additional service related information, e.g., geographical coordinates of the participating nodes instead of just network latency measurements.

8.4 Comparison with Other Approaches

To our knowledge, there exist very few systems today that support caching of the dynamic content generated by data-centric network services, so it is not surprising that there are no standard benchmarks that we can use to place the DataSlicer approach in context. Therefore, we compare our architecture against three representative alternatives: (1) a barebones system where clients request the services directly from the origin service without any caching support; (2) a traditional caching system where the clients can cache individual responses, with the cache indexed by the request parameters; and (3) a proxy server system where one or more carefully chosen proxy servers relay traffic between clients and the origin server, detecting locality patterns and replicating service portions at a local repository. The proxy server system can be viewed as a specialization of the DataSlicer network.

8.4.1 Barebones System

In the experiments on the bare system, we used the same set of the PlanetLab nodes before: one origin service server located at NYU, and about 130 client nodes which used to be our routers in the previous experiments (spanning North America and Europe). On each client node, there were 5 clients each of whom sent requests to the origin service every 500 ms, yielding a total number of 650 clients.

Our first experiment was to have each client establish a persistent connection between itself and the origin service for sending requests and receiving responses. All of the clients were invoked simultaneously for sending requests to the origin service. The experiment lasted about 2 and a half hours. The results show that throughout

the experiment, only 150 clients succeeded in sending requests and receiving the responses from the origin service, due to the resource constraints on the origin server: the Apache server can only accept 150 concurrent connections.

To understand how the system behaves in the case that clients use non-persistent connections, we conducted the following experiment. As before, each client still tried to send a request every 500 ms. However, to send a request, the client had to first connect itself to the origin service, and the connection was closed once the response returns. Our results show that about 98.5% of the clients could send requests to and receive responses from the origin services, but with a significant degradation of observed response time on the client side. Figure 8.15 shows the running average of response times observed by a client which resided at NYU (physically close to the origin server). The figure shows that the client observed approximately 4 – 5 ms average response time in the previous experiment where the clients used persistent connections; however, the average response time increased sharply to about 750 ms in the second experiment.

Given a QoS threshold of 500 ms, the first system allows only 23% of the clients to access the service while observing performance within the threshold; the second system allows up to 98.5% of the clients to be able to access the origin service, but none of the clients are satisfied due to the extra delays introduced by competition for the available TCP connections at the origin server.

8.4.2 Traditional Caching System

In this experiment, we implemented a traditional caching mechanism in the client program. Each client maintains a hash table whose entries consist of information

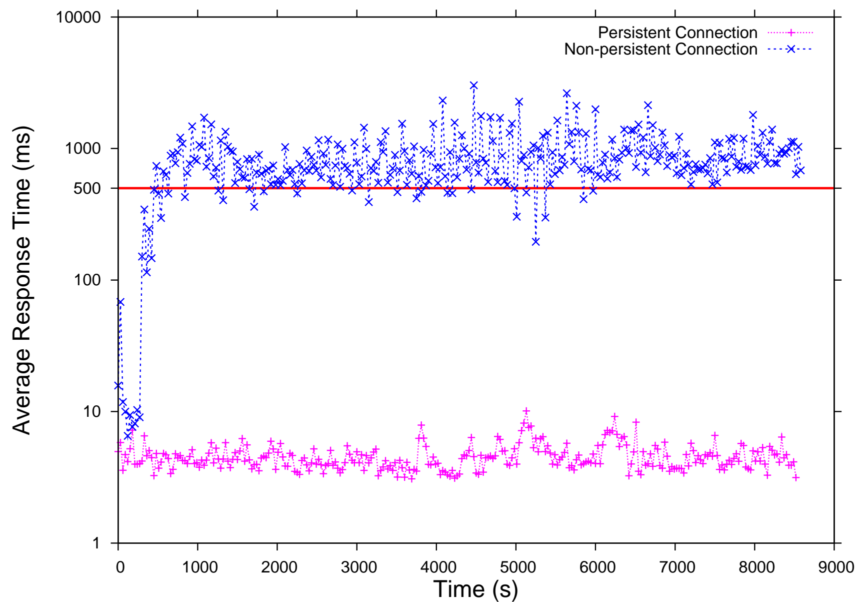


Figure 8.15: The running average of response times observed by a client at NYU in the barebones system.

about the service interface and the parameter values of the requests. Before sending out a request, the client first checks if the request already exists in the hash table. If the request exists, the client simply retrieves the cached content as a response; otherwise, it sends the request to the origin service. The cache is maintained in a way that whenever its capacity is reached, we evict one or more cached requests that are least-recently-accessed.

Such a system achieves good performance only if clients repeatedly request the content that is already in the cache. However, for the synthetic map service used in our experiments where the client randomly requests a map within a region with center point located at the client's location, the possibility to have a cache-hit is very slim. For example, assuming that (1) there are only two parameters used to index the client

requests in the hash table, latitude and longitude of the map center point; (2) all of the latitude and longitude parameters are rounded to the fourth decimal place; and (3) the capacity of hash table is 5000. For a region with a size of $3^\circ \times 3^\circ$, our clients can only have a hit rate of approximately 18$\sim$24 out of 1,000,000; by reducing the size of the region to $1^\circ \times 1^\circ$, the hit rate of the cache increases a little bit to $\sim$80 out of 1,000,000.

Obviously, this system can not really get much benefit giving such extremely low hit rates of the cache. Therefore, the performance of the traditional caching system is similar to the performance of the barebones system.

8.4.3 Proxy Server System

This experiment involved four service replica sites residing at NYU, Purdue, Berkeley and INRIA (France), and 135 client nodes spanning across North America and Europe. Each replica site consisted of one back-end replica node and one or two frontend(s) which acted as a proxy server to relay service requests/responses between clients and service. Each client node selected the nearest replica site to send service requests to. The selection of these proxy servers were based on the geographic distribution of the involved nodes in this experiment such that each replica site could cluster about 30$\sim$40 client nodes that are geographically close. Figure 8.16 shows the formed network: the NYU site clusters 36 clients, the Purdue site clusters 29 clients, the Berkeley site clusters 40 clients, and the INRIA site clusters 30 clients.

Initially, the NYU site served as the origin service and its replica node contained all of the service data, and the other three replica sites did not contain any replication at their back-end replica nodes and had to redirect requests to the NYU site. The proxy

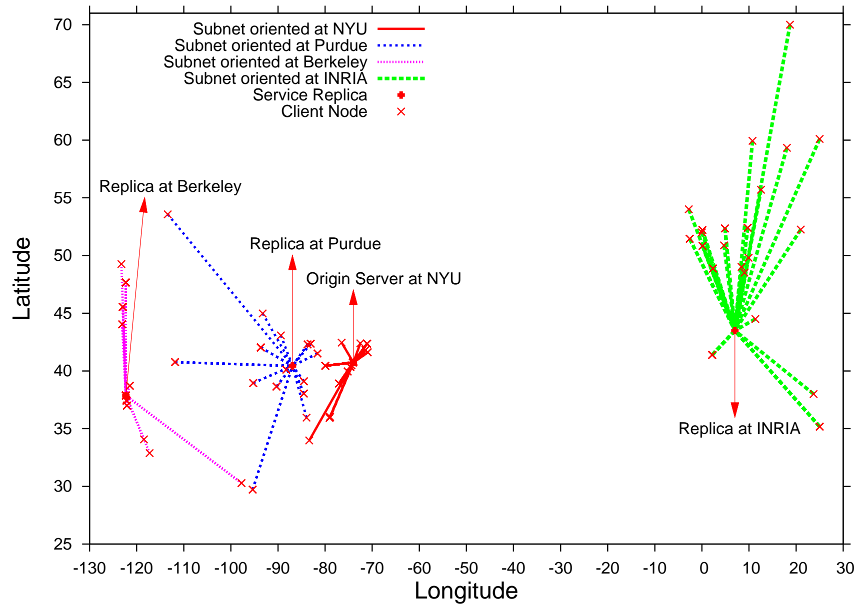


Figure 8.16: The manually configured system using 4 proxy servers.

server implemented our locality detection technique and was able to create replicas at the back-end replica node for data regions seeing poor performance. However, once a region was created, the subsequent requests hit in this region could only be served from the replica node.

The configuration of client nodes and the workload are exactly the same as the ones in our previous experiments on the large scale network: each client node has 5 client programs running; each client program sends out requests at a rate up to 2/s; the client program requests maps in a geographic region with sizes ranging from $0.1^\circ \times 0.1^\circ$, $1^\circ \times 1^\circ$, $3^\circ \times 3^\circ$, to $5^\circ \times 5^\circ$.

The proxy server system, due to the manual careful selection of replica sites, presents an ideal case in clustering client workloads that exhibit similarity stemming

Table 8.7: **Number of replicated cells for different response sizes.**

Response Size	Region Size	DataSlicer	Manually Configured System
1KB	$0.1^\circ \times 0.1^\circ$	13	38
	$1^\circ \times 1^\circ$	21	86
	$3^\circ \times 3^\circ$	21	141
4KB	$0.1^\circ \times 0.1^\circ$	56	30
	$1^\circ \times 1^\circ$	83	71
	$3^\circ \times 3^\circ$	42	93

from network proximity. Consequently, we expect it to provide better location detection ability compared to the DataSlicer approach, which has no apriori knowledge about the workload or client locations and is additionally attempting to cluster requests over a larger set of intermediate routers. Results on the proxy server system validate this expectation: each replica site ends up caching a larger number of regions as compared to a DataSlicer site in large part due to better clustering of requests, hence better detection of exploitable locality (see Table 8.7). The DataSlicer architecture created less replicated cells because (1) traffic was distributed to multiple paths leading to the origin server which reduced the likelihood of an intermediate router finding regions that received more than 500 requests per 600 seconds; and (2) due to the traffic distribution, for regions that did receive more than 500 requests per 600 seconds, in several cases, the intermediate routers could re-balance the traffic to achieve good performance and therefore prevent these regions being replicated.

In terms of client-perceived response time, these additional replications result in the proxy server system delivering marginally better performance over the DataSlicer architecture when the response sizes are relatively small (1KB each). However, even with marginally larger response sizes (4KB), the advantages of DataSlicer's better

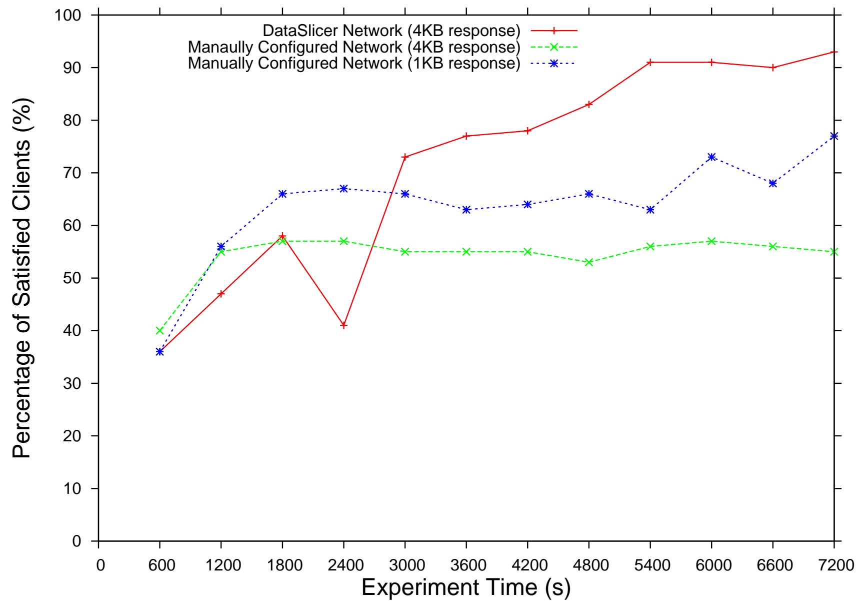


Figure 8.17: Comparison of client satisfaction.

scalability (access to more replica sites) and load redistribution comes through: the proxy sites end up becoming network bottlenecks and limit client performance benefits. Table 8.7 shows with the response size set to be 4KB, the DataSlicer architecture created more replicated cells than the manually configured network for the $0.1^\circ \times 0.1^\circ$ and $1^\circ \times 1^\circ$ region sizes — the situations that present more locality.

Additionally, the proxy server system turns out to be less resilient to faults and misconfigurations. As an example, our choice of the INRIA site was motivated by its network proximity to clients in Europe; however, due to its lower computational power, most of its clients ended up not seeing satisfactory performance resulting in an overall satisfaction percentage of 77% (see Figure 8.17). Similarly, for larger response sizes, the clients in the Berkeley cluster started to see unsatisfactory performance and

further reduced the overall satisfaction percentage to 55%. In contrast, DataSlicer saw up to 97% satisfied clients in the former case, and about 93% satisfaction in the latter.

Our results also reveal that the DataSlicer architecture provides better sustainable load capacity compared with the manually configured system: due to resource sharing on the PlanetLab nodes, the proxy servers failed to maintain all of the concurrent connections from the client nodes (approximately 150 to 200 connections), resulting in some client programs being terminated due to the broken connection. In fact, a significant fraction of the client nodes in the manually configured system achieved about 150 responses every 30 seconds (this contrasts with the 300 requests per 30 seconds we expect from 5 client programs at each node, each of whom sends 2 requests per second). Our architecture, on the other hand, was able to maintain the expected throughput for most of the client nodes in the two cases where sufficient locality was detected.

These results show that the DataSlicer architecture has competitive locality detection ability compared with the ideal manually configured system while providing additional advantages including better load distribution, fault resilience to node failures or misconfiguration, and better system sustainable load.

8.5 Summary

This chapter has evaluated the benefits and the costs of the DataSlicer architecture when used to host data-centric network services in heterogeneous environments. The results show that: (1) the oriented overlay construction technique is able to cluster client requests according to various application-specific metrics which permits the lo-

cality patterns among the underlying traffic to be detected dynamically in the network; (2) the load-balancing and the service replication techniques together reduce redundant creation of service replicas while maintaining the client requirements; (3) the architecture is able to integrate multiple techniques together to provide QoS-assured services with “reasonable” replication cost; and (4) the architecture delivers robust performance in a wide-area network.

Chapter 9

Conclusions and Future Work

This chapter summarizes the work presented in this dissertation and identifies some avenues for future work.

9.1 Summary

The key problem in improving the performance and scalability of data-centric network services in wide-area network environments is how to deal with the dynamic content used in service responses and the massive volume of data in a back-end database.

One solution to this problem is to leverage the existence of locality in service usage patterns to on-demand replicate small service portions representing the locality at appropriate network intermediaries to deliver good performance to end clients.

Realizing benefits from this approach requires answering the following questions: (1) Does locality exist in service usage patterns for data-centric network services? (2) How to dynamically model service locality patterns in a distributed fashion? (3) How

to utilize service locality information (if it exists) to improve performance and scalability for data-centric network services? (4) How to maintain the system robustness in a wide-area network?

This dissertation work has explored the thesis that by identifying the qualitative and quantitative locality characteristics of service access patterns, improving the performance and scalability for data-centric network services in wide-area environments becomes feasible. The feasibility relies on the observations that (1) locality patterns if they exist, can be detected on distributed network intermediaries if the traffic flowing between clients and services can be clustered properly; (2) the volume of service portions representing the locality usage patterns is usually small; (3) proper selection of replica sites to host the replicated data could significantly reduce the replication cost; and (4) client QoS requirements can be satisfied by routing requests along multiple paths.

In order to validate this thesis, this dissertation has described four main techniques: (1) in-network inspection of traffic flowing between clients and services to dynamically infer locality usage patterns, (2) construction of an oriented overlay network to cluster client requests in order to facilitate the locality detection, (3) on-demand initiation of actions such as load-balancing and service replication based on the detected locality information, and (4) exploitation of a variety of mechanisms to maintain system robustness in wide-area network environments. In addition, this work has described how the four techniques are integrated together in the DataSlicer architecture, in order to achieve the goal described above.

9.2 Conclusions

This work has presented a set of techniques that can improve the performance and scalability for data-centric network services in wide-area network environments. The main contributions of this work are:

- Investigation of the existence of locality in service usage patterns for data-centric network services: the accesses of service data exhibit a high degree of locality across multiple dimensions, such as data space, network regions and time epochs, which imply that it is possible to replicate small service portions at a small number of network locations, so as to satisfy a large fraction of client QoS requirements.

- In-network inspection of underlying traffic to dynamically infer locality patterns on the distributed network intermediaries.

- Construction of oriented overlay networks so as to cluster client requests at varied network intermediaries according to some application-specific metrics in order to facilitate the locality detection.

- Cooperation of load-balancing and service replication techniques to reduce the cost of replicating data in wide-area network environments while maintaining the client QoS requirements.

- A set of mechanisms to maintain system robustness in face of various faults including network outages, resource competition, and fluctuations in measurement of network metrics.

In conclusion, the DataSlicer architecture manifests the desired behavior of an alternative caching infrastructure, which dynamically detects locality in the service usage patterns, maintains the service performance and scalability by initiating on-demand actions such as service replication, request redirection and admission control, and adapts itself to maintain system stability in wide-area network environments.

9.3 Future Work

The DataSlicer architecture addresses a subset of the issues raised when improving the performance and scalability for data-centric network services in wide-area network environments: characterizing the locality patterns in accesses of service data, dynamically detecting locality patterns in a distributed fashion, efficiently constructing oriented overlay network to facilitate locality detection, coordinating load-balancing and service replication techniques to reduce replication cost while maintaining client QoS requirements, and maintaining system robustness in wide-area network environments. However, there are many other challenges that remain.

Validating additional locality patterns. Our work characterizes the locality patterns in accesses of service data in terms of *Temporary Locality*, *Spatial Locality* and *Network Locality*, and validates the existence of such locality by investigating the webtraces from two well-known imagery services. However, there also exist other kinds of locality in service usage patterns, e.g., requests that originate from clients who use similar types of devices or have similar network bandwidth requirements may share commonalities. Although we have advocated a general oriented overlay network con-

struction scheme which is capable of clustering client requests according to some application-specific metrics, we don't have quantitative analysis of these kinds of locality due to the difficulty in gaining access to the webtraces of services that could demonstrate such locality in their usage patterns. Additional analysis of this and other form of locality can further improve the ability of our architecture to improve service performance and scalability.

Decoupling the exploited techniques. The primary intended use of the DataSlicer architecture is as a service-neutral platform to host a variety of network services in wide-area network environments. Therefore, DataSlicer integrates four main techniques together to improve service performance and scalability. However, there might be benefits in decoupling these four techniques and applying individual techniques on some specific problem scenario, e.g., one might only want to leverage the locality detection technique to identify certain service usage patterns and use this information to manually determine a static replication strategy. Ideally, we would like to extend our architecture to allow modularization of the involved functional components, i.e., users can customize the architecture by indicating which parts of our architecture they would like to use and how to use these parts.

Exploiting additional service structure. DataSlicer works with the notion of a logical "view" of a data space to model service usage, assuming that the details of the back-end database may either not exist or are unlikely to be exposed. In cases where the service owner would like to expose such information, the service replication algorithm can be extended to come up with a replication solution for regions of the backend

database as opposed to the materialized view [119] embodied in the responses.

End-to-end security. We have assumed a trust relationship between the service owner and the DataSlicer architecture. This manifests itself, among other places, in the fact that in permitting inspection of service request messages, we have assumed that messages are either not end-to-end encrypted, or when they are, the service permits their decryption at the intermediate sites. When the router is only partially trusted by the service, we can relax this assumption by requiring that only a portion of the message content be made public (similar to the notion of *message properties* in BPEL4WS). As long as these properties suffice to associate a request with the service's data space, the benefits of the infrastructure can be made available while still protecting sensitive information.

Bibliography

[1] Microsoft MapPoint Web Service. http://www.microsoft.com/mappoint/webservice/default.mspx.

[2] TerraServer Web Services. http://terraservice.net/webservices.aspx.

[3] Sloan Digital Sky Survey / SkyServer. http://skyserver.sdss.org/.

[4] Windows Live Home. http://www.live.com/.

[5] Google Web APIs Home. http://www.google.com/apis/.

[6] Amazon Web Service Home. http://www.amazon.com/gp/aws/landing.html.

[7] FermiLab Home. http://www.sdss.org/members/fermi.html.

[8] P. Selvridge, B. Chaparro, and G. Bender. The World Wide Wait: Effects of delays on user performance. *International Journal of Industrial Ergonomics*, 29(1), 2001.

[9] Zona Research Inc. The Need for Speed II. http://www.keynote.com/downloads/Zona_Need_For_Speed.pdf, 2001.

[10] T. Wilson. E-biz Bucks Lost under SSL Strain. http://www.internetwk.com/lead/lead052099.htm, 2002.

[11] Zona Research Inc. The Economic Impacts of Unacceptable Web-Site Download Speeds. http://www.webperf.net/info/wp_downloadspeed.pdf, 1999.

[12] R. Fielding et al. Hypertext Transfer Protocol. http://www.w3.org/Protocols/rfc2616/rfc2616.html, 1999.

[13] Extensible Markup Language (XML). http://www.w3.org/XML/, 2003.

[14] Namespaces in XML. http://www.w3.org/TR/REC-xml-names/, 1999.

[15] XML Schemas. http://www.w3.org/XML/Schema, 2001.

[16] XML-RPC Home. http://www.xmlrpc.com/.

[17] D. Box et al. Simple Object Access Protocol (SOAP) 1.2. http://www.w3.org/TR/SOAP/, 2000.

[18] H. Nielsen and S. Thatte. Web Services Routing Protocol (WS-Routing). http://msdn.microsoft.com/library/en-us/dnglobspec/html/ws-routing.asp, 2001.

[19] H. Nielsen, E. Christensen, S. Lucco, and David Levin. Web Services Referral Protocol (WS-Referral). http://msdn.microsoft.com/library/en-us/dnglobspec/html/ws-referral.asp, 2001.

[20] Web Services Activity. http://www.w3.org/2002/ws/, 2002.

[21] E.Christensen et al. Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl, 2001.

[22] Universal Description, Discovery and Integration (UDDI). http://www.uddi.org/specification.html, 2003.

[23] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3), 2000.

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

[25] T. Zhao and V. Karamcheti. Enforcing Resource Sharing Agreements among Distributed Server Clusters. In *Proc. of International Parallel and Distributed Processing Symposium (IPDPS)*, 2002.

[26] W. Rosenberry, D. Kenney, and G. Fisher. *Understanding DCE*. O'Reilly & Associates, Inc., 1992.

[27] W. Rubin et al. *Understanding DCOM*. Prentice Hall, 1999.

[28] Object Management Group. CORBA Security Services, Version 1.8. http://www.omg.org/, 2002.

[29] X. Fu, W. Shi, A. Akkerman, and V. Karamcheti. CANS: Composable, Adaptive Network Services Infrastructure. In *Proc. of the 3rd USENIX Symposium on Internet Technologies and Systems*, 2001.

[30] S. Czerwinski et al. An Architecture for a Secure Service Discovery Service. In *Proc. of Mobile Computing and Networking*, 1999.

[31] I. Foster, C. Kesselman, J. Nick, and Tuecke S. The Physiology of the Grid: An Open Grid Services Architecture for Distributed System Integration. `http://www.globus.org/research/papers.html`, 2002.

[32] A. Ivan, J. Harman, M. Allen, and V. Karamcheti. Partitional Services: A Framework for Seamlessly Adapting Distributed Application to Heterogeneous Environments. In *Proc. of the 11th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, 2002.

[33] E. Freudenthal et al. dRBAC: Distributed Role-based Access Control for Dynamic Coalition Environments. In *Proc. of the 21st International Conference on Distributed Computing Systems (ICDCS)*, 2001.

[34] A. Ivan and V. Karamcheti. Using Views for Customizing Reusable Components in Component-Based Frameworks. In *Proc. of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, 2003.

[35] D. Andresen, T. Yang, V. Holmedahl, and O.H. Ibarra. SWEB: Towards a Scalable World Wide Web Server on Multicomputers. In *Proc. of the 10th International Parallel Processing Symposium*, 1996.

[36] M. Aron, D. Sanders, P. Druschel, and W. Zwaenepoel. Scalable Content-aware Request Distribution in Cluster-based Network Servers. In *Proc. of the USENIX 2000 Annual Technical Conference*, 2000.

[37] O. Aubert and A. Beugnard. Towards a Fine-grained Adaptivity in Web Caches. In *Proc. of the 4th Int'l Web Caching Workshop*, 1999.

[38] G. Barish and K. Obraczka. World Wide Web Caching: Trends and Techniques. *IEEE Communication*, 2000.

[39] H. Bryhni, E. Klovning, and O. Kure. A Comparison of Load Balancing Techniques for Scalable Web Servers. *IEEE Networks*, 2000.

[40] Coral: The NYU Distribution Network. `http://www.scs.cs.nyu.edu/coral/overview.html`.

[41] L. Degenaro, A. Iyengar, I. Lipkind, and I. Rouvellou. A Middleware System which Intelligently Caches Query Results. In *Proc. of Middleware Conference*, 2000.

[42] A. Datta et al. A Comparative Study of Alternative Middle Tier Caching Solutions to Support Dynamica Web Content Acceleration. In *Proc. of the 27th International Conference on Very Large Data Bases (VLDB)*, 2001.

[43] Q. Luo and J. F. Maughton. Form-based Proxy Caching for Database-backed Web Sites. In *Proc. of the 27th International Conference on Very Large Data Bases (VLDB)*, 2001.

[44] W. Shi and V. Karamcheti. CONCA: An Architecture for Consistent Nomadic Content Access. In *Proc. of Workshop on Cache, Coherence, and Consistency(WC3)*, 2001.

[45] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan. DBProxy: A dynamic data cache for Web applications. In *Proc. of the IEEE International Conference on Data Engineering (ICDE)*, 2003.

[46] J. Morris et al. Andrew: A Distributed Personal Computing Environment. *Communications of ACM*, 29.

[47] T. Loukopoulos, I. Ahmad, and D. Papadias. An Overview of Data Replication on the Internet. In *Proc. of the International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN)*, 2002.

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

[49] R. Tewari, M. Dahlin, H. M. Vin, and J. S. Kay. Design Considerations for Distributed Caching on the Internet. In *Proc. of the 19th International Conference on Distributed Computing Systems (ICDCS)*, 1997.

[50] J. Wang. A Survey of Web Caching Schemes for the Internet. *ACM Computer Communication Review*, 29(5), 1999.

[51] A. Chankhunthod et al. A Hierarchical Internet Object Cache. 1996.

[52] S. Michel et al. Adaptive Web Caching: Towards a New Caching Architecture. 1998.

[53] J. Yang, W. Wang, R. Muntz, and J. Wang. Access Driven Web Caching. Technical Report #990007, UCLA, 1999.

[54] D. Wessels and K. Claffy. Internet Cache Protocol (ICP) version 2, RFC 2186. `http://icp.ircache.net/rfc2186.txt`, 1997.

[55] V. Valloppillil and K. Ross. Cache Array Routing Protocol (CARP) v1.0, Internet Draft (draft-vinod-carp-v1-03.txt). `http://icp.ircache.net/carp.txt`, 1998.

[56] D. Povey and J. Harrison. A Distributed Internet Cache. In *Proc. of the 20th Australian Computer Science Conference*, 1997.

[57] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. *IEEE/ACM Transactions on Networking (TON)*, 8(3), 2000.

[58] L. Qiu, V. N. Padmanabhan, and G. M. Voelker. On the Placement of Web Server Replicas. In *Proc. of the IEEE Conference on Computer Communications (INFOCOM)*, 2001.

[59] M. R. Korupolu and M. Dahlin. Coordinated Placement and Replacement for Large-Scale Distributed Caches. *IEEE Transactions on Knowledge and Data Engineering*, 14(6), 2002.

[60] A. Wolman et al. On the scale and performance of Cooperative Web Proxy Caching. In *Proc. of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, 1999.

[61] BitTorrent Protocol. http://bitconjurer.org/BitTorrent/index.html.

[62] M. Theimer and M. B. Jones. Overlook: Scalable Name Service on an Overlay Network. In *Proc. of the 22nd International Conference on Distributed Computing Systems (ICDCS)*, 2002.

[63] M. Freedman and D. Mazières. Sloppy Hashing and Self-Organizing Clusters. In *Proc. of the 2nd International Workshop on Peer-to-Peer Systems*, 2003.

[64] F. Douglis, A. Haro, and M. Rabinovich. HPP:HTML Macro-Preprocessing to Support Dynamic Document Caching. In *Proc. of the 1st USENIX Symposium on Internet Technologies and Systems (USITS)*, 1997.

[65] J. Challenger, A. Iyengar, K. Witting, C. Ferstat, and P. Reed. A publishing system for efficiently creating dynamic web content. In *Proc. of the IEEE Conference on Computer Communications (INFOCOM)*, 2000.

[66] P. Cao, J. Zhang, and K. Beach. Active Cache: Caching Dynamic Contents on the Web. In *Proc. of Middleware Conference*, 1998.

[67] R. Caceres et al. Web proxy caching: The devil is in the details. In *Proc. of ACM SIGMETRICS Internet Server Performance Workshop*, 1998.

[68] Websphere Edge Server. http://www.ibm.com/software/webservers/edgeserver/.

[69] EdgeSuite Services. http://www.akamai.com/html/en/sv/edgesuite_over.html.

[70] K. Candan, W. Li, Q. Luo, W. Hsiung, and D. Agrawal. Enabling Dynamic Content Caching for DatabaseDriven Web Sites. In *Proc. of ACM SIGMOD*, 2001.

[71] S. Dar, M. Franklin, B. Jonsson, D. Srivastava, and M. Tan. Semantic Data Caching and Replacement. In *Proc. of the 22nd International Conference on Very Large Data Bases (VLDB)*, 1996.

[72] B. Chidlovskii, C. Roncancio, and M. Schneider. Semantic Cache Mechanism for Heterogeneous Web Querying. *Computer Networks*, 31(11–16), 1999.

[73] Y. Ishikawa and H. Kitagawa. A Semantic Caching Method Based On Linear Constraints. In *Proc. of International Symposium on Database Applications in Non-Traditional Environments (DANTE'99)*, 1999.

[74] D. Lee and W. Chu. Semantic Caching via Query Matching for Web Sources. In *Proc. of the 8th ACM International Conference on Information and Knowledge Management (CIKM)*, 1999.

[75] B. Chidlovskii and U. B. Borghoff. Semantic Caching of Web Queries. *The Very Large Data Bases (VLDB) Journal*, 9(1), 2000.

[76] A. Szalay et al. The SDSS DR1 SkyServer: Public Access to a Terabyte of Astronomical Data. `http://skyserver.sdss.org/dr2/en/skyserver/paper/`, 2001.

[77] The Universal Transverse Mercator projection and grid system. `http://www.maptools.com/UsingUTM/`.

[78] R. Nauss. The 0-1 knapsack problem with multiple choice constraints. *European Journal of Operational Research*, 2, 1978.

[79] J. E. Pitkow. Summary of WWW characterizations. *Computer Networks and ISDN Systems*, 30(1–7), 1998.

[80] J. C. Moful. Hinted caching in the web. In *Proc. of the 7th workshop on ACM SIGOPS European workshop: Systems support for worldwide applications*, 1996.

[81] V. Almeida, A. Bestavros, M. Crovella, and A. de Oliveria. Characterizing reference locality in the www. In *Proc. of the IEEE Conference on Parallel and Distributed Information Systems (PDIS)*, 1996.

[82] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. In *Proc. of the IEEE Conference on Computer Communications (INFOCOM)*, 1999.

[83] A. Mahanti, D. L. Eager, and C. L. Williamson. Temporal Locality and its Impact on Web Proxy Cache Performance. *Performance Evaluation*, 42(2–3), 2000.

[84] M. Busari and C. L. Williamson. On the sensitivity of web proxy cache performance to workload characteristics. In *Proc. of the IEEE Conference on Computer Communications (INFOCOM)*, 2001.

[85] Q. Wang, D. Makaroff, H. K. Edwards, and R. Thompson. Workload characterization for an e-commerce web site. In *Proc. of the 2003 conf. of the Centre for Advanced Studies conf. on Collaborative research*, 2003.

[86] C. E. Wills and M. Mikhailov. Studying the impact of more complete server information on web. In *Proc. of the 5th International Conference on Web Content Caching and Distribution (WCW)*, 2000.

[87] J. Challenger, A. Iyengar, and P. Dantzig. A scalable system for consistently caching dynamic web data. In *Proc. of the IEEE Conference on Computer Communications (INFOCOM)*, 1999.

[88] W. Shi, R. Wright, E. Collins, and V. Karamcheti. Workload characterization of a personalized web site - and its implications for dynamic content caching. In *Proc. of the 7th International Conference on Web Content Caching and Distribution (WCW)*, 2002.

[89] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report TR-UCB/CSD-01-1141, University of California, 2001.

[90] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content Addressable Network. In *Proc. of ACM SIGCOMM*, 2001.

[91] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. of ACM SIGCOMM*, 2001.

[92] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, 2001.

[93] Gnutella Home. `http://gnutella.wego.com/`.

[94] Freenet Home. `http://freenet.sourceforge.net/`.

[95] Kazaa Home. `http://www.kazaa.com/`.

[96] Y. Chu, S. G. Rao, and H. Zhang. A Case for End System Multicast. In *Proc. of ACM Sigmetrics*, 2000.

[97] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. W. O'Toole. Overcast: Reliable Multicasting with an Overlay Network. In *Proc. the 4th Symposium on Operating System Design and Implementation (OSDI)*, 2000.

[98] M. Castro, P. Druschel, A-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: High-bandwidth multicast in a cooperative environment. In *Proc. of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[99] L. Garces-Erice, E. W. Biersack, and P. A. Felber. MULTI+: Building Topology-Aware Overlay Multicast Trees. In *Proc. of the 15th International Workshop on Quality of Future Internet Services (QofIS)*, 2004.

[100] B. Krishnamurthy and J. Wang. On Network-aware Clustering of Web Clients. In *Proc. of ACM SIGCOMM*, 2000.

[101] B. Krishnamurthy and J. Wang. Topology Modeling via Cluster Graphs. In *Proc. of the 1st ACM SIGCOMM Workshop on Internet Measuremen*, 2001.

[102] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Topologically-aware overlay construction and server selection. In *Proc. of the IEEE Conference on Computer Communications (INFOCOM)*, 2002.

[103] Z. Xu, C. Tang, and Z. Zhang. Building Topology-Aware Overlays Using Global Soft-State. In *Proc. the 23rd International Conference on Distributed Computing Systems (ICDCS)*, 2003.

[104] P. Francis et al. An Architecture for a Global Internet Host Distance Estimation Service. In *Proc. of the IEEE Conference on Computer Communications (INFOCOM)*, 1999.

[105] T. S. Eugene Ng and H. Zhang. Predicting Internet Network Distance with Coordinates-based Approaches. In *Proc. of the IEEE Conference on Computer Communications (INFOCOM)*, 2002.

[106] Y. Chen and R. Katz. On the Placement of Network Monitoring Sites. `http://www.cs.berkeley.edu/yanchen/wnms/`, 2001.

[107] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: a Decentralized Network Coordinate System. In *Proc. of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, 2004.

[108] Cisco Systems Inc. LocalDirector. `http://www.cisco.com`.

[109] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based Scalable Network Services. In *Proc. of the 16th ACM Symposium on Operating System Principles (SOSP)*, 1997.

[110] IBM Corporation. IBM Interactive Network Dispatcher. `http://www.ics.raleigh.ibm.com/ics/isslearn.htm`.

[111] Resonate Inc. Resonate Dispatch. `http://www.resonateinc.com`.

[112] V. S. Pai et al. Locality-Aware Request Distribution in Cluster-based Network Servers. In *Architectural Support for Programming Languages and Operating Systems*, 1998.

[113] V. Pai, L. Peterson, and K. Park. CoDeen: A Content Distribution Network for PlanetLab. `http://codeen.cs.princeton.edu/`.

[114] K. Ranganathan and I. T. Foster. Identifying Dynamic Replication Strategies for a High-Performance Data Grid. In *Proc. of the 2nd International Workshop on Grid Computing*, 2001.

[115] P. Mirchandani and R. Francis. *Discrete Location Theory*. John Wiley and Sons, 1990.

[116] B. Li, M. Golin, G. Italiano, and X. Deng. On the Optimal Placement of Web Proxies in the Internet. In *Proc. of the IEEE Conference on Computer Communications (INFOCOM)*, 2000.

[117] A. Vigneron, L. Gao, M. Golin, G. Italiano, and B. Li. An Algorithm for Finding a k-median in a Directed Tree. *Information Processing Letters*, 74, 2000.

[118] M. R. Korupolu, C. G. Plaxton, and R. Rajaraman. Placement Algorithms for Hierarchical Cooperative Caching. In *Proc. of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1999.

[119] Ashish Gupta and Inderpal S. Mumick. *Materialized Views: Techniques, Implementations, and Applications*. MIT Press, 1998.