Copyright

by

Lei Gao

2005

The Dissertation Committee for Lei Gao

certifies that this is the approved version of the following dissertation:

SAR: Semantic-Aware Replication

Committee:

Mike Dahlin, Supervisor

Lorenzo Alvisi

James C. Browne

Greg Lavender

Arun Iyengar

Harrick M. Vin

SAR: Semantic-Aware Replication

by

Lei Gao, B.S.; M.A.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 2005

Dedicated to my beloved wife,

my caring parents,

and my grandma who always lives in my heart.

Acknowledgments

Thanks to Mike Dahlin, my research advisor, who offered me the opportunity to work in the LASR lab with a group of talented and energetic people. Mike showed me the art and science in research in computer systems. His valuable suggestions and the discussions on research ideas lead me towards better and clearer understanding of my research subject. Further more, those valuable inputs have been the keys for composing this thesis and my other publications.

I must also extend my thanks to Professor Lorenzo Alvisi and colleagues Amol Nayate, Jiandan Zheng, Yin Jian, JP Martine, Jeff Napper, Ramakrishna Kotla, and Ravi Kokku, not only for their cooperative works and helping hands during these school years, but also for the friendly and relaxing working environment created by them.

Last, but not the least, I would like to extend my appreciation to my family and all my friends for their long-term caring and support.

Lei Gao

The University of Texas at Austin

December 2005

SAR: Semantic-Aware Replication

Publication No. _____

Lei Gao, Ph.D.
The University of Texas at Austin, 2005

Supervisor: Mike Dahlin

This dissertation presents a replication framework that facilitates semantic-aware data replication (SAR) in wide area networks (WANs). WAN data replication is fundamentally difficult. As a result, generic replication algorithms must make compromises among *C*onsistency, *A*vailability, *R*esponse time, and *P*artition resilience (CARP) when used in WANs. This dissertation seeks to design algorithms based on specific semantics of the shared data sets (e.g. data properties, workload characteristics, and update patterns) to achieve the optimized CARP trade-offs. Integrating a set of semantic-aware algorithms using distributed objects to form the SAR framework, we implement a practically important e-commerce application, the distributed TPC-W benchmark. Our prototype evaluations show significant improvements on system availability and response time while preserving the consistency guarantees desired by the TPC-W benchmark. The primary focus of the dissertation is on the development of the SAR framework. Within the framework, contributions include (a) exploiting application semantics using the object-oriented approach, (b) employing a hybrid method that integrates a number of novel replication algorithms to make an important class of applications work, (c) proposing a novel replication algorithm for the multi-writer/multi-reader replication scenario with a high access locality, and (d)

outlining a general purpose replication library that uses semantic-aware objects for building other distributed applications in WANs.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In recent years, Internet services have evolved from publishing simple static content to hosting dynamic, interactive applications backed up by large-scale storage systems. This rapid change has revealed limitations of the traditional Internet service architecture and driven the development of new technologies to deliver Internet services to end users with improved response time and availability.

In additional to designing Internet services with good presentation and rich features, service providers need to improve the availability and response time of their services to attract customers. Because the Internet is convenient and fast, it has become a crucial method for delivering to end users business solutions, including online commerce, supply chains management, online health care, and grid systems for scientific research. For instance, purchasing a book online at home (or in the office) is more convenient than driving to the bookstore; integrating the flow of data between customers and suppliers via web-based supply chains can reduce inventory and cost, add product value, extend resources, accelerate time to market, and help retain customers; sharing medical records electronically within the healthcare industry can increase physician efficiencies and reduce costs. Although innovative business concepts are essential to many Internet services, preserving and maximizing the benefits of the Internet, i.e. being convenient and fast, play important

roles in operating Internet services. By the time a class of Internet service becomes mature and standardized, the quality of the service, such as the service availability and response time, will determine the popularity of service providers in the particular type of Internet service.

Because the traditional Internet service architecture requires most Internet services to be delivered to end users through wide area networks (WANs), the service availability and response time are primarily limited by the relatively frequent network failures and volatile network delays in WANs. The abstraction of the traditional Internet architecture consists of a central server (or a server cluster) hosting services at the site of the service provider and a set of clients, normally web browsers, accessing the central server from local machines of end users. An end user accesses the service by submitting requests to the central server via the browser. The information presented to an end user as the result of the user's requests is computed based on the user's requests and the state of the server. The communication between a user and the central server is over WANs that are uncontrolled environments in which network congestion and hardware failures lead to partitions; furthermore, routing through multiple ISPs may causes long and volatile delays. Those WAN characteristics prevent the central server from being highly accessible to end users and introduce a large overhead in the end-to-end response time. Furthermore, because both user requests and the server state are dynamic, the information returned to the end user is difficult to cache by the browser or the user proxy. One cannot simply use traditional web caching techniques to improve both the availability and response time of Internet services on the traditional architecture.

This dissertation offers a solution to effectively replicate dynamic data to improve the availability and performance of Internet services. By leveraging application-specific semantics, our solution, *semantic-aware replication* (SAR), provides the replication support for the *edge services architecture* that distributes business logic (code) and the underlying storage system (data) onto different geographical regions to minimize the communication

in WANs. While other studies have addressed issues for distributing business logic [3, 9, 21, 119, 127] and routing of client queries [2, 16, 40, 119, 133] among distributed server replicas, this dissertation focuses on the replication of storage systems that are shared by distributed server replicas. While preserving the required consistency guarantees for the shared data, our replication solution minimizes the synchronous communication across WAN when accessing an Internet service to enhance the service availability and response time.

1.1 The Evolution of Internet Service Architecture

The ongoing evolution of Internet service architectures aims to enhance the availability and response time of Internet services. In this section, we will first present the basic architecture, the *client-server architecture*, that consists of a central server and a set of clients. Then we will describe two common variations of the client-server architecture, the *cluster-based architecture* and the *client-side cache and proxies*. Each variation intends to improve the client-server architecture on either the client or the server side. Finally, we discuss the *edge service architecture*, an emerging architecture for pushing Internet services closer to end users to maximize the availability and minimize the response time of Internet services.

1.1.1 The client-server architecture

The traditional Internet service architecture consists of a central server and many web clients, as illustrated in Figure 1.1. The central server hosts services at the site of the service provider. A web client usually refers to a web browser running on a user's local machine that provides an interface for the user to access the services offered by the central server. Using a web client, a user accesses the service by sending requests to the central server that generates replies based on the user input and the state of the system. The communication between the central server and the web client is over WANs.

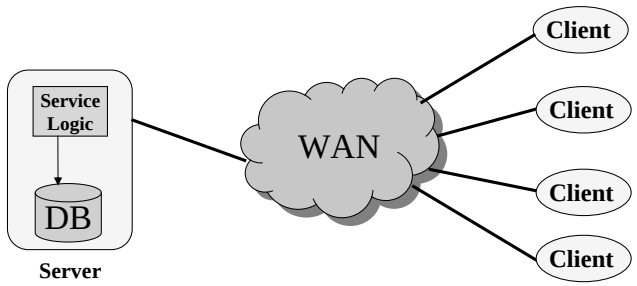


Figure 1.1: The client-server architecture

Although the Internet is a great way for making services (or information) available to end users from a central location, its traditional client-server architecture has many drawbacks. First, the central server is a single point of failure. Second, the Internet is a uncontrolled environment where hardware failures, mis-configuration, and network congestion lead to network partitions. Although the central server is running, users on different network partitions cannot access the service. Third, network delays can be high and volatile, especially when routing across different ISPs. The WAN delay that is up to hundreds of milliseconds dominates the end-to-end response time of the Internet services [28, 66]. Therefore, regardless of the performance improvement brought to the central server in processing user requests, end users will experience little improvement on the service response time.

1.1.2 The cluster-based architecture

Cluster-based techniques eliminate the single point of failure in the client-server architecture by providing server redundancy at the site of the service provider. Some cluster-based architectures [4, 111] use a primary server to actively process requests and other servers in the cluster are backups for the primary servers. When the primary server becomes unavailable, one of the backups becomes the primary server to continuously process requests. Some other cluster-based architectures [17, 79, 83] consider all servers equal and synchronously forward requests to all servers in the cluster. Special protocols are required to maintain the memberships of all active servers. For scalability purposes, cluster-based

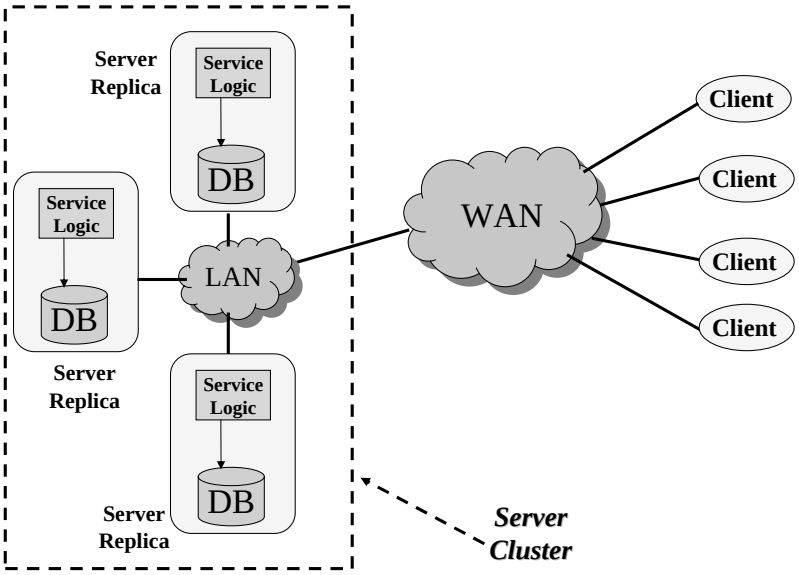


Figure 1.2: The cluster-based architecture

architectures may partition the service and the underlying data sets onto different sets of servers in the cluster. When client requests come in, a load balancer routes the requests to the server(s) of the appropriate service partition based on the nature of requests. Figure 1.2 illustrates the abstraction of the cluster-based architecture.

However, two other drawbacks in the traditional client-server architecture are not addressed by the cluster-based solutions. Because the server cluster runs at one central location, network partitions can still prevent end users of other partitions from accessing the service running on the server cluster. And WAN delays still dominate the end-to-end response time.

1.1.3 Client-side cache and proxies

Figure 1.3 presents the abstraction of the client-server architecture that employs cache techniques to improve the availability and response time of Internet services [23, 47, 52, 64, 75, 57, 132]. There is a reason to be concerned that caching alone will not provide significant improvement because an increasing amount of HTTP traffic is uncachable [38, 129]. The traditional client-side cache or the proxy cache saves web documents from the central server onto local machines. A request to the same document will be satisfied by the locally

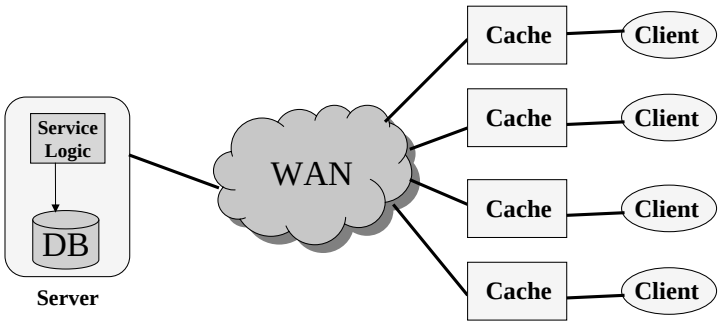


Figure 1.3: The client-server architecture with caching

cached copy. Usually the central server specifies a time-to-live value [81] associated with each document. The client polls the server for changes to the cached document when the time-to-live value expires. Both client-polling [52] and server-pushing [75, 132] techniques can be used to refresh the cached documents.

Traditional caching techniques do not effectively minimize the necessary communication between a client and the central server when dynamic web pages are considered. Although some techniques allow the static segments of web pages to be cached [24, 37], the dynamic segments are always generated and sent by the central server. As long as one WAN trip is required to the central server to process each client request, network partitions and WAN delays are unavoidable.

1.1.4 Edge service architecture

To minimize the effect of network partitions and WAN delays on Internet services, we need a new model that can bring the services near end users. The edge service architecture [2] is an example of the new model. In the edge service architecture presented in Figure 1.4, both the business logic and the underlying storage system are replicated onto server replicas, namely *edge servers*, that are distributed into geographically different regions closer to end users. User requests are directed to the nearest available edge server with routing techniques [2, 16, 40, 119, 133]. Upon receiving a user request, an edge server processes the request by operating on its local data. The edge server communicates with other edge servers and the central server over WANs to propagate any updates to the shared data.

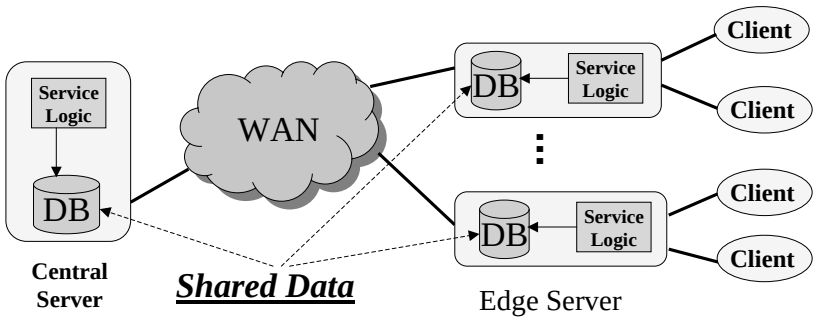


Figure 1.4: The edge service architecture

However, an edge server is not required to fully replicate the central server. Based on application needs, data and services can be partially or fully replicated on edge servers.

Pushing Internet services closer to end users effectively address all three drawbacks in the traditional Internet architecture. Because edge servers are widely distributed, there is no single point of failure. Using appropriate re-routing techniques, the system can direct end users to the next closest edge server when their default edge server becomes unavailable. Furthermore, because users access Internet services through their nearest available edge servers and edge servers can process user requests using their local data, the service is unlikely to be affected when network is partitioned. As long as an end user can connect to his local ISP, he should be able to access the service through the edge server located within that ISP's network although this ISP might have temporary connectivity problems with other ISPs. Finally, network delays within an ISP are much shorter and less volatile compared with delays for packets going across multiple ISPs or across countries.

1.2 Dissertation Focus

The dissertation focuses on providing the replication support for the edge service architecture to deliver Internet services with the optimized CARP (i.e. *C*onsistency, *A*vailability, *R*esponse time and *P*artition resilience) trade-offs. Currently, the challenge in building the edge service architecture is to effectly support the replication of dynamic data among servers. The challenge comes from the theory that no generic replication algorithms can

achieve the optimized CARP trade-off desired for Internet services. In this dissertation, we build a WAN replication library by leveraging on application-specific semantics, which circumvents limits in the general WAN replication.

1.2.1 The challenge in the edge service architecture

The challenge in building the edge service architecture is to effectively maintain the consistency of the shared data operated by all edge servers and the central server. The advantage of the edge service architecture is to have user requests processed by edge servers with their local data with a minimum amount of communication to other servers. It requires both business logic (code) and storage systems (data) to be replicated on those edge servers. Many systems for business logic (code) distribution and execution at edge servers have been built [3, 9, 21, 119, 127], but the core challenge, dynamic data distribution and consistency, still remains.

1.2.2 Challenges in general WAN replication

Although web-scale replication is well understood for traditional web caching where all updates are made at a central server [23, 47, 52, 64, 75, 57, 132], replication and consistency for edge servers that can both read and write data are more challenging. The data on which edge servers operate must be consistently replicated for edge servers to correctly deliver the services. But existing studies [20, 73] have found that generic WAN replication algorithms always have to make trade-offs among CARP.

The CAP dilemma Brewer [20] suggests that there is a fundamental *CAP dilemma* for data replication in large scale systems: systems cannot simultaneously achieve both high **C**onsistency and high **A**vailability if they are subject to network **P**artitions. As a result, distributed code is used for caching and content assembly [2, 21, 119] but seldom used for replication of web services with a rich mix of reads and writes.

The PC principle The study by Lipton and Sandburg [73] shows that there is a trade-off between performance and consistency when data are shared by multiple nodes. The basic principle, the *PC principle*, indicates that the **P**erformance of any sequentially **C**onsistent algorithm is bounded by the minimal packet delay between nodes. In other words, when two nodes are about to update the same shared value, they have to ensure that two update events are observed in the same order by all nodes in the system to preserve a sequentially consistent view of the shared value. Therefore, one node needs to wait for the other to complete its current write before starting the next one.

1.3 Goal and Contributions

1.3.1 Goal

This dissertation seeks to optimize trade-offs among (CARP) consistency, availability, response time, and partition resilience by taking advantage of application semantics. According to the CAP [20] dilemma and the PC [73] principle, application designers are forced to make difficult choices among CARP when relying on generic replication frameworks for distributing/replicating shared data in WANs. Our approach is to provide a SAR library to support dynamic data sharing in WAN by distributed edge servers. Each replication algorithm in the library distributes/replicates some shared data set in a specific replication scenario based on the semantics (e.g. workloads, update topologies, and data properties) of the shared data set. As a result, the individual algorithms can achieve the optimized CARP trade-offs when used to replicate their target data sets.

1.3.2 Contributions

This dissertation makes the following contributions.

1. We exploit application semantics using the object-oriented approach. Although other systems have exploited application-specific semantics in data replication, we are the

first to systematically encapsulate such semantic-aware replication using the object-oriented programming abstraction. Applying the object-oriented concepts to the semantic-aware replication is important because it simplifies the development of distributed application by hiding the complexity of consistency maintenance of the shared data sets. In this dissertation, we implement replication algorithms as a set of distributed objects to encapsulate the algorithm details. One benefit is that we can implement the tunable consistency [135] by restricting the access to the shared data from the application logic using object encapsulation. For instance, we encapsulate the inventory distribution within a distributed object. The object controls the access to the store inventory from the application (business logic) by restricting what the application can ask for. Instead of having the application ask for the total amount of inventory, the object requires the application to ask for the specific amount of inventory desired to process a user request. Accurate information from all edge servers is necessary to compute the total amount of inventory in store while approximate information at the local edge server is usually sufficient to correctly process a user purchase request. The algorithm used to establish the approximate local inventory information among edge servers as the local information changes is hidden from the application. In general, once objects encapsulate the complexity of the consistency maintenance of the shared data, application programmers can focus more on application logic. Another benefit of the object-oriented approach is its extensibility. Because current replication algorithms exist in the system in the form of distributed objects, any optimizations of those algorithms can be easily implemented by inheriting from the existing objects.

2. We employ a hybrid method that integrates a number of novel replication algorithms to make an important class of applications work. Although many of the specific algorithms have been used separately for other applications, bringing them together to build the distributed TPC-W system is a significant contribution because it offers the

system optimized CARP trade-offs that have been proven difficult to achieve using any of the existing algorithms alone. TPC-W is a transaction processing benchmark for the Web that portrays an online e-commerce application. When building the distributed TPC-W application, we observe that this e-commerce application consists of many underlying data sets, each with unique application semantics, i.e. different data properties and workload characteristics. Our replication solution employs a number of existing replication algorithms to separately take advantage of the unique semantics exhibited by individual data sets. As a result, this hybrid method circumvents limits in the general WAN data replication (CAP and PC) and achieves optimized CARP trade-offs desired by the TPC-W application.

3. We propose a novel replication algorithm for the multi-writer/multi-reader replication scenario with a high access locality. Existing replication algorithms do not provide optimized trade-offs among consistency, availability, and response time for the multi-writer/multi-reader replication scenario. The read-one/write-all (ROWA) protocol family (including read-one/write-all-available) offers the optimal read performance, but suffers from poor write availability. Quorum-based approaches yield good overall availability, but cannot achieve the same read availability and performance as offered by the ROWA protocol family. This dissertation proposes the *dual-quorum* replication algorithm that approximates the optimal read availability and performance of the ROWA protocol family while preserving the same availability level as the quorum-based approaches. This algorithm works extremely well under workloads consisting of a large number of consecutive reads or writes to the same node. The dual-quorum algorithm evolves from a TPC-W specific replication scenario, the replication of the user profile.

4. Evolving from TPC-W specific distributed objects, our semantic-aware objects may be used as a general purpose replication library for building other distributed applications in WANs. Although existing objects in the library are designed based on

TPC-W specific requirements, those objects capture the abstractions of many important and representative data replication scenarios commonly seen in a broad range of applications. When building new applications, some or all of the objects in the library can be reused by matching the semantics exhibited by individual shared data sets. For example, the inventory object implements the algorithm for solving resource allocation problems. It can be used in other e-commerce, supply-chain, and ticket reservation systems to allocate identical resources on distributed nodes. The order object implements the abstraction of the single-reader/multi-writer scenario that can be used as an administrative tool in the distributed environment to collect statistical information from multiple nodes. The catalog object implements the abstraction of the multi-reader/single-writer (dissemination) scenario. It can be used for the client-server oriented data replication such as AFS and IBM sports and event information systems. Furthermore, when data sets within applications of other classes exhibit new semantics not captured by the existing replication library, we can introduce additional objects to exploit the new semantics with little change required to existing objects.

1.4 Dissertation Map

In the next chapter, we provide the motivation for the dissertation with a trace-based simulation that quantifies the limits of the traditional Internet service architecture and suggests new architectures that push Internet services closer to end users to improve the end-to-end service availability.

Chapter 3 presents the semantic-aware replication (SAR) framework that allows us to build a practically important e-commerce system with optimized CARP trade-offs. Through our prototype system, a distributed version of the TPC-W benchmark, we show that we can circumvent limits in general WAN replication by leveraging on application-specific semantics. In addition, the framework consisting of a collection of distributed

objects simplifies the system design by applying the object-oriented concept to encapsulate the complexity of the consistency management.

Chapter 4 describes a novel replication protocol, dual-quorum with volume leases (DQVL), that deals with the multi-writer/multi-reader (MWMR) replication scenario with a strong access locality. Because of limits suggested by the CAP dilemma and the PC principle, general replication algorithms that support the MWMR replication scenario have to either sacrifice the consistency level to improve availability and response time or reduce the degrees of availability and response time to gain stronger consistency guarantees. But our experience in building distributed TPC-W system suggests that we can design a new algorithm to optimize those trade-offs for MWMR by leveraging the underlying workload characteristics. DQVL outperforms existing replication protocols in the MWMR replication scenario when the workload exhibits a strong access (both read and write) locality. We describe the detailed design and the evaluation of DQVL in this Chapter.

In Chapter 5, we outline our vision for the unified replication architecture. The unified architecture merges our semantic-aware replication design with a set of powerful replication primitives, namely the *PRACTI* replication mechanisms. We describe the architectural design of the unified replication system and the associated benefits in this Chapter.

In Chapter 6, we describe other related work. And we discuss future research directions in Chapter 7. Chapter 8 concludes the dissertation.

Chapter 2

Coping with Internet Failures - the model

In Chapter 1, we briefly described the evolution of the Internet service architecture that ultimately aims to enhance the availability and response time of Internet services. For the emerging edge service infrastructure to live up to its promise, i.e. pushing services closer to end users to improve the quality of Internet services, our semantic-aware replication (SAR) solution is essential because it answers to the challenge in the general WAN replication by leveraging the semantics of the shared data sets.

Before diving into the detailed discussion on the edge service infrastructure and our SAR solution, we devote this chapter to explain the motivation of the SAR work by quantifying limits of the existing Internet architecture using a trace-based simulation study. And because there are extensive studies [22, 28, 41, 67, 66, 69, 80, 126] on analyzing the end-to-end WAN performance and its impact on Internet services, our simulation focuses on the end-to-end *availability* of Internet services in this chapter.

Using the simulator implemented based on the Internet failure model by Dahlin et al. [35], we show that traditional web caching techniques alone cannot cause significant improvement on the end-to-end Internet service availability. But with a more sophisti-

cated infrastructure, like the edge service infrastructure, that combines overlay routing and more aggressive replication techniques (e.g. server replication and selection), can service providers offer revolutionary end-to-end improvements to their Internet services.

2.1 The End-to-end Service Availability

Although several commercial hosting services today advertise 99.99% or 99.999% ("four 9's" or "five 9's") server availability, providing highly available servers is not sufficient for providing highly available service because it is not an end-to-end approach: other types of failures can prevent users from accessing services. Internet connectivity failures, unfortunately, are not rare. Studies measuring the end-to-end Internet availability [94, 138, 63] have reported that service delivery failure rates – even for these presumably carefully-engineered sites – are about 2 to 3 orders of magnitude worse than state-of-the-art end-server availability. In contrast with the 5 minutes per year of unavailability for a five-9's system, a typical two-9's Internet-delivered service will be unavailable to typical clients for nearly 15 minutes per day.

Although caching can improve file system availability [57, 64], there is reason to be concerned that caching alone will not significantly improve WAN service availability because much HTTP traffic is uncachable [38, 129]. This limitation motivates us to study the potential effectiveness of other techniques such as hoarding [64], push-based content distribution [51], relaxed consistency, mobile extensions to ship service code to proxies or clients [7, 21, 61, 117, 119, 133], anycast [15, 40, 133], and overlay routing [97, 110, 98, 105, 54, 139]. Although the performance benefits of many of these techniques have been studied, their potential impact on end-to-end availability has not been quantified.

This chapter seeks to understand how network failures affect the availability of service delivery across wide area networks (WANs) and to evaluate classes of techniques for improving end-to-end service availability. By providing a quantitative analysis of these techniques, we hope to illustrate the effectiveness of the emerging Internet service archi-

tecture that employs a number of such techniques, including server replication and selection [23, 48, 68, 95, 102, 104, 135], replication of active objects [7, 21, 61, 117, 119, 133] and overlay routing [97, 110, 98, 105, 54, 139]. While overlay routing and replication of active objects have been extensively studied in the context of wide area networks, existing server replication techniques are limited when used in the WAN environment (i.e. the edge service environment). We will describe our SAR approach that offers the effective WAN replication support for the edge service architecture in the following chapters.

2.2 Network Failure Model

2.2.1 Model abstraction

Our simulation program is based on the Internet failure model developed by Dahlin et al. [35]. In this model, a service is *available* to a client when that client can communicate with it. A service is *unavailable* to a client when that client cannot communicate with it due to a network or end-host failure. For each client, a service alternates between being available and unavailable. The model terms a period of time when a service is continuously available to a given client an *availability event* and a period of time when a service is continuously unavailable to a given client a *unavailability event*. A service's *average availability* is the fraction of time when a service is available to an average client, and a service's unavailability is the fraction of time when a service is unavailable to an average client. The model also consider *request-average availability* (or unavailability) - the fraction of requests in a data set that succeed (or fail) in accessing a service.

These definitions describe a binary on/off model of availability: if a service is reachable it is available; otherwise it is unavailable. Whereas the model tracks periods of complete disconnection, for some applications, the network has "failed" if the bandwidth falls below a certain level or the latency rises above some level. However, the model is reasonablely adequate considering that our goal is to evaluate the effectiveness of techniques that

can mask network failures atop a reasonable Internet failure model.

2.2.2 Model parameters

The failure model includes parameters of network failures that most directly affect techniques to improve availability. The most basic parameter is failure rate: what fraction of time are two nodes unable to communicate? The model also analyze failure patterns along two dimensions: failure location and failure duration.

Failure duration influences the effectiveness of techniques for coping with failures. For example, it may be simpler to use caching or prefetching to mask short failures than long failures since masking long failures requires predicting access patterns across longer periods of time, transferring more data to the cache, and storing more data in the cache.

Failure location influences the effectiveness of routing-based strategies. Dahlin et al. use a simple model that classifies failures into three operationally significant categories – "near-source," "in-middle," and "near-destination." Near-source failures represent failures of the client stub network that disconnect a source machine or source subnet from the rest of the Internet. Near-destination failures have a similar effect on destinations. In-middle failures represent connectivity failures in the middle of the network that prevent a pair of nodes from contacting one another (on the default route), but where both nodes are able to contact a significant fraction of the remaining nodes on the Internet. They also use the term "stub failures" to refer to the combined near-source and near-destination categories.

The failure model uses a simple model for failure inter-arrival times. Given a failure rate (expressed as a fraction of time a particular class of failures occurs at a particular location) and an average failure duration, the model calculates the average inter-arrival time for each class of failures. The model then assumes that failures arrive independently with exponentially distributed inter-arrival times with the given average arrival rate.

Table 2.1 summarizes key parameters for the model. Since this dissertation makes no claims for the contribution of the failure model, readers can refer to the study by Dahlin

Parameter	Default value	Comment
Rate	1.5% (all failures) 1.25% (> 30s)	Varies from .4% to 7.4% in different data sets
Location	Src: 25% Mid: 50% Dest: 25%	All locations significant. Ratio varies widely across traces.
Duration	avg = 609 sec. pdf(x) = $16x^{-1.85}$	Appears heavy-tailed
Interarrival	avg = 48111 sec.	

Table 2.1: Default parameters for failure model.

et al. [35] for the details on how these parameters are obtained.

2.3 Masking Network Failures

This section studies two classes of techniques for improving end-to-end service availability by masking network failures. Client-independence techniques – such as data caching, prefetching, mobile code, and edge servers – provide a (possibly degraded) version of a service using local resources when the remote server cannot be contacted. Routing and connectivity techniques use alternate network paths to route around failures.

These experiments show that the traditional web cache alone yields limited availability improvements. In the process, we also seek to quantify the potential effectiveness of two classes techniques at improving service availability. In order to provide information about of a broad range of techniques, our experiments abstract away implementation details and thus provide an upper bound on the techniques' effectiveness.

Although this section focuses on service-level techniques for improving availability, researchers will certainly work to improve reliability at the hardware and transport layers as well. Indeed, achieving the goal of four- or five-nine services will likely require advances at all layers. So, in addition to the experiments described above, we assess the sensitivity of our results to changes in the reliability of the underlying infrastructure.

Workload	Date	Clients	Servers	Sessions
Squid-P	3/28/00–4/03/00	1	131193	1557875
Squid-C	3/28/00	107	52526	403235
BU-P	1/17/95–5/17/95	1	4614	56789
BU-C	1/17/95–5/17/95	33	4614	68949

Table 2.2: Web access trace parameters.

2.3.1 Workload and methodology

In addition to the failure model described in the previous section, our simulator uses two sets of web service access traces to represent Internet service access patterns. Table 2.2 summarizes key parameters for these traces. The workload is based on the Bo1 Squid trace, a four-month trace taken at clients at Boston University [33]. This trace is old, but it includes client cache hits, and the client-ID mappings are not changed over the trace period. We examine both traces from the point of view of a proxy shared by all clients in the trace (Squid-P and BU-P) and from the point of view of individual client machines (Squid-C and BU-C) with no shared proxy. Because the Squid traces change the client-ID mappings daily, we only look at the first day of the Squid-C trace.

For our simulations, we post-process the traces to group individual accesses into *sessions*. We define a session as a set of accesses from a client (-C traces) or proxy (-P traces) to a single server in which the maximum gap between successive requests is 60 seconds. Our figure of merit for availability is the fraction of sessions that complete without interruption.

Our simulator tracks the references to objects in the traces and uses trace information to classify the objects as cachable or uncachable and to identify when objects change. It assumes that each simulated client (-C traces) or proxy (-P traces) has an infinite cache that stores all objects accessed previously in the simulation. Otherwise, cache replacement policies, which are not the focus of this work, will affect our measurements.

Because we do not know the details of some client independence techniques, such as the prefetching and mobile code, we use the simulation parameter *install_time* to represent

the amount of time from the first access by a client or proxy to a service until the service has downloaded sufficient state or programs or both to the cache to cope with network disconnections. Our default *install_time* is 100 seconds. During the *install_time*, clients and proxies must access the service from cached data or via the origin server.

If the network remains up during an entire session, the simulator classifies the session as *No Failure*. For sessions in which the network fails, the simulator examines the objects referenced in the session and classifies the session as follows: *Cache Hit* if all requests are for fresh cached web objects; *Stale Hit* if all requests are for cached web objects and if some of those objects require updates from the server; *Hoardable Degraded* if the *install_time* for the service has completed at time of the failure and all requests are for cachable objects but some miss; *Dynamic Degraded* if the *install_time* has completed at the time of the failure but not all session data are cachable; and *Fail* if the *install_time* has not completed at the time of the failure and either some data are not cachable or some data are cache misses. Note that due to limitations of the trace and of the HTTP protocol, the traces may overstate the cachability of data and may underestimate the rate of change of data. Thus our results may understate the Stale Hit and Dynamic Degraded rates and overstate the Cache Hit, Stale Hit, and Hoardable Degraded rates.

2.3.2 Client independence

A range of client independence techniques are available.

1. **Caching.** Caching hides network and server failures by serving requests from a nearby cache rather than a distant server [57]. Most web clients today include some form of caching.

2. **Relaxed consistency and push-updates.** Relaxed consistency can improve availability by allowing caches to serve potentially stale data during failures rather than requiring the cache to use (unavailable) current data. Alternately, under a push-updates protocol [72, 115], servers may update cached copies before clients issue

reads requesting the new versions. Push-updates thus improves the chance that a cache will contain current data during a disconnection.

3. **Prefetching.** Prefetching brings objects close to a client before the client accesses them. **Hoarding**, a form of prefetching in which a user identifies groups of objects to fetch, is effective for disconnected operation in file systems [64], and the Microsoft Internet Explorer browser implements a hoarding option for web pages. **Server push** [51, 65] such as the content distribution networks becoming commercially available can be thought of as a form of server-directed prefetching. Note that prefetching is more aggressive than the "push update" approach described in the previous paragraph. "Push update" only distributes new versions of objects that have already been referenced by a cache, while prefetching can distribute unreferenced objects in order to avoid compulsory misses.

4. **Replication of active objects.** Several researchers have proposed systems in which active service objects may be cached or replicated and then executed [7, 21, 61, 117, 119]. These techniques may provide ways to extend the benefits of caching, relaxed consistency (or "application-specific adaptation" [89]), and prefetching to the significant fraction of web services that are not cachable [38, 129].

This set of experiments examines the potential effectiveness of using these client independence techniques to improve robustness of Internet services by transforming *failed sessions* that are interrupted by network disconnections into *degraded sessions* that are served by the cache or by downloaded mobile extensions. Clearly, the relative advantage of degraded sessions over failed sessions will vary from service to service: some services can provide full service while disconnected, others can provide tolerable service across short disconnections, and still others require continuous on-line communication with a remote site to be effective. To cope with this wide range of service behaviors, this experiment does not attempt to quantify the benefit of degraded service over failed service; instead it

seeks to quantify how often services have the option to use caching, relaxed consistency, prefetching, or mobile extensions to improve their robustness to network disconnections.

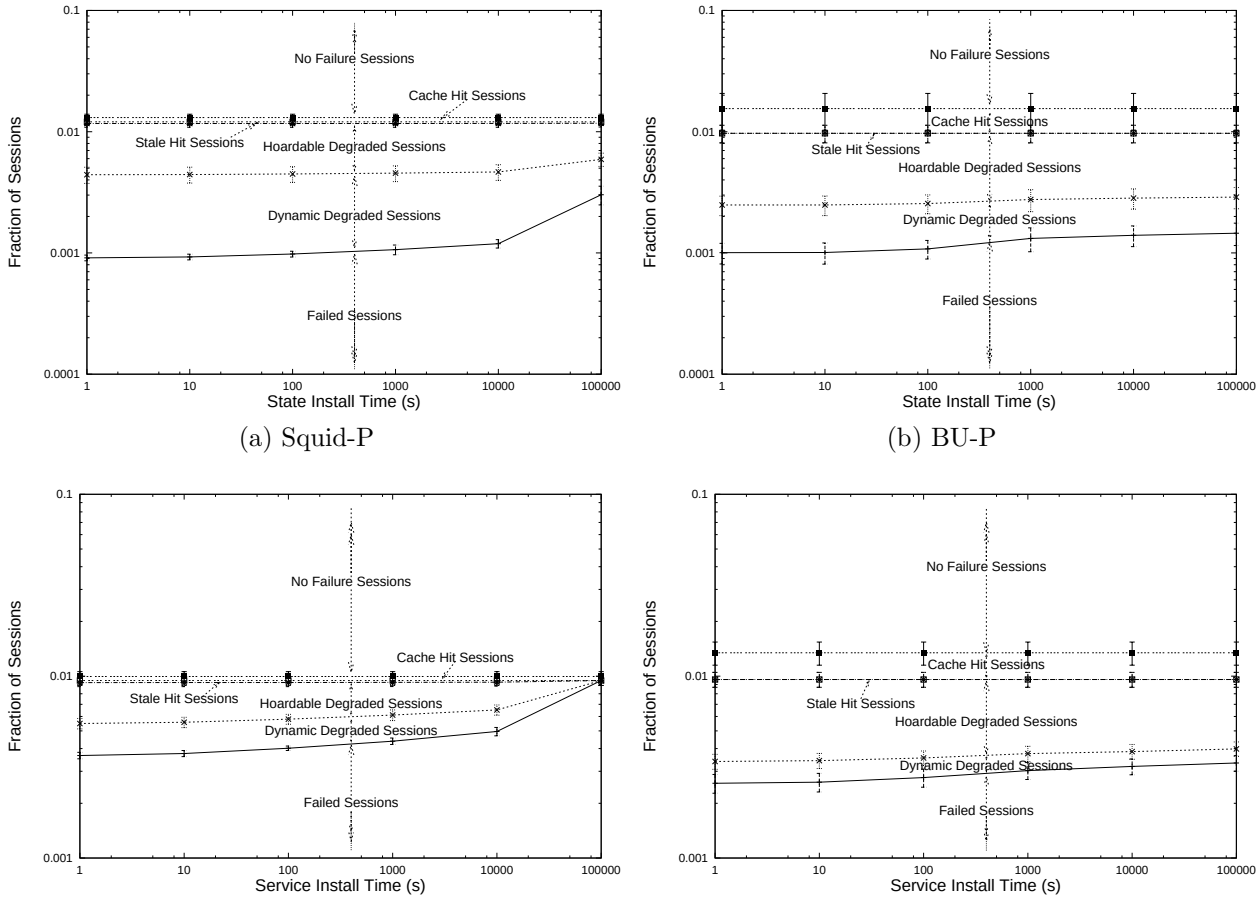
For these experiments, we set the failure-location distribution to make all failures "in-middle" failures, and we conduct five trials with different random seeds for the network failure model and graph the mean and standard deviation of results. We describe improvements to failure rates in terms analogous to the common definition of "speedup" [55]: $improvement = \frac{failureRate_{orig}}{failureRate_{new}}$.

Results. First, we examine the effectiveness of these general techniques as well as the extent that installation time limits improvements. The y-axis of Figure 2.1 shows the fraction of sessions classified in the categories listed above on a logarithmic scale so that equal intervals reflect equal improvements to failure rates. The x-axis shows the *install_time* for each service also using a log scale, and each graph shows these results for a different workload. When installation times are short, the combined effect of all techniques is to improve the failure rate by at most factors of 14.4 (Squid-P), 15.4 (BU-P), 2.7 (Squid-C) and 5.22 (BU-C) for the four workloads compared to the failure rate that would be encountered if each request were sent to the origin server.

The improvements available from caching alone appear small (improvements to failure rates of 1.1, 1.6, 1.1, and 1.4 for caching and of 1.1, 1.6, 1.1, and 1.4 for caching plus relaxed consistency or push-updates). Note that the Squid workload's lower-level caches may hide sessions that only reference cached data, causing us to understate the benefits of caching alone. Conversely, the BU trace are not filtered by caches, but they are old and may reflect a workload that is unrealistically easy to cache. It seems likely that caching's benefits lie between these values.

In contrast with caching alone, aggressive prefetching plus caching may be able to achieve significant improvements for those services where prefetching is feasible; the simulations indicate upper bounds of 3.0, 6.2, 1.8, and 4.0 for this combination.

The only limiting factor to active object replication in this model is our assumption

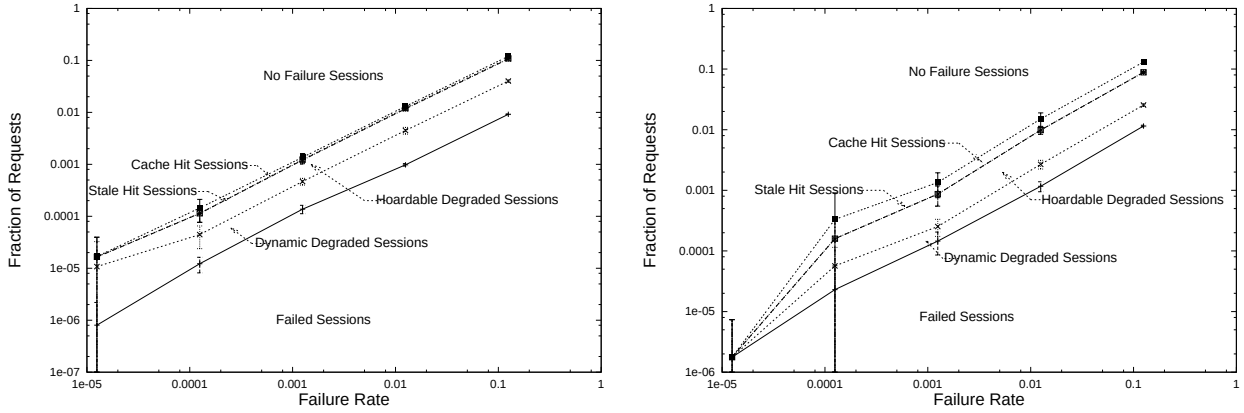


(a) Squid-P (b) BU-P

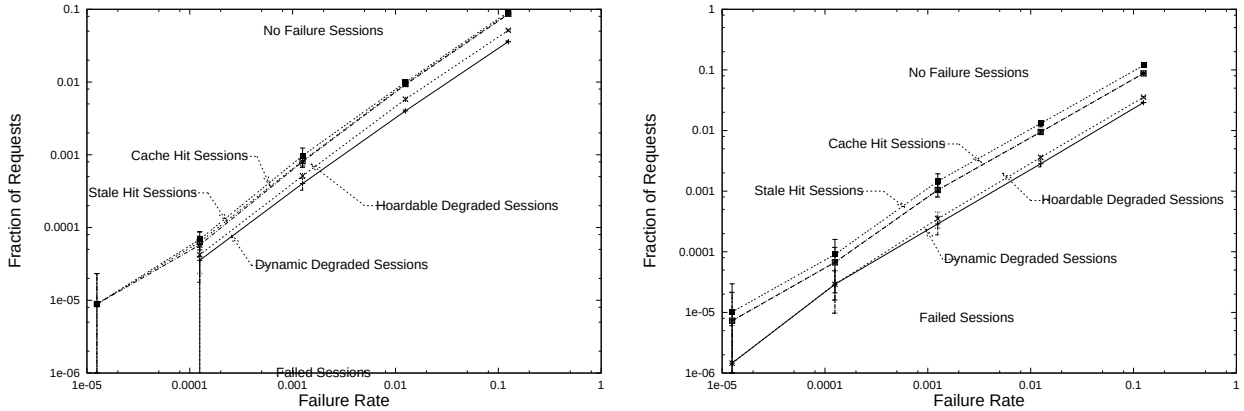
(c) Squid-C (d) BU-C

Figure 2.1: Session result v. state installation time. Each region between two lines represents the fraction of sessions that can be handled by the specified technique plus those above it in the graph.

that each service requires different extension code and data, and that extensions cannot be downloaded until a service is first accessed. Under this assumption, improvements to failure rates are limited to about an order of magnitude for these traces because if the network is down when a service is first accessed or during the first *install_time* of accesses, no code and data is available to mask the failure. These "compulsory misses" also limit the prefetching line in these graphs. If compulsory misses and initialization times are ignored, prefetching could provide improvements in failure rates of up to 3.7, 9.7, 4.7, and 12.2 and



(a) Squid-P (b) BU-P



(c) Squid-C (d) BU-C

Figure 2.2: Session results as network failure rates vary.

replication of active objects and their data could, in principal, provide at least degraded service 100% of the time.

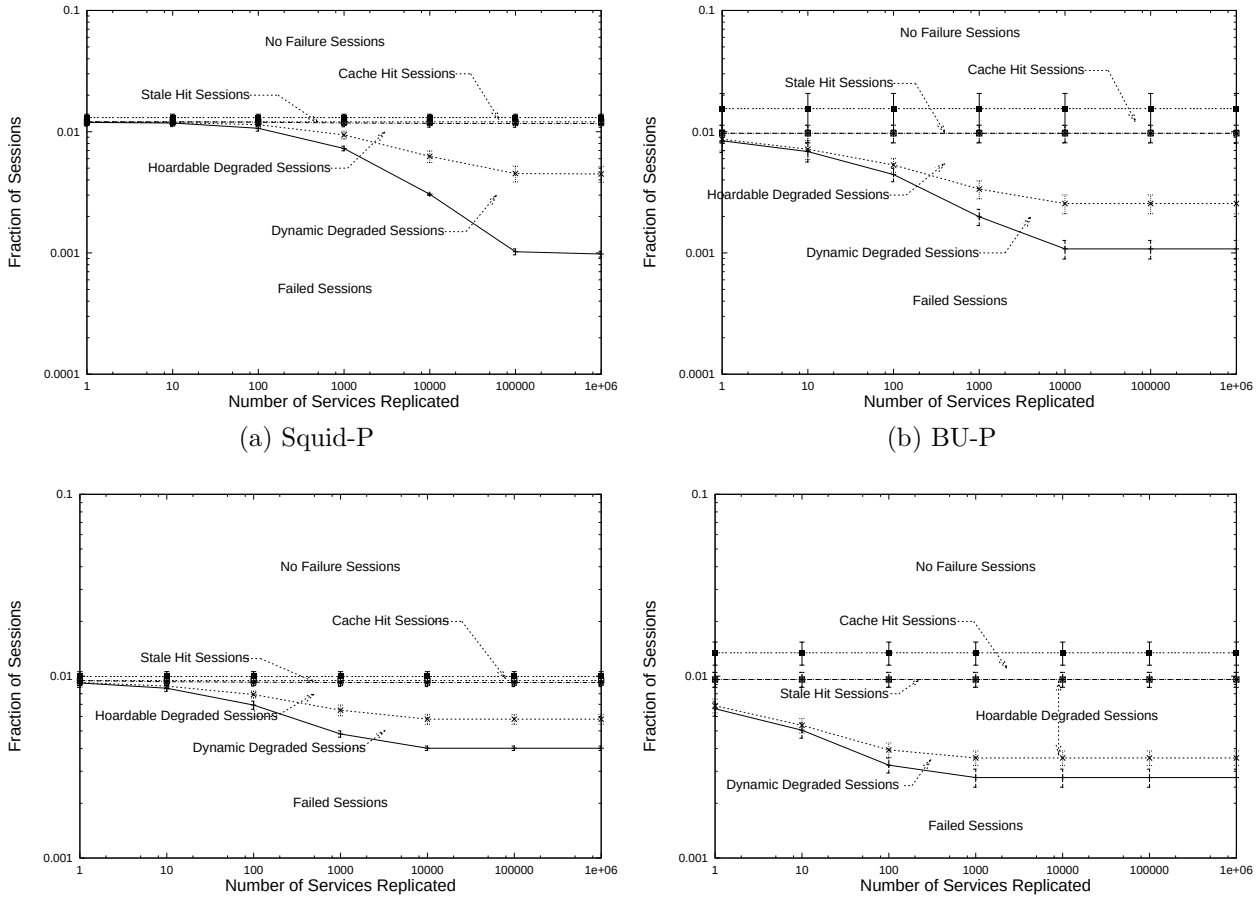
The available benefits fall gradually as installation time increases and compulsory misses become more expensive. At a 10,000 second installation time the upper bound on availability improvements are 11.0, 11.1, 2.0, and 4.2 for the four workloads. This result is promising: it suggests that services that need to download significant amounts of state to provide acceptable disconnected service may have the opportunity to do so.

Next, we examine the sensitivity of our results to the underlying network failure

rate. Figure 2.2 shows session results for both simulated client and proxy using tow traces (Squid and BU) as we vary network failure rates by reducing the time between failures and leaving the failure duration distribution unchanged. All four workloads are qualitatively similar. These data suggest the improvement in session failure rates provided by caching, prefetching, and replicas of active objects are relatively insensitive to the underlying network failure patterns between failure rates of .0125% and 12.5%. At failure rates below that, the traces are so short that relatively few failure events occur, and our results have too much variance to reach definitive conclusions.

The experiments above suggest that to significantly improve overall service availability, services may need to resort to prefetching and mobile extensions rather than relying on caching alone. Unfortunately, these techniques can dramatically increase the demand for resources at a client, proxy, or network. A key limiting factor, therefore, may be how many resources a cache can devote to each hosted service and how many services a cache can simultaneously host. Figure 2.3 shows session results when the simulated client and proxy maintain only a finite number of local copies of prefetched services and mobile extensions with MFU policy to evict the rest (results for LRU replacement and exponentially decaying average MFU are similar but not shown). Graphs (a) and (b) show configurations for proxies shared by all clients in the trace; graphs (c) and (d) show per-client configurations. For all four workloads the results are qualitatively similar, but the cache size needed for full benefits is larger for the Squid-P workload and smaller for the Squid-C and BU-C workloads due to the differing number of services accessed by each of these workloads.

Then, we evaluate the sensitivity of different services to client-independence techniques. In the previous experiments, we generalized all web services to evaluate their service availability improvement after applying those failure masking techniques. It is also necessary to explore the benefit that different kind services would receive from those client-independence techniques. Figure 2.4 shows the cumulative distribution function of availability improvement of services examined. We eliminate services with fewer than five

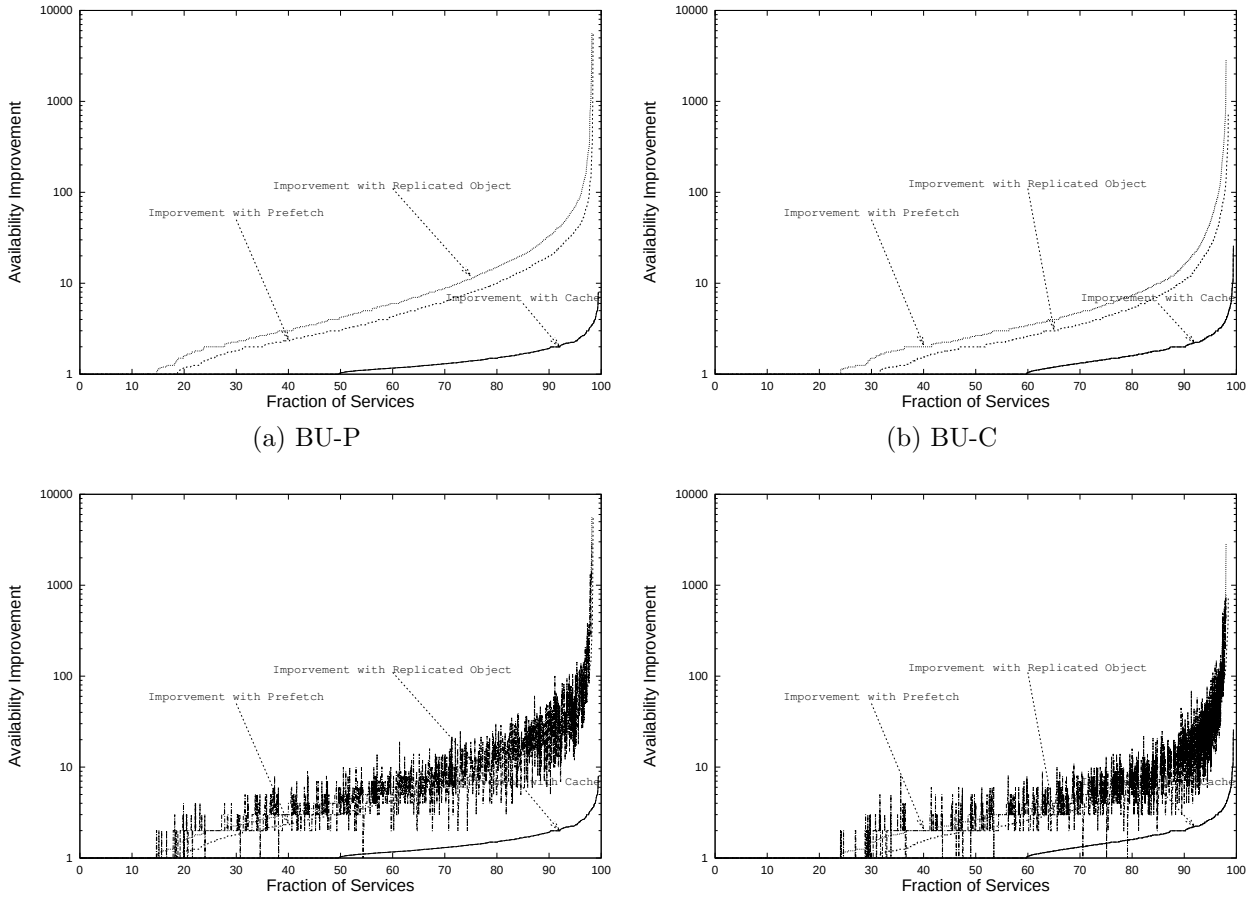


(a) Squid-P (b) BU-P

(c) Squid-C (d) BU-C

Figure 2.3: Session failure rate v. number of cached service extensions.

failed access by clients in out BU trace. The graph shows that about half of services receive some (2-10 times) improvement with prefetching and replicated-object techniques. 15% of services receive improvement in 2 to 3 orders of magnitude. And there are less than one quarter of services benefited little from those client-independence techniques. Notice that in graph (c) and (d) of Figure 2.4 the improvement is proportional to the number of requests made to each service. These vertical lines represent number of requests made to each of examined web services. Failures could only occur to a particular client or proxy on its first request to a particular service because the extension requires an install time. After



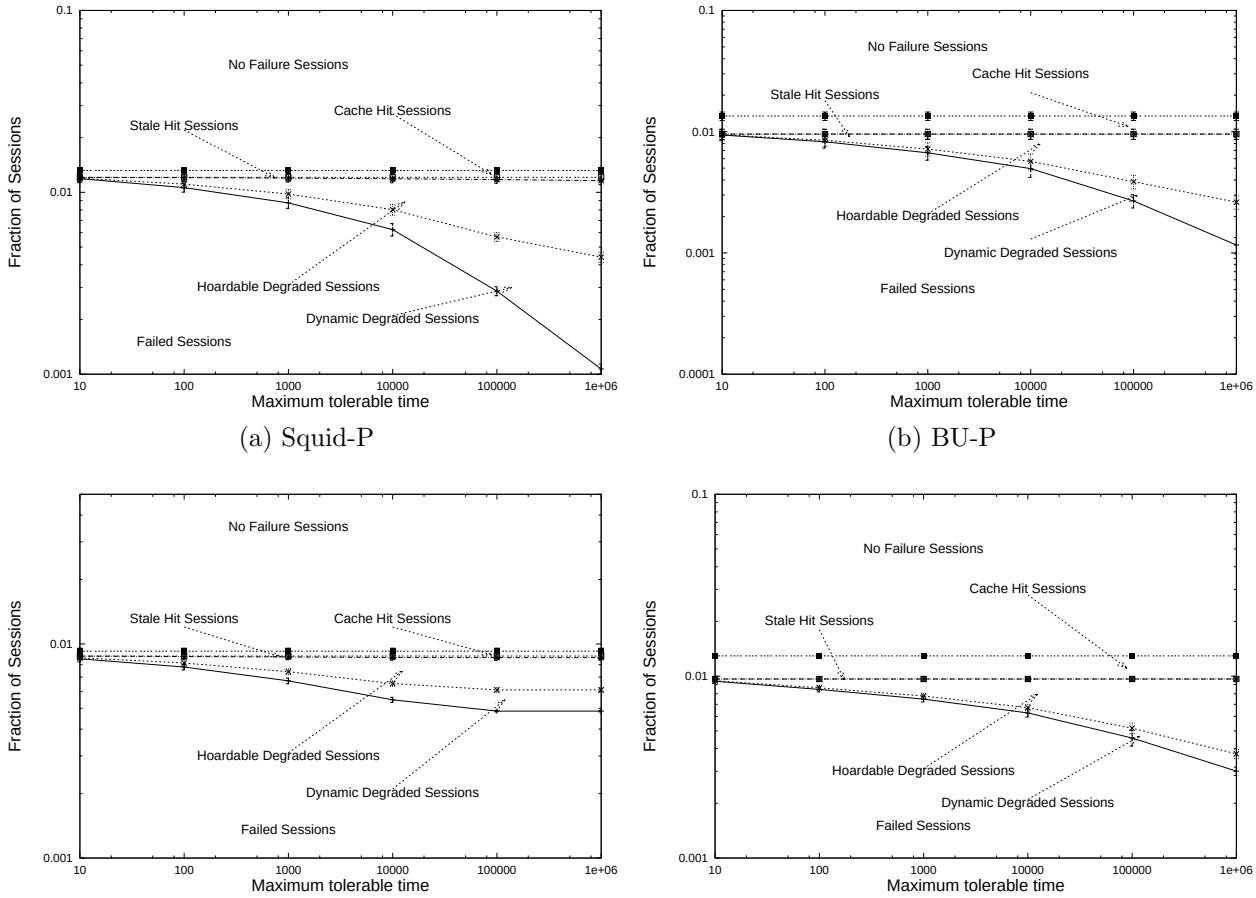
(a) BU-P (b) BU-C

(c) BU-P-checksum (d) BU-C-checksum

Figure 2.4: Availability improvement v. fraction of services.

the extension is installed at the client or proxy, no requests will encounter any failures for this session. Therefore, a popular service is likely to gain better availability improvement.

Finally, we examine the maximum duration requirements for client-independence techniques to improve service availability to different levels. Figure 2.5 shows the service improvement as we increase the maximum time that client-independence techniques support. For both proxy traces (Squid and BU), we gain one order of magnitude improvement at $1e + 06$ seconds. In other words, to improve the service availability one degree better, we have to mask failures lasting for $1e + 06$ seconds. Although designing techniques to



(a) Squid-P (b) BU-P

(c) Squid-C (d) BU-C

Figure 2.5: Session failure rate v. maximum tolerable time required.

tolerate long failures could be challenging, the employed failure model suggests that long failures are rare.

These experiments suggest that to take full advantage of client-independence techniques for improving availability, the virtual machines on clients and proxies must be scalable to handle hundreds or thousands of simultaneously downloaded extensions in order to replicate a significant fraction of accessed sites. And some of techniques might require a large amount of system resources to mask long failures. We examine the resource management challenges posed by such a scalable workload in a separate study [27].

2.3.3 Network routing

In this section, we evaluate strategies that route around network failures. To simplify the analysis, we classify strategies into two broad categories: network re-routing and server replication and selection. The following discussions state the type of failures masked by each strategy, how we model the strategies in our experiments, and the service availability improvements that the strategies yield.

1. **Re-routing.** Techniques of this category send requests to the service's original server. But they may use alternate routes when failures occur. Overlay networks [97, 110, 98, 105, 54, 139] are examples of re-routing techniques. In the terminology of this paper, these techniques address in-middle failures, but will be ineffective against near-source and near-destination failures.

2. **Server replication and selection.** This category of techniques directs requests to replicas of the origin servers when the origin servers are unreachable. Several file systems [104] and databases [82] provide replicated servers to handle failures in distributed environments. In the context of the Web, mirror site with "manual failover", as well as replicated servers with anycast [15, 40, 133] can support server replication. This class of techniques can resolve near-destination and in-middle failures but is ineffective against near-source failures.

As in our analysis of client independence techniques, we abstract implementation details of routing-based techniques and focus on bounding improvements that they may provide. Several factors may limit these improvements in practice. For re-routing strategies, overheads include the failure detection time and route switching time. For server replication and selection, there are costs to maintain extra replicas and overheads to select alternative servers. These overheads vary for different implementations and may vary for different services (e.g. depending on failures, consistency, and semantics). Therefore, as with client-independence techniques, clients may experience sessions handled by re-routing

or server replication as "degraded" with the significance of the deterioration varying on a service-by-service and implementation-by-implementation basis.

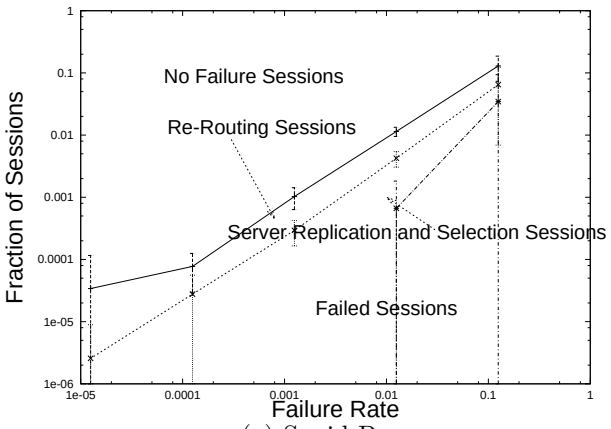
We group requests into sessions, and classify each failure by its network location: near-source, in-middle, or near-destination. We run each experiment 25 times and plot the mean with 90% confidence intervals.

Results. In our first set of experiments, we vary the fraction of failures in each location category. These graphs are omitted due to space limits. Across a wide range of ratios, the findings are as expected: the fraction of failures that each class of techniques can handle varies in proportion to the fraction of failures assigned to a particular location category. For example, when in-middle failures account for 50% of all failures, techniques that avoid in-middle failures but not others can improve failure rates by about a factor of two. Given that experiments found significant fraction of failures at each location, Amdahl's Law limits improvements from routing based strategies that do not address failures in all three locations.

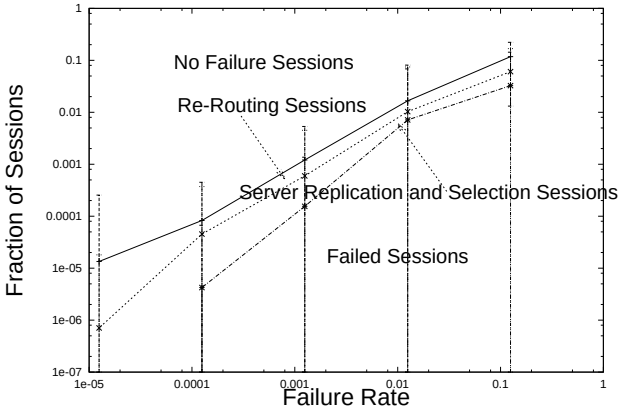
Figure 2.6 shows the sensitivity of these results as we vary the network failure rate. As for the client-independence strategies, the relative improvements to failure rates provided by these techniques remains stable over a wide range of underlying failure rates. More study is needed to quantify the prevalence of near-source failures precisely, but the preliminary result of our study suggests that near-source failure account for at least 10%-20%, probably limiting routing-based techniques to less than an order of magnitude improvement. As noted above, our methodology is likely to underestimate near-source failures.

2.3.4 Combined Techniques

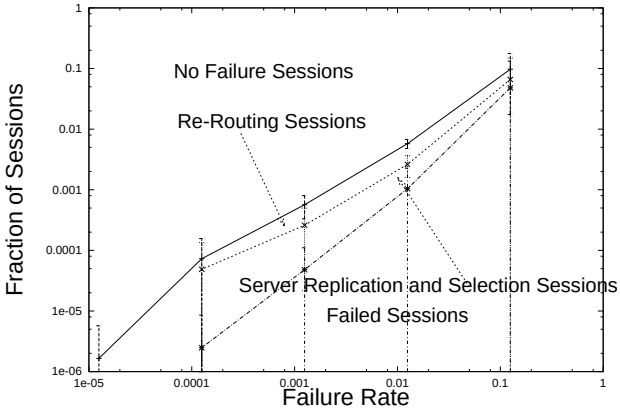
Client-independence techniques are limited by compulsory misses and installation time, and re-routing techniques are limited by near-source failures. Since these techniques fail in different circumstances, they may be combined to reduce system unavailability.



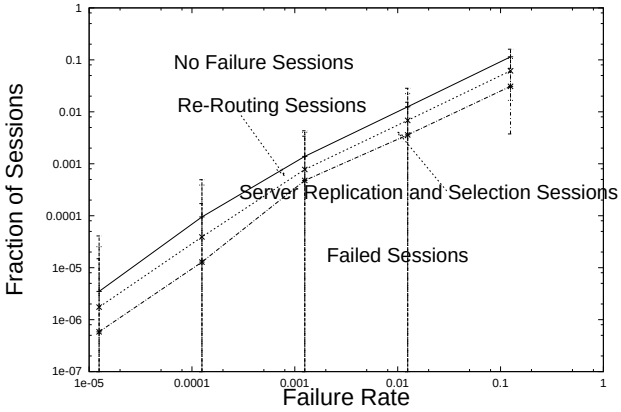
(a) Squid-P



(b) BU-P

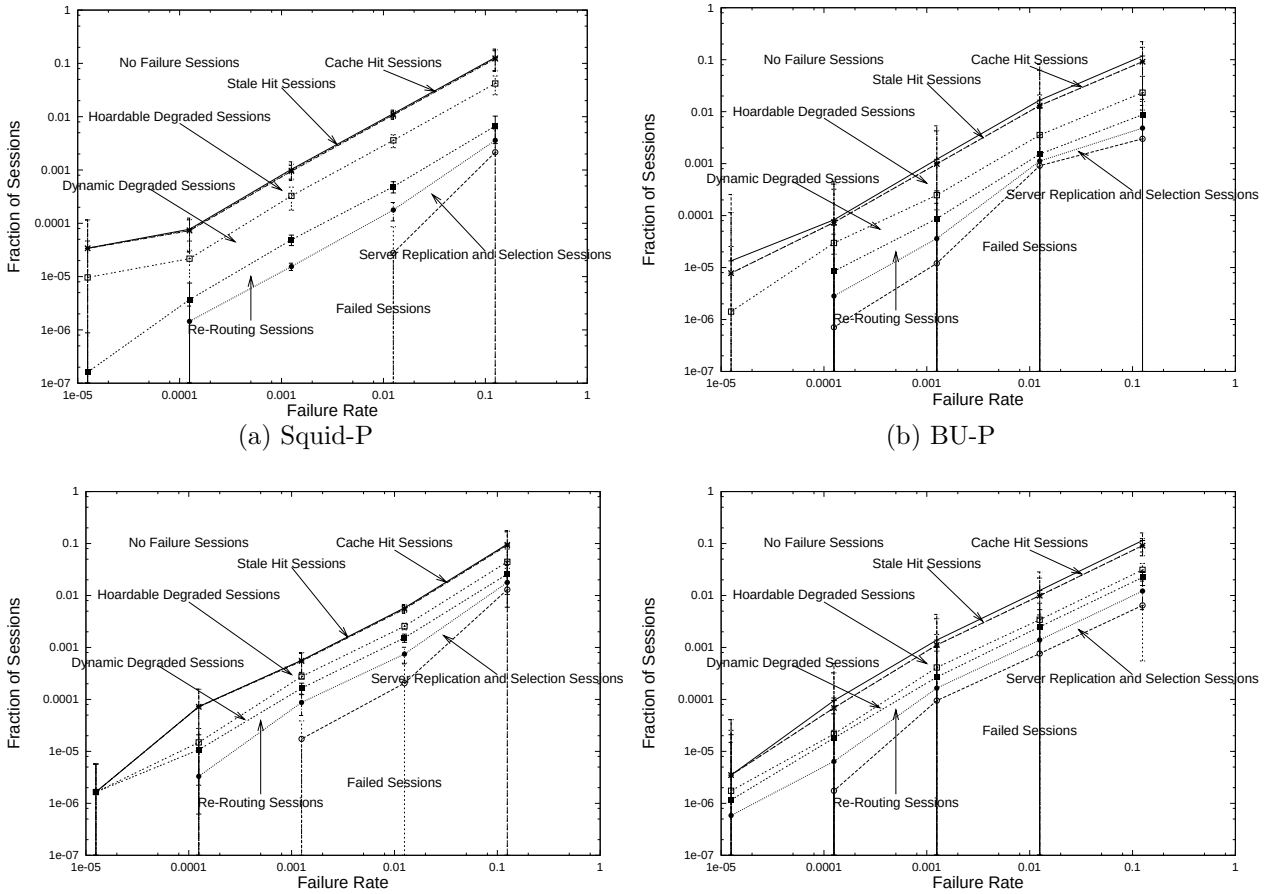


(c) Squid-C



(d) BU-C

Figure 2.6: Session failure rate v. network failure rate.



(a) Squid-P

(b) BU-P

(c) Squid-C

(d) BU-C

Figure 2.7: Session failure rate v. network failure rate (all techniques).

For example, Figure 2.7 shows session failure rates under a combined scheme in which failures are masked by caching, prefetching, and active objects and in which prefetching and installation of active objects use anycast to access replicated servers. This combined approach thus masks all failures except near-source failures during prefetching or active object installation time. Figure 2.1 suggests that these results will be relatively insensitive to increases in *install_time*. Overall improvements in BU-P for this combined scheme are factors of 117, 100, 18.2, and 24.5 for network failure rates of 0.0125%, 0.125%, 1.25%, and 12.5%, respectively. This relatively wide improvement range appears to be due to experimental variation magnified by the small number of failure events observed in the simulations. Other three graph, (a), (c) and (d) seem to have some variations in improvement at different failure rate as well.

2.4 Discussions

2.4.1 Result summary

The simulation results above provide quantitative evidence that a centralized architecture cannot solely rely on traditional caching technique to significantly improve the end-to-end availability of Internet services. Although service providers can host Internet services on highly available server clusters, deploying Internet services with high end-to-end service availability remains problematic due to connectivity failures. A typical client may not be able to reach a typical server for 15 minutes per day. During the disconnection, traditional caching has limited availability improvement (speedup) of up to 1.6 while combining two classes of techniques achieves a factor of over 100 availability improvement.

2.4.2 Dissertation context

This dissertation offers a semantic-aware replication (SAR) solution in the context of the edge service model that is an abstraction of the infrastructure made commercially available

by Akamai [2]. The edge service model outlines a distributed infrastructure where Internet services are replicated and distributed to the *edge* of the Internet, i.e. being physically closer to end users rather than at the site of the service provider. Server replicas hosting replicated Internet services at the edge of the Internet are called *edge servers*. On edge servers, both business logic (code) and storage systems are replicated either fully or partially. When an end user accesses an Internet service through a web browser, the request sent by the browser will be directed to the nearest available edge server by some routing techniques. Although there is no limit to how many edge servers can be used in a typical deployment, our SAR solution is targeted to support up to a few hundreds of edge servers. Because the dissertation focuses on the data replication component of the edge service model, we assume that we can simply refer to existing techniques to implement other components such as the routing of user requests and security related issues.

Chapter 3

Semantic-Aware Replication for TPC-W Benchmark

Existing replication approaches fail to provide the optimized trade-offs among CARP (consistency, availability, response time, and partition-resilience) when used to replicate dynamic data in the edge service architecture. As we have described in Chapter 2, traditional architectures are incapable of delivering high end-to-end availability (i.e. "four-9's"(99.99%) or "five-9's"(99.999%) of availability) and low response time (i.e. approximating LAN delays). The emerging edge service architecture aims to solve this problem by distributing Internet services to a collection of edge servers across WANs and near end users to process requests [2, 9, 21, 119, 127]. This architecture minimizes communication over WANs during request processing in order to improve service availability and response time. However, studies [20, 73] suggest that large-scale distributed systems like the edge service architecture always face trade-offs among CARP when distributing/replicating its underlying dynamic data.

This chapter describes the *semantic-aware replication* (SAR) that offers the optimized CARP trade-offs for our prototype e-commerce system running atop the edge service architecture by exploiting the semantics of shared data sets and encapsulating the corre-

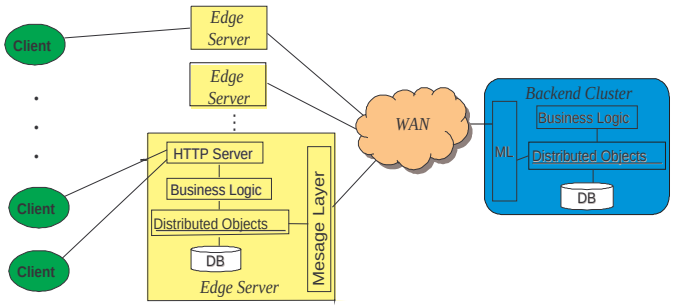


Figure 3.1: Internet edge service architecture

sponding replication techniques within distributed objects [121]. Standard e-commerce implementations allow business logic (e.g. servlets, Enterprise Java Beans, or CGI programs) to access the central databases directly. However, if business logic were distributed, accesses to a central database would become costly remote operations. We target an edge service architecture that replicates both business logic and data to edge servers to minimize the accesses to a central database. Data shared among the edge servers are encapsulated within distributed objects that are aware of the semantics (i.e. workloads, update topologies, and data properties) of the underlying shard data. As illustrated in Figure 3.1, we deploy business logic, distributed objects, a database, and a messaging layer on a set of distributed servers that are accessed by clients via standard HTTP front ends. The distributed objects interpose between the business logic and the local database to control data access. They also communicate with other instances of the distributed objects through the persistent messaging layer [31, 59, 109] to manage data replication and consistency.

Our prototype system targets the TPC-W benchmark [32] that simulates an online bookstore. To explore data replication issues in the edge service architecture, our study uses a variation of the TPC-W benchmark, called the distributed bookstore. When we replicate the benchmark on multiple edge servers and allow edge servers to process requests using their local databases, we have to manage the consistency of the data across multiple databases in WANs.

SAR includes five simple distributed objects to manage the consistency of different shared data sets of the prototype system. The *catalog* object maintains catalog information in our system. It exploits the fact that catalog updates occur at one place and are read at many others. We use the *order* object to collect finalized orders at multiple locations and process them at the backend server. This object takes advantage of the fact that many nodes write orders but only one needs to read them, and it exploits the loose requirements on sequencing updates across nodes. The *profile* object represents the user profile information. It takes advantage of the low concurrency of accesses to each record, and it makes use of field-specific reconciliation rules to cope with (rare) update conflicts [114]. The *inventory* and *best-seller-list* objects track a bookstore's inventory and best seller lists. The *inventory* object exploits the fact that edge servers care about whether the inventory is zero but do not need to know the actual value. The *best-seller-list* object takes advantage of the fact that a few purchases of a non-popular book do not necessarily alter its ranking.

Encapsulating database access behind object-specific interfaces yields many advantages. First, client requests are locally satisfied by distributed objects, which asynchronously manage the local database consistency. Thus, edge servers are able to continue to operate even in the case when network partitions occur; and because requests are satisfied locally at edge servers, the response time is better than that of the centralized system. Second, each distributed object can make use of object-specific strategies to replicate data and to enforce exactly the consistency semantics it requires. Third, distributed objects restrict data access to a narrower interface than a general database interface, which permits us to relax consistency guarantees internally while preserving the consistency requirements from the application.

We construct and evaluate a prototype system based on Apache web servers, Tomcat Servlet engines, the JORAM implementation of the Java Message Service, and a DB2 database, and we find that the prototype has excellent availability, consistency, and performance. Under this implementation, our edge servers approximate the ideal system in

which high speed and reliable links connect end users to service front-ends and connect service front-ends to backend databases. For instance, our system continues to process requests with the same throughput and response time before, during, and after a 50-second network partition that separates edge servers and the backend server. And the response time of our system is nearly 5 times better than that of the traditional centralized system, in which end users connect to web servers via slow WAN links.

Qualitatively, we find the semantic-aware consistency rules easy to build and understand. We speculate that this approach may be useful for engineering systems for two reasons. First, once developed, distributed objects encapsulate the complexity of data replication and provide simple interfaces for engineers to use to build edge services without worrying about the intricacies of consistency protocols. Second, for the experts constructing the distributed objects, the restricted interface makes it easier to build distributed objects with the ability to handle consistency than to write reconciliation rules [114] for generic database interfaces.

SAR's main contribution is to demonstrate that object-based data replication makes it easy to build a distributed e-commerce web service and thereby dramatically improve both availability and performance. Although we focus on TPC-W and the more demanding distributed bookstore benchmark in this study, we speculate that similar techniques might also apply to a broader range of applications. Some consistency optimizations we exploit are similar to some proposed in previous studies [84, 86, 112, 114, 121, 135], but our emphasis is on how to integrate application semantics with the design of replication algorithms and effectively apply the class of specially tuned algorithms to make a broad range of applications work.

3.1 TPC-W Background

TPC Benchmark W (TPC-W) is an industry-standard transactional web benchmark that models an online bookstore [32]. It is intended to apply to any industry that markets

and sells products or services over the Internet. It defines both the workload exercising a breadth of system components associated with the e-commerce environment and the logic of a business oriented transactional web server. The benchmark defines activities including multiple concurrent online browsing sessions, dynamic page generation from a database, contention of database accesses and updates, the simultaneous execution of multiple transaction types, and transaction integrity (ACID properties). Above activities are the crucial components in many e-commerce applications, but their weights may be different across applications.

The benchmark defines three *scenarios* (workload mixes): browsing, shopping, and ordering. The *browsing scenario* consists of a mix of 95% browsing interactions, such as searches and product detail displays, and 5% ordering interactions, such as shopping cart activities and customer registrations. The *shopping scenario* consists of a mix of 80% browsing interactions and 20% ordering interactions. The *ordering scenario* comprises equal amounts of browsing and ordering. For scalability measurements, the benchmark defines the number of data entries which include numbers of unique books, registered customers, and book photos of various sizes.

The primary metric of the TPC-W benchmark is *WIPS*, which refers to the average number of Web Interactions Per Second completed. This metric is used for measuring the system throughput. Another metric is the Web Interaction Response Time, (*WIRT*), which is used for measuring the responsiveness of the system.

3.2 System Design

3.2.1 Overall architecture

As Figure 3.1 indicates, the edge services architecture consists of a backend cluster and a collection of edge servers distributed across the network. The common components on both edge and backend servers are business logic, a messaging layer, a database, and the

distributed objects. Edge servers have an additional component, the HTTP front-end server, through which clients access the service.

The edge services model works as follows. Clients use HTTP to access services through edge servers that are located near them. A number of suitable mechanisms for directing clients to nearby servers exist [2, 16, 40, 119, 133], and these mechanisms are orthogonal to our design. The HTTP front-end passes user requests to business logic units for processing and forwards replies from the business logic units (e.g. servlets, cgi, or ASP) to the end users. The business logic executes client requests on the edge server, and it stores and retrieves shared data using the interface provided by distributed objects. Each distributed object stores and retrieves data in the local database and also communicates with remote instances of the object in order to maintain the required globally consistent view of the distributed state [121]. Distributed objects use JDBC to operate on the local database and use the messaging layer to communicate with instances on other servers.

The messaging layer uses persistent message queues [31, 59, 109] for reliable message delivery and an event-based model for message handling at the receivers. To ensure exactly once reliable delivery even in the presence of partitions and machine crashes, the local messaging layer instance logs messages on the local disk before attempting to send them. Upon the arrival of each message at its destination, the destination's messaging layer instance invokes the message handler to pass this message to the corresponding distributed object instance. The messaging layer provides transactional send/receive for multiple messages.

We choose IBM DB2 for the database in our distributed TPC-W system. On each edge server, we use the Apache Web Server as the HTTP front-end and Tomcat servlet engine to host business logic servlets. We use a third party implementation of Java Message Service (JMS), called JORAM [60], for the messaging layer. In some of our experiments, we find that the relatively untuned JORAM implementation limits performance. Therefore, as a rough guide to the performance that a more tuned messaging layer might deliver, we also implement a *quick messaging layer* that provides the same interface as JORAM but

without the guaranteed correct behavior across long network partitions or node failures. We report performance results for both systems. We modify the TPC-W database schema and business logic for the TPC-W online bookstore from the University of Wisconsin [118] to fit in our object-based edge service architecture. We add five distributed objects on both the backend and edge servers to manage the shared information, namely the *catalog*, *order*, *profile*, *inventory*, and *best-seller-list*.

In the rest of the section, we focus our discussion on the design of the five distributed objects. By targeting consistency requirements for each individual distributed object, we explain how to design simple consistency models to resolve the *CAP* dilemma in building a replication framework for edge services at the object level.

3.2.2 Design Principles

Design trade-offs for our distributed TPC-W system are guided by our goal of providing high availability and good performance for e-commerce edge services as well as by technology trends. When making trade-offs, we consider the fact that technology trends reduce the cost of computer resources while making human time relatively expensive [26]. Therefore, we are willing to trade hardware resources, such as network bandwidth and disk space, for better system availability and shorter latency for users as well as design simplicity and better consistency for system builders. Our first set of priorities are, therefore, availability and latency because both availability and latency directly impact the service quality experienced by end users. The second set of priorities are the consistency and simplicity of the system. Good consistency that restricts the range of *observable* behaviors by the memory system [42] is a high priority because a key challenge in any relaxed consistency system is reasoning about subtle corner cases. Increasing the strength of consistency guarantees makes this reasoning more straightforward for system designers. Simplicity is important for making the approach useful in practice by making the system feasible to understand, build, and deploy. The third and lowest set of priorities is optimizing resource usage such as

network bandwidth, processing power, and storage. Therefore, we seek a simple distributed object architecture that improves availability and response time while keeping throughput and system cost competitive with existing systems.

We have made several design decisions based on these priorities. We focus our attention on moderate scale replication with 2-20 edge server locations rather than large scale replication to hundreds or thousands of edge servers. Recent work has suggested that moderate scale replication provides better availability when consistency constraints are considered [137], and this assumption also simplifies the design of distributed objects. Because our main objective is to show the feasibility and the effectiveness of using the distributed object architecture for WAN replication, we place a heavy emphasis on simplicity, and we bypass a number of potentially attractive optimization options for each distributed object. Future work may further enhance the benefits of the architecture by systematically optimizing performance.

Our distributed object architecture assumes that edge servers are trusted. This requirement of trust is another argument for focusing on replication to a few (2-20) edge servers and not hundreds or thousands of replicas. This trust model is reasonable in the environment where the service provider owns and manages geographically distributed service replicas, and it also is appropriate when a service provider out-sources replication to a trusted edge service infrastructure provider or CDN that ensures physical and logical security of edge-server resources. We also assume edge servers and the backend server communicate through secured channels although our current prototype does not encrypt network traffic.

3.2.3 Distributed objects

Distributed objects may be a simple way to achieve high availability, good performance, and good consistency compared to a more general shared data interface for two reasons. First, the restricted interface of a given object encapsulates the internal state of the object and

may prevent data inconsistencies from being observed. Second, the fact that the workload may be known to each object allows the data replication protocol used by the object to exploit the specific workload characteristics to improve availability, responsiveness, or consistency.

In this section we discuss the design of the key distributed objects in our distributed TPC-W system. We seek to demonstrate insides on how semantic-aware designs of data replication/distribution can enhance the availability and responsiveness of the system.

The catalog object

The *catalog* object provides the abstraction of one-to-many updates. It accepts writes at one place and propagates changes to multiple locations for subsequent reads. In the distributed TPC-W system, we use this object to manage catalog information, which contains book descriptions, book prices, and book photos. Update operations on catalog data are performed at the backend and propagated to edge servers.

The interface of the *catalog* object includes a write operation that takes a *key-value pair*, and a read operation that takes a *key* and returns the corresponding *value*. The backend server issues updates by invoking the write operation, and edge servers retrieve the updates with the read operation. An update from the backend server must be seen at some future time by all edge servers, who retrieve a set of values corresponding to keys. For correctness, the system must guarantee FIFO consistency [113] (aka PRAM consistency [73]) in which writes by the backend are seen by each edge server in the order they were issued. Enforcing FIFO consistency guarantees that, for example, if the backend server creates an object and then updates a page to refer to that object, then an edge server that reads the new page will also see the new object. Note that because only one node issues writes, FIFO consistency is equivalent to sequential consistency [86]. But for this same reason it is much easier to implement than general sequential consistency. Also note that although FIFO consistency provides strong guarantees on the order that updates are

observed, it does allow time delays between when an update occurs and when it is seen by an edge server. Also, FIFO consistency does not require different edge servers to operate in lock step. For example, if a web page is updated while an edge server, *se1*, is unable to connect to the backend server, another edge server, *se2*, may still read and make use of this updated page while *se1* continues to use the old version.

In our prototype, the *catalog* object uses a simple push-all update strategy to distribute updates. Once the update is made at the backend, the *catalog* object immediately hands it to the local messaging layer for forwarding to all edge servers. Some time later, the update arrives at each edge server. The *catalog* instances at edge servers read the update, apply it to the local database, and serve it when requested by clients. Although this simple strategy can potentially use a lot of bandwidth by sending all updates, we see little need to optimize the bandwidth consumption for our TPC-W catalog object because the writes to reads ratio is quite small for the catalog information. In particular, TPC-W benchmark defines the catalog update operations as 0.11% of all operations in the workload.

This simple implementation meets our system design priorities. It provides high availability and excellent latency to our system because edge servers can always respond immediately to requests using local data. Furthermore, this implementation provides FIFO/PRAM consistency for shared catalog information using a straightforward approach.

Variations of the *catalog* object may be useful for other applications that require one-to-many *data dissemination* semantics. For example, a *dissemination* object could provide a mechanism for propagating edge service infrastructure information such as program or configuration updates. Similar behaviors can also be found in other applications such as IBM's geographically-distributed sporting and event service [23], traditional web caching, content assembly, dynamic data caching [24], and personalization. Systems may benefit from additional features/optimizations under different workloads. We discuss three of such features/optimizations that may be useful to other distributed applications, but that are not included in the design of the *catalog* object.

1. Atomic multi-object update: Some distributed applications require a mechanism to atomically update multiple objects. For example, it may be desirable to atomically update several component parts that are assembled into a single page [25]. Given the support of transactional updates provided by most persistent messaging layers, it should be straightforward to modify the *catalog* object to support atomic multi-object operations (read/write). Potential costs for this feature include a slightly more complex interface and/or a reduction in concurrency of writes and reads due to locking.

2. Data lease: The data served by some time critical applications, such as stock quotes, are meaningful only within a fixed interval. If the local data becomes excessively stale (for instance due to a network partition), some time-critical applications may prefer to deny service rather than serve bad data. To extend our *catalog* object to support such functionality, we could add a new parameter in the write operation to specify a lease [39, 47, 131] for each update. Of course such a feature may reduce availability because servers may be forced to deny service rather than serving stale data.

3. Bandwidth constrained update: Applications that have high write/read ratio with large data objects might not want to use a push-all strategy for propagating updates because it would take a lot of bandwidth to send all updates to all edge servers. Thus, applications with high write/read ratio might need a more sophisticated algorithm to propagate updates. Nayate et al.'s transparent information dissemination system [87] can be viewed as a highly tuned version of our catalog object. It implements both optimization 2 (data lease) and optimization 3 (bandwidth constrained update).

The order object

The abstraction of the *order* object is that of many-to-one updates. It gathers writes at various locations and forwards them to a single place for reading. In our distributed online

bookstore application, we use the *order* object to manage the propagation of completed orders. Locally, edge servers accept user orders, which need to be processed at the backend server for fulfillment.

The interface for the *order* object includes an insert operation that takes an *order*, an *order sequence ID*, an *edge server ID*, and a *message handler* that processes orders when they arrive from edge servers. Each order is identified by the pair, *edge server ID* and *order sequence ID*, which increments by one whenever a new order is created on an edge server. Orders are sent by each edge server in the sequence that they are initially created on that edge server, and the messaging layer delivers messages in the same sequence as they are inserted. Therefore, orders from the same edge server maintain FIFO consistency at the backend server but different servers' orders can be arbitrarily interleaved. The handler interacts with the persistent message layer to guarantee that all orders are to be processed exactly once by the backend order object instance.

An incoming message is deleted from the local messaging layer only if the handler successfully processes the order. If a crash happens while an order is being processed, the incomplete processing is rolled back during database recovery. In such a case, because the message handler did not complete, the messaging layer invokes the handler again during recovery. The handler also detects duplicates when it processes an order. In that case, it executes a *no-op* and returns to the messaging layer as if the order had been successfully processed.

The *order* object provides high availability and excellent latency to our system by decoupling edge servers' local requests processing from the persistent store-and-forward processing of orders to the backend server.

The mechanism of the *order* object can be extended for other applications. For example, because it supports FIFO consistency for updates from the same machine, we can use it to gather the system logs in distributed systems to, for example, gather user click patterns at a web site.

The profile object

The *profile* object handles reads/writes with low concurrency and high locality. Each entry contains information about a single user such as name, password, address, credit card information, and the user's last order. Users can only access or modify fields of their own profile records.

The interface of the *profile* object includes a simple read operation and a write operation. The read operation takes the *user ID*, and returns the corresponding profile record. The write operation takes the *user ID*, the *field ID*, and a *value*. The profile information has a low write/read ratio of less than 12.86% [32]. We assume the server selection logic that directs users to specific edge servers will generally send the same user to the same edge server for relatively long periods of time so that the user usually modifies his/her profile record on the same edge server. Therefore, the chances for concurrent access of the same profile record at two edge servers is generally low. However, sometimes users will be switched from one edge server to another (e.g. in response to geographic movement of the user, load balancing, or network or server failures). Therefore, we require an implementation of the *profile* object to allow edge servers to access any profile.

Given the low concurrency and high locality of access to profile records and relatively low volume of writes, our prototype implementation (1) uses a write-any read-any policy that does not require locking across servers, (2) propagates updates among all edge servers with best effort to propagate all changes quickly, and (3) applies object-specific "reconciliation rules" [91, 114] to resolve conflicting updates to the same field of the same record on multiple edge servers. Whenever a profile record is modified, the update is enqueued in the message layer and then sent to the other edge servers. If a set of edge servers is disconnected at the time of the update, the persistent messaging layer ensures delivery of the update after those servers recover. If two concurrent write operations update the same field of a record on different edge servers, the object code resolves the conflict with reconciliation rules at the field level. For example, the object-specific reconciliation rule for

the last-order field of a profile record is to compare the orders' timestamps and to select the more recent order; the rule for credit card records or shipping addresses is to merge multiple updates and prompt the user for selection when the client makes a subsequent purchase.

The design of the *profile* object ensures availability and minimizes latency by relaxing consistency compared to sequential consistency [20]. Updates can take place on any edge server without having to lock the targeted record. Access locality and rapid best-effort propagation of all updates to all locations reduce the number of conflicts [48], and rare update conflicts are satisfactorily resolved by simple per-field reconciliation rules.

Our decision to replicate all profile records on all edge servers maximizes availability, optimizes response time, and emphasizes simplicity at the cost of increasing storage space and update bandwidth in keeping with our design priorities. Because the profile objects are small and updates to them are infrequent, partial replication would modestly reduce overhead and might hurt performance, availability, or simplicity. However, systems with large numbers of replicas could see benefits from more sophisticated partial replication.

A wide design space exists for providing consistency on read/write objects in distributed systems [113], and the trade-offs selected for the *profile* object may not be appropriate for other read/write records. In an environment where access patterns and object semantics are less benign than the *profile* object, general approaches might proceed in two dimensions.

1. Strengthening consistency from the underlying FIFO/PRAM propagation of updates to provide stronger semantics such as casual consistency (which may require Bayou's anti-entropy [95]) or sequential consistency (which may require locking). Quorum based solutions such as [30] could also be explored.

2. The "reconciliation rules" currently hand-coded in the *profile* object logic might be made more general by, for instance, providing an interface on a read/write object to specify reconciliation rules as a parameter [114].

The inventory object

To examine consistency constraints beyond that of the standard TPC-W benchmark, our distributed-bookstore benchmark adds the constraint of a finite inventory for each item. It requires that if the inventory of an object is 0, users requesting this object must be notified that delivery may take longer than normal (e.g. the item is not in stock and is on back-order). We enforce this constraint with an *inventory* object. We observe that the actual count of the inventory is not important for processing order requests as long as stock is sufficient. The inventory responds either "OK" to process the order or "warning" for back-orders. It is acceptable to be conservative and issue warnings when the inventory is unsure whether items remain. (The downside is that users may cancel orders when they receive warnings in the ordering process. But we can minimize these false positives with careful system design and implementation.)

The inventory information can be interpreted as *ID* and *quantity* pairs. Every pair maps a particular book in the store to the number of copies of the book. The interface of the *inventory* object is the *reserve* operation, which takes a *numeric value* and a *book ID*, and returns a boolean value. If the returned value is *true*, it implies that the *reserve* operation successfully decrements the number of copies of the specified book by the given amount. If the inventory is insufficient to accommodate the request, *false* is returned. Note that the use of a transactional database and persistent messaging layer allows us to restore this escrowed inventory if the transaction fails to complete due to a failure or user cancellation.

In our simple prototype system, the total available inventory is divided among edge servers by giving each object instance a *localCount* and enforcing the invariant that the sum of all local counts across all instances never exceeds the global inventory count. Initially, inventory is evenly distributed among all edge servers. Edge servers process requests with their local inventory without communicating with the backend or other edge servers, and their local inventory decreases over time. We implement a simple protocol between the

backend server and edge servers for inventory re-distribution. By observing the orders received at the backend server (see section 3.2.3), the *inventory* object instance at the backend server keeps track of the edge server with the most inventory and the edge server with the least inventory. Whenever the inventory difference between these two servers exceeds a certain threshold, the inventory instance at the backend server requests inventory re-distribution between them. In this edge server pair, the one with higher inventory is the donor and the other is the recipient. Note that such a re-distribution request may fail because the backend might have stale information about donor's inventory. Such a failure is benign because the backend server eventually becomes aware of the donor's true inventory and selects a different donor. Also note that our use of a persistent messaging layer greatly simplifies the design of this redistribution by ensuring that inventory is never lost or duplicated in transfer.

The inventory implementation meets our design goals by increasing the overall availability of the system while providing acceptable consistency guarantees on the data served to clients. It also reduces the communication between edge servers and the backend because edge servers do not need to check availability of the central inventory upon every order request. Therefore, we improve the system response time and make the system more tolerant to network partitions. The limitation of our design is occasional "false positives" when local count is 0 and inventory instance reports *false* while counts on other edge servers is not 0. However, "false positives" only occur under some extreme conditions as illustrated in Section 3.3.4. Furthermore, we can reduce those rare cases with enhancements described below that we considered but did not adopt in our implementation for the purpose of simplicity.

1. Fetch on-demand: When the system realizes the local inventory is insufficient to accommodate an incoming request, it could delay processing the request and send messages to other edge servers to request more inventory. If it receives a positive response, the request could then be processed. If no positive response is returned

within a time period, the request would be reported as back-ordered as it is now.

2. Peer-to-peer inventory exchange: The mechanism of the *inventory* object is similar to the *numerical error* guarantee mechanism in TACT [135]. Unlike TACT, our system adjusts the local inventory with a centralized coordinator for simplicity. We could change this object to employ the peer-to-peer to model in which edge servers exchange inventory directly.

3. Adaptive redistribution: When a particular edge server experiences heavy demand for an item, the system might allocate a larger percentage of inventory to that edge server.

4. Partition-aware redistribution: Our current design of the *inventory* object assumes that no partitions last infinitely long so that lost messages can be recovered when partitions heal. Therefore, inventory may not leak from the system. But this design cannot bound the amount of inventory that may be lost when network partitions do not heal. Sussman et al. present four partition-aware resource distribution solutions in their Bancomat study [112], which can be used to distribute inventory in the environment where network partitions last arbitrarily long. Their solutions minimize the inventory leakage due to the lost of messages when network partitions occur.

The best-seller-list object

The *best-seller-list* object maintains lists of best selling books for each subject. In TPC-W the best sellers are the fifty most popular books computed for each subject based on the 3,333 most recent orders with each order containing up to 100 books.

The interface of this object includes a read operation that takes a *string* as the subject and returns a list of best selling books under the subject. The best sellers change over time as different books are sold. For the best seller lists to be accurate on every edge server, all sales activities on all edge servers must be taken into account when computing

Object	Object State Replication	Updates Propagation	Concurrent Updating Rules
Catalog	all records at all servers	backend ⇒ all edges	n/a
Order	1/N at edge; N/N at backend	edges ⇒ backend	timestamp ordering
Profile	all records at all servers	all edges ⇒ all edges and backend	field-level reconciliation rules
Inventory	local view at edge; all local views at backend	on threshold: an edge ⇒ backend ⇒ an edge	on threshold: timestamp ordering
Best-seller-list	approximate view at edge; current view at backend	on threshold: backend ⇒ all edge	on threshold: timestamp ordering

Table 3.1: Distributed object state replication and propagation.

the lists. However, the lists may not change on every sale. For example, several additional purchases of books that are already in the best seller lists may not change the lists. The system only cares about the sales activities exceeding some threshold. Furthermore, it is preferable to return slightly stale best seller lists rather than to stop serving requests. Some delay in propagating order information is also acceptable.

In our prototype system, we maintain a copy of the best seller lists on every edge server. The approach that we take to maintaining the best seller lists is similar to that for maintaining the inventory among edge servers. By observing the orders received at the backend server (see section 3.2.3), the *best-seller-list* object instance at the backend server keeps track of the sales volumes of all books. As soon as the lists change, the instance at the backend server sends messages through the messaging layer to *best-seller-list* instances on all edge servers to update the lists.

This simple implementation meets our design goal. It improves system response time and increases system availability by minimizing the communication among edge servers and to the backend server for computing and updating the lists and detecting the changes in the lists. It reduces bandwidth consumption and dependencies among edge servers by monitoring all orders at the backend server instead of exchanging order information among edge servers.

Table 3.1 contains the summary of state replication and update propagation of

distributed objects.

3.3 System Evaluation

The experiments evaluate the availability, performance, and consistency of the distributed bookstore system in normal operation and while the system is partitioned due to network failures.

3.3.1 Environment and implementation

To demonstrate our distributed bookstore system, we deploy a prototype across four servers, three of which act as edge servers and one as the backend server. Each server runs on a Pentium 900MHz machine with 256MB memory. IBM DB2 databases are installed on all server machines. On the three edge servers, we use Apache and Tomcat to host the servlets that implement the server logic. Machines in our lab are connected via 100Mbit Ethernet connections. However, in order to simulate a wide area network (WAN) environment among servers during experiments, we direct all the traffic (both in and out) of server machines to an intermediate router, which simulates WAN delays and temporary network outages with Nistnet [88]. In the remaining discussion, we refer to links via Nistnet with bandwidth of 10Mbit/s and latency of 50ms as WAN links, and we refer to direct 100Mbit/s links between machines as network links in a local area network (LAN). We use three client machines to generate workload. These three machines have Pentium 900MHz processors, and each of them connects to a separate edge server via a LAN link. One instance of the TPC-W client program is running on each client machine generating a pre-defined workload against each edge server. TPC-W defines three workload mixes, each with a different concentration of writes. In our experiments, we focus on the *ordering mix*, which generates the highest percentage of writes (50% of browsing and 50% of shopping interactions in this mix).

One of our goals was simplicity. It is difficult to precisely characterize the extend

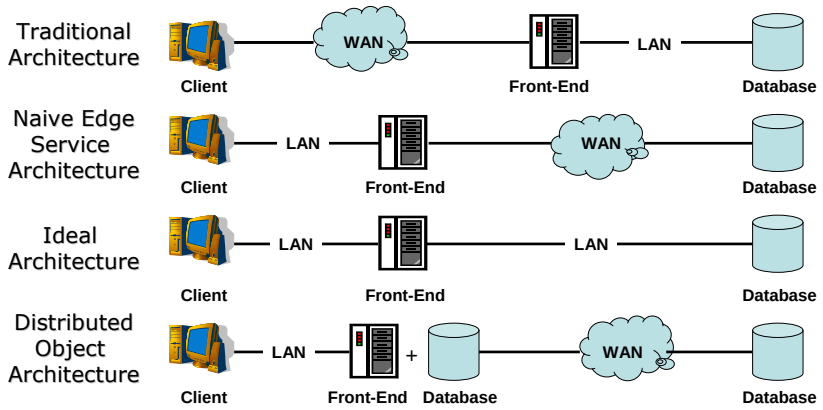


Figure 3.2: The network configuration of WAN service architectures.

to which the goal was met. Qualitatively, the designers regard the system as easy to understand. As a very rough quantitative measure, we note that the latest distributed TPC-W implementation consists of 6600 lines of source code, which is about 1000 lines more than that of the centralized version. The additional source code is primarily for the implementation of the specialized and simple replication protocols employed by distributed objects for managing shared data.

3.3.2 Performance

In this section, we evaluate the performance of our distributed bookstore system with respect to two criteria: latency and throughput. As noted in Section 3, our most important performance goal is to minimize the system latency because latency alters the expensive "human waiting cost." At the same time, we want to see if our system throughput is competitive with a traditional centralized architecture.

To evaluate the system performance, we run the benchmark on four architectures as illustrated in Figure 3.2. We use one frontend machine and one backend machine in this experiment to evaluate the performance of each architecture. The *traditional centralized* architecture has both its front-end and central database connected by LAN links, but

end users must access the front-end via WAN links. The *naive edge service* architecture replicates its front-ends at the edges of the network near end users. The front-ends connect to end users via LAN links and connect to the central database via WAN links. The *ideal* architecture has end users, front-ends, and the central database all within a LAN environment. This architecture is unrealistically optimistic, but it serves as a point of reference. The *distributed object* architecture, presented in this chapter, replicates both its front-ends and databases at the edges of the network near end users. The front-ends (edge servers) connect to end users via LAN links and connect to the core server and other front-ends (edge servers) through the distributed objects via WAN links. In addition to the one front-end system, we examine the performance of the distributed architecture with 3 edge servers (front-ends). For the communication layer, we use both JORAM that uses persistent message queues to send messages and the quick messaging layer that asynchronously sends messages without storing them on the local disk. Note that the latter configuration is intended to illustrate the range of performance that different messaging layers might provide, but because it does not provide reliable messaging across failures, it would not be appropriate for production deployment.

By comparing the performance results of the distributed bookstore application across four architectures, we seek to demonstrate three points. First, at low workloads, the latency when using the distributed object architecture matches that of the ideal architecture and is significantly better than that of the traditional architecture or the naive edge server architecture. Second, the throughput when using the distributed object architecture is competitive with the ideal or the traditional architecture. Third, when the edge server becomes the bottleneck under heavy workloads, we can increase system throughput by adding more edge servers.

We measure both system throughput and response time while varying the request rate. In all systems we expect to have the best response time when the request rate is low. Then, as the request rate increases, the response time will increase as well, until the

maximum system throughput is reached and the system becomes saturated, at which point the response time will increase sharply.

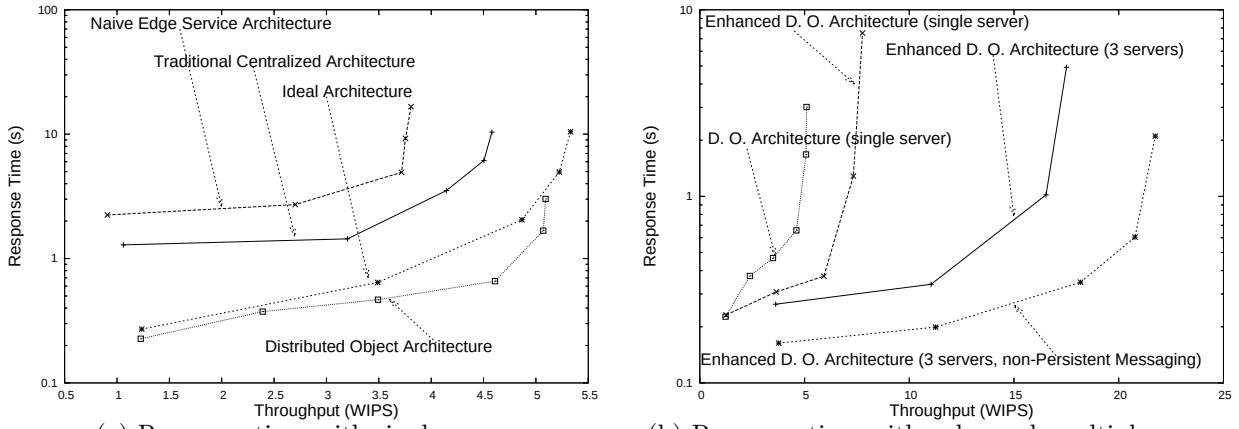
In Figure 3.3, curves from graph (a) indicate the performance of four architectures when using only one edge server. Graph (b) shows the performance of the distributed object architecture with additional enhancements and the scalability of the enhanced architecture when running on two different messaging layers, the JORAM messaging layer and the quick messaging layer. In both graphs, the x-axis represents the throughput in WIPS (web interactions per second), and the y-axis represents the response time of the bookstore application deployed on different architectures.

First, we explain the curves in graph (a) from the top to the bottom. The top curve represents the response time for the naive edge service architecture. This system experiences the worst minimum response time of 2.42s/req because a client request to the edge server usually triggers multiple requests from the edge server to the central database at the core server across the WAN. The WAN delays, which are set to 100ms RTT, dominate the system response time. In contrast, under the traditional centralized architecture, every client request goes across WAN links just once. The overall response time for the traditional centralized system is indicated by the second curve from the top, and it shows nearly a factor of two improvement to 1.25s/req. The third curve from the top indicates that the response time of the ideal architecture improves response time by nearly another factor of five, to 0.26s/req. The response time of the distributed object architecture is slightly better than that of the ideal architecture while both architectures yield approximately the same maximum throughput of 5.2WIPS, as indicated by the forth curve from the top in graph (a). The slight improvement of response time for the distributed object architecture is due the caching of *Shopping Cart* information at the edge server. (In all three other architectures, *Shopping Cart* information is stored in the central database only).

After demonstrating the excellent performance of our distributed object architecture with one edge server, we evaluate the performance of this architecture with multiple edge

servers. Before we add more edge servers to the system, we try to reduce implementation specific bottlenecks that may limit the system performance when heavier workloads are applied. In the subsequent tuning process: (1) we double the size of the main memory to accommodate message buffering; and (2) we cache the best-seller lists in each distributed object instance instead of computing them from the local database for client request. Note that both of these improvements are orthogonal to our decision to use distributed objects and the similar optimizations are applicable to the other architectures as well. The performance improvement of the enhanced architecture is illustrated by the second curve from the left in graph (b). Comparing with the first curve from the left in graph (b), which represents the performance of the original distributed object architecture, the enhanced architecture shows a consistent response time at approximately 0.26s/req under the moderate workload and an improved maximum throughput of nearly 8WIPS. After we add two more edge servers, the maximum system throughput of the enhanced architecture increases to 17.5WIPS as shown by the third curve from the left. Notice that we do not achieve a full linear improvement in throughput by adding two edge servers. This speedup shortage is due to both the overhead for the message logging and the inefficiency of the JORAM messaging layer implementation. We also implement a quick messaging layer that sends messages without logging them to disk first. The maximum throughput of the system with three edge server reaches 21.74WIPS when using the quick messaging layer for the message exchange, as illustrated by the bottom curve in graph (b).

The edge service architecture sends all updates to the backend server, which ultimately limits scalability of throughput. However, two facts allow adding more machines to increase system throughput. First, the read operations, which constitute more than 50% of the workload, are distributed among edge servers. Second, technology exists to make the backend database scalable, and indeed current centralized architectures achieve good scalability by directing all of their queries to a scalable backend database. We believe that the distributed architecture approach should be viewed as a way to increase availability



(a) Response time with single server (b) Response time with enhanced, multiple servers

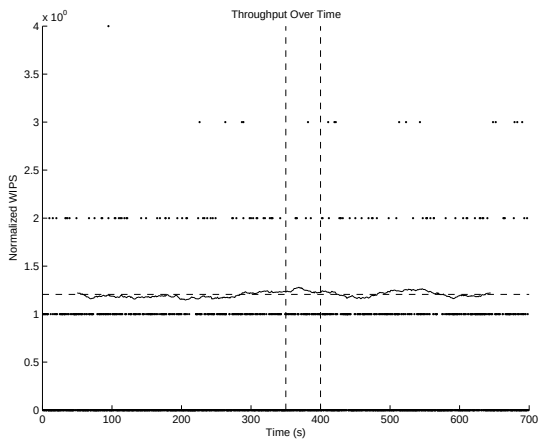
Figure 3.3: System response time as the workload increases.

and improve latency while scalability of throughput is improved with cluster technology.

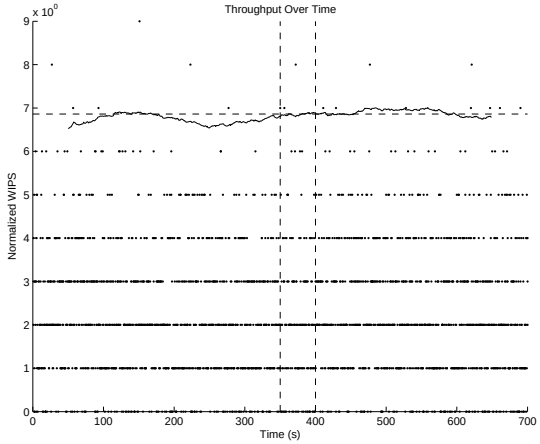
The throughput of our distributed TPC-W bookstore system is competitive with that of other academic systems [5, 44, 118]. If we assume that a typical Pentium III machine costs roughly $800, the price/performance cost of our system is roughly 147.19-182.85$/WIPS, which falls in the range of published standard industry TPC-W performance results, 24.50-277.80$/WIPS [128]. The throughput of the enhanced distributed object architecture is primarily limited by the throughput of its underling databases, which is not the concern of our current investigation.

3.3.3 Availability

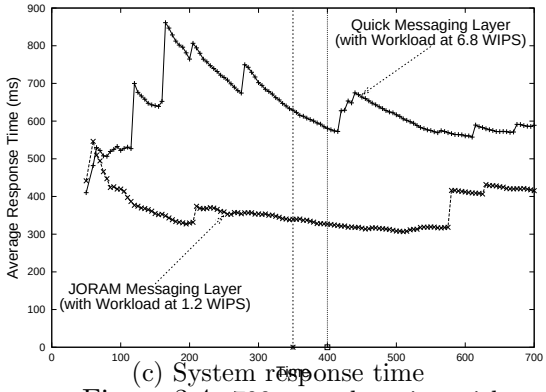
A key aspect of our design is that each edge server processes all requests with only local information. As long as a client can access any edge server, it can access the service even if some of the servers are down or if network failures prevent communication among some or all of the servers. In this section, we examine the performance impact of message buffering and processing during and after failures with both *JORAM Messaging Layer* and *Quick Messaging Layer*.



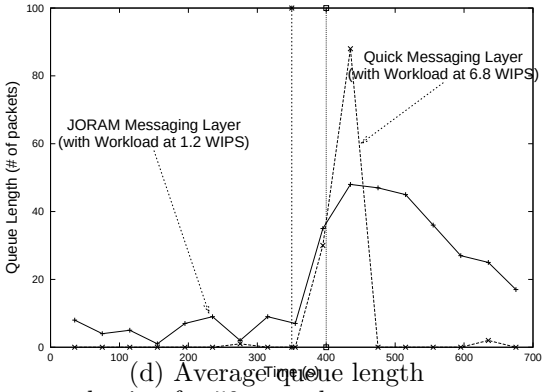
(a) System throughput (JORAM Messaging Layer) with Workload at 1.2 WIPS



(b) System throughput (Quick Messaging Layer) with Workload at 6.8 WIPS



(c) System response time



(d) Average queue length

Figure 3.4: 700-second session with network outage lasting for 50 seconds.

Figure 3.4 shows the system throughput, average response time, and message queue lengths when the system uses either *JORAM Messaging Layer* or *Quick Messaging Layer* before, during, and after a network failure. Each run lasts for 700 seconds, and a network outage occurs roughly 350 seconds after the experiment starts and lasts for 50 seconds. During the network outage, no server can communicate with any other server, but the normal communication among servers resumes once the network is restored. To provide a moderate load that does not cause queues to develop before the network fails, we apply a workload of 1.2WIPS to the system that runs on the top of JORAM and apply a workload of 6.8WIPS to the system on the top of the quick messaging layer.

Graph (a) and (b) in Figure 3.4 indicate the system throughput throughout the

700-second session. The x-axis represents the time progression and the y-axis represents the system throughput. In each graph, the straight horizontal dashed line represents the average throughput of the 700-second session and the solid slightly wiggly line represents the running average of throughput over 50-second intervals. The wiggly line stays close to the straight dash line in both graphs. It implies that the throughput of systems with both messaging layers is consistent throughout the session, and the network failures during the session have little effect on the system. Our distributed TPC-W system can operate normally while being partitioned because the databases are replicated locally through distributed objects, and they can continuously provide data for server computation while partitioned by network outage.

Figure 3.4 (c) shows the 50-second average response time for the two systems, one running JORAM messaging layer, the other running the quick messaging layer. The x-axis in the graph represents the time progression in seconds and the y-axis represents the system response time. The system response time appears unaffected by the network outage during the session because the graph does not show an increase in response time during the failure interval, between 350 and 400. Because the response times for different interactions vary, the curves in this graph tend to fluctuate throughout the session.

Figure 3.4 (d) shows the average queue lengths in the two messaging layers. The x-axis represents the time progression and the y-axis represents the queue length in the number of messages queued. There are few messages queued by the messaging layers before the failure starts, but the number of queued messages starts growing after 350 seconds. The curve that represents the queue length of the quick message layer indicates a sharper increase in message length than that of JORAM because the workload used on the quick message layer is about 4 times bigger than the workload on JORAM. But all messages are quickly cleared out of the queues after the network partition is fixed. Note that the JORAM Messaging Layer clears out queued messages relatively slower because it has a fixed message forwarding rate, approximately 4 msg/sec, which is much less than that of

the Quick Messaging Layer. This behavior is due to the persistent queuing overhead and the vendor specific design of JORAM Messaging Layer.

During the network failure, the information on each edge may become stale. However, instead of completely stopping sales during these failures, the service provider prefers to continue serving users with stale information, such as a stale catalog and stale best seller lists, accepting orders with stale inventory which may cause false-positive back-order rate, and delaying orders to be processed at the backend server by buffering them on local disks. These trade-offs seem appropriate and acceptable for this application.

3.3.4 Consistency

Because the system slightly relaxes consistency for higher availability and performance, users may view stale information even during normal system operations. In this section, we evaluate the impact of the relaxed consistency model on the distributed bookstore system, during normal operations, by examining the staleness of local best-seller lists and local inventory.

Local inventory: By distributing the bookstore inventory among all edge servers, the system allows edge servers to accept orders locally. However, when a heavy workload is unbalanced across servers and the inventory is low, some books may be sold out on a particular edge server during a short time frame before the inventory re-distribution arrives from other edge servers. In this case, some order requests targeting the sold out books may pessimistically report that the shipment may be delayed. In this experiment, we examine the false-positive back-order rate under a condition where the inventory is low and workload is unbalanced. We expect the false-positive back-order rate to approximate the ideal back-order rate seen by a centralized system as long as the inventory re-distribution time is less than the inter-arrival time between requests targeting the same book.

In order to create a purchasing imbalance across edge servers, we direct all order requests to only one of the three edge servers. To maintain a low inventory count at each

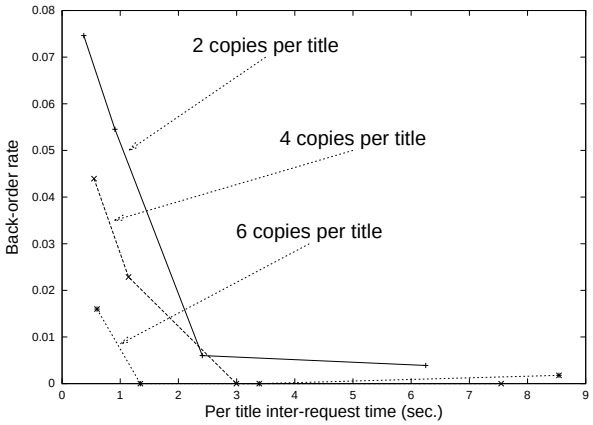


Figure 3.5: The back-order rate.

edge server, we choose three sets of inventory for each run of the experiment: *2 copies per title with 5 titles*, *4 copies per title with 5 titles*, and *6 copies per title with 5 titles*. The workload is designed such that each order request randomly targets one of 5 books, and we run the experiment long enough so that the average total number of books ordered is 50% of the inventory on the edge server, which is roughly 16.7% of overall inventory in the system. By varying the average inter-arrival time of requests targeting the same book, we can measure the average back-order rate for different sets of inventory. If we run against the traditional centralized architecture with given sets of inventory and workload, there will be no back-order because even under the most extreme case where all requests target the same book in the centralized system, the total number of requested copies is less than the number of copies of any particular book. The ideal back-order rate is zero for the defined sets of inventory and workload.

Furthermore, we speculate that if the distributed-object architecture has the inventory re-distribution time (RDT) much less than the requests inter-arrival time (RIT) per title, the distributed-object architecture can approximate the ideal back-order rate, i.e.:

$$RDT \ll RIT/titles$$

Figure 3.5 shows the percentage of orders resulting in false-positive back-orders due

to the inventory shortage as we vary the request inter-arrival time per book title. In the graph, the x-axis represents the average inter-arrival time of requests targeting the same book and the y-axis represents the percentage of rejected requests over all requests. All three curves approach the x-axis (the true back-order rate of orders) as they extend to the right where the request inter-arrival time is large. The workload that has the average request inter-arrival time of less than 2 seconds has the false-positive back-order rate greater than 1%. It indicates that our system inventory re-distribution process takes roughly 2 seconds or less to complete, which is expected because edge servers use asynchronous message exchange across WAN for computing and re-distributing inventory.

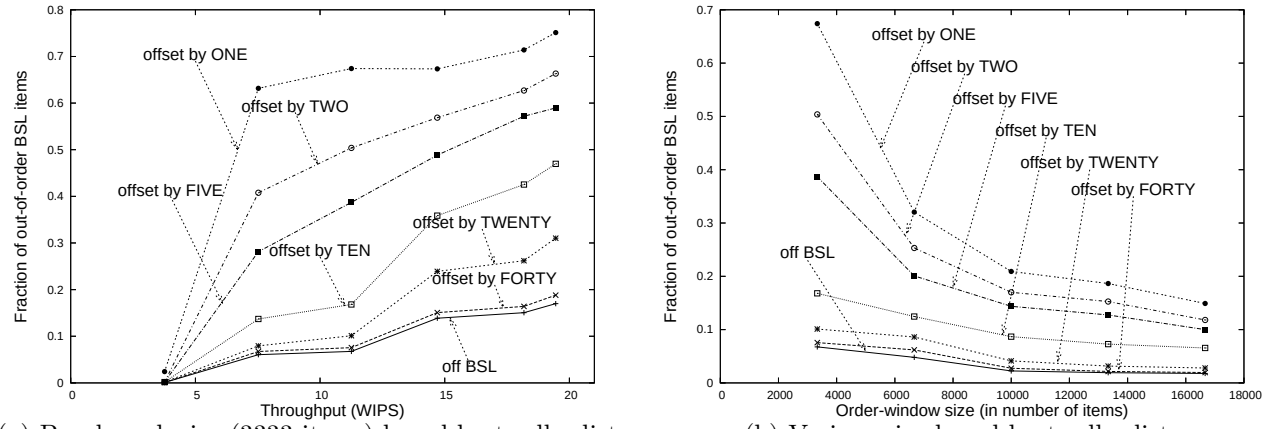
It is worth noting that the small back-order rate shown in Figure 3.5 only represents the system consistency in the extreme cases where inventory is small, 2-6 copies per book with 5 different books, and the workload is unbalanced. Also as noted in section 3.2.3 several optimizations can be applied to further reduce the system inconsistency.

Local best-seller lists: To maximize the availability and performance, every front-end in our system keeps a local copy of best-seller lists, which refer to the fifty most popular items in the most recent *order-window* in every category. Note that an *order-window* refers to a given number of purchases completed. The back-end monitors incoming orders that can potentially alter best-seller lists. Whenever incoming orders trigger a change in best-seller lists at the back-end, the back-end multicasts the change to all front-ends. However, changes may not immediately be reflected at all front-ends because of the asynchrony of updates to front-ends. Because front-ends always serve clients with their local copies of best-seller lists, stale best-seller lists are sometimes returned to clients. In this experiment, we evaluate the effect of such a lazy update approach on best-seller lists in the distributed bookstore, and we show that the amount of stale best-seller lists served by front-ends is small and tolerable under two conditions: (1) moderate workload that is within the system steady-state throughput; and (2) a reasonably sized order-window for computing best sellers.

Figure 3.6 shows, in the best-seller lists served to clients, the fraction of items that are out of position relative to their positions in the best-seller lists at the back-end. For this set of experiments, we use 3 enhanced edge servers all running atop the non-persistent messaging layer. Graph (a) shows that with the TPC-W defined order-window size of 3333, the fraction of out-of-position items increases as client workload increases. Graph (b) illustrates that, given the workload of 11.5WIPS, the fraction of out-of-position items decreases when the order-window for computing best sellers increases.

In graph (a), the x-axis represents the system throughput in WIPS, and the y-axis represents the fraction of client-received items that are out of position with respect to their positions in lists at the back-end. In this experiment we use TPC-W defined parameters: the order-window for computing best-sellers is 3333 and each order purchases up to 100 items [32]. Each curve in the graph indicates the fraction of client-received lists with items that are off their original positions at the back-end by at least the specified number of positions. For instance, the top most curve shows the fraction of items that are off their original positions by at least one. Respectively, curves below the first one indicate the fraction of list entries displaced by at least two positions, five positions, etc. The bottom most curve shows the fraction of items that are not in the client-received best-seller lists but are in the lists at the back-end. Notice that when the workload is light (e.g., less then 5WIPS), we see few (less then 1%) out-of-position items returned to clients. Under a light workload there is only a small probability for a best-seller query to closely follow a purchase request that alters the best-seller lists, but as the workload increases, this probability increases. Under the high workload the fraction of out-of-position items is relatively high, e.g., the fraction of items with minimal off-distance of one is around 70% and the fraction of items that are entirely displaced from the best-seller lists is between 10% and 15%. However, more then half of the out-of-position items have an off-distance of less than 10.

This relatively high inconsistency ratio is due to the small size of the order-window:



(a) Benchmark size (3333 items) based best-seller lists (b) Various size based best-seller lists

Figure 3.6: Staleness of local best-seller lists subject to workload & the base size.

an order-window of 3333 purchase transactions spans only the past few minutes, and when the order-window is small, the probability that a new order can change the best-seller lists at the back-end is high. Graph (b) illustrates that as the order-window becomes reasonably large, the fraction of out-of-position items decreases. The x-axis represents the order-window size and the y-axis still represents the fraction of client-received items that are out of position with respect to their positions in lists at the back-end. Similar to graph (a), each curve in this graph represents the fraction of items with some minimal off-distance. Those curves indicate that when the order-window increases from 3333 (about 50 minutes) to 16665 (about 250 minutes) under a fixed workload of 11.256WIPS, the fraction of out-of-position items decreases by 75% on average, e.g., the fraction of items with minimal off-distance of one drops from 67.4% to 14.9% and the fraction of items that are incorrectly moved off the best-seller lists drops from 6.76% to 1.77%.

This result suggests that a system that takes a lazy update approach for maintaining best-seller lists can provide clients with the consistent data when the size of the order-window is sufficiently large. In practice, online bookstore systems usually track orders over days or weeks for computing best-seller lists, and the fraction of out-of-position items will

be smaller in such systems. We should note that even in cases where inconsistent best-seller lists are returned to clients, most client-received lists are useful since most items are only out of position by a small distance.

3.4 Summary

Our TPC-W bookstore is built using a distributed object architecture and appears to provide high availability and good performance. The throughput and response time of our system are consistent before, during, and after network partition. By measuring all WAN latencies of four architectures, we show that the response time of our system closely approximates that of the ideal system under a normal workload.

Building the replication framework with a distributed object approach is relatively straightforward. We design the consistency model for each individual distributed object by using the corresponding application specific semantics. It then becomes easy to reason about the trade-offs between availability and consistency for each object. Usually, we can slightly relax the consistency of a distributed object to achieve high availability and efficiency. In addition, distributed objects encapsulate the complexity of data replication and provide simple interfaces for applications to access shared data. Thus, an attractive software engineering strategy is to combine WAN availability, performance, and consistency expertise with semantics to craft distributed objects for a class of distributed application/services on the Internet. In our particular case, we provide a WAN replication library for building distributed e-commerce applications.

3.4.1 Scalability

Replicating shared data everywhere might be seen limit the system's scalability. We do not view the distributed architecture approach primarily as a way to improve the system scalability in terms of throughput, but mainly as a way to increase availability and improve latency. Still, edge services architecture provides opportunities to improve throughput by

processing reads at edge servers and absorbing bursts of updates in edge servers' message queues for deferred processing. Furthermore, partitioning techniques can be used to break each shared data sets into multiple subsets and replicate each subset among a group of edge servers.

Therefore, by adding new edge server groups and breaking each shared data sets into a larger number of subsets, we have the potential increase the throughput of the overall system. In our on-going study, we are exploring the partitioning and related routing techniques that can be applied in the edge service architecture to enhance the scalability.

3.4.2 Consistency related issues

In order to avoid complexity in our evaluation, we keep the design of distributed objects simple while meeting the performance and consistency demands of our TPC-W system. However, we see some limitations in the design of existing objects that might not be desirable when used in building systems with different consistency requirements. For instance, the design of the profile object uses Bayou gossip protocol that can result in conflicting updates. Because most conflict-resolution solutions are based on application-specific semantics, the corresponding implementations could be complex and not desirable for certain classes of applications. But on the other hand, algorithms that can prevent update conflicts usually suffer from their suboptimal performance or poor availability compared with our current solution. In next chapter, we address this issue by introducing a novel algorithm, the dual-quorum with volume leases.

As mentioned before, distributed objects are designed based on the specific application semantics such that they hide the complexity of WAN data replication and consistency with simple interfaces. In addition, objects provide consistency guarantees that are straightforward and easy to reason about for both developers and users of the objects. Our distributed TPC-W system works well using distributed objects because the consistency guarantee of one object has few dependencies on other objects. However, the assumption

we used in building the distributed TPC-W system may not hold in other distributed WAN applications/services. In Chapter 5 we outline the high-level design of the unified replication approach that provides a base for reasoning about the cross-object consistency for the object-oriented replication architecture. One important future work is to precisely characterize the system consistency guarantees in the presence of interactions among different consistency models.

The distributed objects maintain the consistency of each edge server such that each edge server has a consistent view of the shared state. However, occasionally the edge server selection algorithm may switch clients from one edge server to another to balance load or in response to node failures, network partitions, or client mobility, and clients could then observe inconsistency. For example, edge server *se1* may have a newer version of the catalog information than edge server *se2*. When a client is switched from *se1* to *se2*, this client may see older catalog information on *se2*. One solution to resolve this issue is to use client browser cookies to enforce Bayou's session guarantees [114] to ensure that clients always communicate with sufficiently updated servers. In this example, we would need to bring the state of *se2* up to that of *se1* before allowing the client to interact with *se2*. We will consider this feature in our future work.

Chapter 4

Dual-Quorum Replication

4.1 Introduction

This chapter introduces dual-quorum replication, a novel data replication algorithm motivated by the desire to support data replication for edge services [2, 9, 43, 127]. As Figure 4.1 illustrates, the edge service architecture attempts to improve service availability and latency by allowing clients to access the closest available edge servers rather than a centralized server (or a centralized server cluster). But as Figure 4.1 also indicates, in order to provide a single service from multiple locations, service logic (code) replicated on all edge servers must access a collection of shared data. Thus, support for data replication is a key problem in realizing the promise of Internet edge services.

By exploiting object-specific workload characteristics, we seek to design a data replication system for edge services that offers good trade-offs among availability, consistency, and response time. Although it is provably impossible to provide simultaneously optimal consistency, optimal availability, and optimal performance for *general-case* wide-area-network replication [20, 73], we can, perhaps, provide nearly optimal behavior for *specific objects* by taking advantage of a given application's workload characteristics. For example, our previous studies show how to provide nearly optimal replication for *infor-*

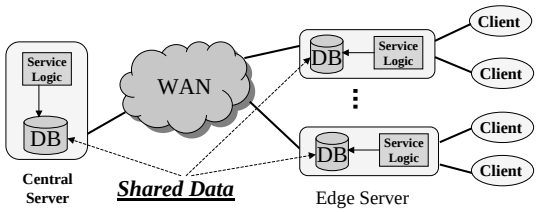


Figure 4.1: The edge service architecture

mation dissemination applications such as news [87] and *e-commerce* applications such as TPC-W [43]. In particular, we developed customized consistency protocols for three categories of objects: (1) single-writer, multi-reader objects like product descriptions and prices; (2) multi-writer, single-reader objects like lists of orders; and (3) commutative-write, approximate-read objects like the current inventory count of each product.

However, a key limitation of our previous efforts to support edge services was our decision to use weak consistency—and thereby introduce considerable complexity—for a fourth category of objects: multi-writer, multi-reader objects such as per-customer *profile* information (e.g., name, account number, recent orders, credit card number, and address.) We, like several other systems [95, 114, 136], made use of a Read-One, Write-All-Asynchronously (ROWA-A) protocol based on local reads and asynchronous epidemic propagation of writes. ROWA-A protocols provide excellent read performance and availability; and although ROWA-A protocols allow applications to observe inconsistencies between reads and writes, such inconsistencies should be rare because multi-reader, multi-writer shared objects often have workloads with low concurrency to any given object. For example, in our edge-server TPC-W application, reads and writes to a given customer's profile typically come from just one edge server for some interval of time, until the customer is redirected to a different server. Unfortunately, although inconsistencies are rare for the workloads of interest, these rare cases introduce considerable complexity into the design, because all cases must be handled no matter how rare they are and because reasoning

about corner cases in consistency protocols is complex. Furthermore, because reads can always complete locally, these protocols provide no worst-case bound on staleness, i.e., it is possible for a read to return stale data arbitrarily long after a write, which can be unacceptable for some applications.

By introducing dual-quorum replication, this chapter provides the key missing piece to achieve highly-available, low-latency, and consistent data replication for a range of Internet services. In particular, dual-quorum replication optimizes these properties for data elements that can be both read and written from many locations, but whose reads and writes exhibit locality in two dimensions: (1) at any given time access to a given element tends to come from a single node and (2) reads tend to be followed by other reads and writes tend to be followed by other writes. For other workloads, our algorithm continues to provide regular consistency semantics [71], but its performance and availability may degrade.

Our dual-quorum replication protocol combines ideas from volume leases [132] with quorum based techniques [45, 46]. The protocol employs two quorum systems, an input quorum system (IQS) and an output quorum system (OQS). Clients send their writes to the IQS and they read from the OQS. The two quorum systems communicate with each other when necessary to synchronize the state of replicated objects. By using two quorum systems, we are able to optimize construction of the OQS's read quorum to provide low latency and high availability for reads while optimizing construction of the IQS's write quorum to provide modest overhead and high availability for writes. In particular, OQS nodes cache data from the IQS servers using a quorum-based generalization of Yin et al.'s volume lease protocol [132], which invalidates individual cached objects as they are updated. The protocol uses short-duration volume leases to allow writes to complete despite network partitions and aggregates these leases across large numbers of objects in a volume to amortize the cost of renewing short leases. Using our dual-quorum protocol, workloads with large numbers of repeated reads (or writes) perform well because reads (or

writes) can often be supplied by a read-optimized *OQS* read quorum (or write-optimized *IQS* write quorum) without requiring communication with the *IQS* (or *OQS*).

Through both analytical and experimental evaluations, we compare the availability, response time, communication overhead, and consistency guarantees of the dual-quorum protocol against other popular replication protocols: the synchronous and asynchronous Read-One/Write-All (ROWA) protocol family, [1] majority quorum system, and grid quorum system [30]. For the important special case of single-node *OQS* read quorum, average read response time can approach a node's local read time, making the read performance of this approach competitive with ROWA-A epidemic algorithms such as Bayou [114]. But, the dual quorum approach avoids suffering the weak consistency guarantees and resulting complexity inherent in ROWA-A designs. Additionally, analytical evaluations show that the overall availability of the dual-quorum protocol is competitive with the majority quorum protocol for the targeted workloads. Finally, for the targeted workloads, the communication overheads of this approach are comparable with existing approaches. However, in the worst-case scenario in which the workload consists of only interleaved reads and writes, the dual-quorum protocol requires significantly more message exchanges than traditional quorum protocols to coordinate internal nodes.

The main contribution of this work is to introduce the dual-quorum algorithm, a novel data replication algorithm targeted to a key workload for Internet edge service environments. Note that although our work is motivated by a specific replication scenario, we speculate that it will be more generally useful. In particular, we believe that it may not be uncommon for systems that can, in principle, have any node read or write any item of data to, in practice, experience sufficient locality to benefit from our approach.

[1] Note that ROWA protocols are, in fact, a special case of quorum protocols, but they are often treated separately in the literature [13, 14].

4.2 System Model and Definitions

Our edge service environment consists of a collection of edge server nodes that each play one or more of the following three roles: (a) *front end* nodes that handle *service client* requests from across the Internet, execute application-specific processing, and act as *edge server clients* or just *clients* to the dual-quorum storage system; (b) *Output Quorum System* (*OQS*) nodes that process client read requests; and (c) *Input Quorum System* (*IQS*) nodes that process client write requests. We assume a *request redirection architecture* that directs clients to a good (e.g., nearby, lightly loaded, or available) front end edge server; a number of suitable redirection systems are discussed in the literature [62, 133]. Note that clients are unaware of the underlying data storage system and never contact the *OQS* or *IQS* interfaces directly.

In an edge service environment, servers typically process sensitive or valuable information, so they must run on trusted machines such as dedicated servers in a hosting center. We therefore assume a fail-stop model in which servers may crash but cannot issue incorrect requests or replies. The network may delay, duplicate, or reorder messages. We assume secure communication among nodes and that if the network corrupts a message, this corruption is detected by low-level checksums and the message is silently discarded. Each node can read a local real-time clock and that there exists a maximum drift rate *maxDrift* (ϵ) between any pair of clocks. Our protocol ensures safety regardless of timing assumptions: nodes may operate at arbitrarily different speeds and we require no bound on message delivery delay. However, long processing times or message delays may interfere with liveness for some requests. In particular, if machine A requests a least at time t_0 and receives a reply at time t_1 from node B granting a least of length T, then A conservatively expires the lease at time $t_0 + (1 - \epsilon)T$; this approach ensures that the receiver of a lease (A) expires the lease no later then the grantor of the lease (B).

For performance, our system assumes that concurrent reads and writes to a given object by different nodes are rare. But, for correctness, we must define the system's

consistency semantics in the presence of concurrent reads and writes to the same object. The dual quorum design provides *regular* semantics [71]: a read r that is not concurrent with any write returns the value of the latest write that completed before r began, and a read r that is concurrent with one or more writes returns one of (a) the value of the last write that completed before r began, or (b) the value of one of the writes concurrent with r.

For convenience of exposition, we describe interactions with a quorum system in terms of a QRPC operation [77]. $replies = QRPC(system, READ/WRITE, request)$ sends *request* to a collection of nodes in the specified quorum *system* (e.g., the IQS or OQS). The QRPC call then blocks until a set of *replies* constituting the specified quorum ($READ$ or $WRITE$) on the specified *system* have been gathered. The call then returns the set of *replies* that it received. The QRPC operator abstracts away details of selecting a quorum, retransmissions, and timeouts. In particular, different implementations may choose different ways to select which nodes from *system* to send requests to, and they may select different retransmission strategies: our simple prototype implementation always transmits requests to the local node if the local node is a member of *system*; it then randomly selects a sufficient number of additional nodes to form a $READ$ or $WRITE$ quorum and transmits the request to them; retransmissions are each to a new randomly selected quorum using an exponentially-increasing retransmission interval. A more aggressive implementation might send to all nodes in *system* and return when the fastest quorum has responded or might track which nodes have responded quickly in the past and first try sending to them. To simplify the discussion of the protocol, we refer to a server holding an invalid object as an *invalid server.* Since there multiple object exists in the system, the status of a server may be considered invalid with respect to one object while being considered valid with respect to other objects. Similarly, a quorum *only* containing invalid servers with respect to a particular object is considered an *invalid quorum* with respect to this object. Note that a quorum is *valid* as long as it includes at lease one valid server.

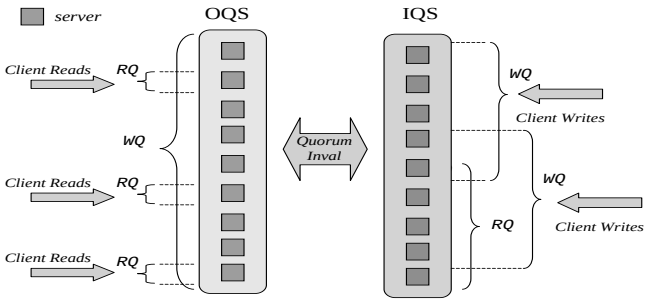


Figure 4.2: Dual quorum architecture overview.

4.3 Dual Quorum Protocol Design

This section describes the design of the dual-quorum replication system and the key ideas for achieving our design goals. The basic idea is to separate the read and write quorum into two quorum systems so that they can be optimized individually to improve response time and availability for read-dominated or write-dominated workloads. The read and write quorum of the *OQS* and *IQS* can be separately configured in any way desired, but we would expect one common configuration to be to optimize read performance by having the *OQS* span all nodes in the system with a read quorum size of 1 and to get good write availability by having the *IQS* span a modest number of nodes with any majority of the *IQS* nodes forming a write quorum. As Figure 4.2 illustrates, in the dual quorum system clients retrieve objects from a read quorum (perhaps containing only one node) in *OQS* and send object updates to a write quorum (typically containing a majority of nodes) in *IQS*. (In the context of the edge service architecture described in Figure 4.1, all dual-quorum servers are edge servers.) The two quorum systems conditionally synchronize with each other to maintain the consistency of data replicated on them when processing both reads and writes.

To simplify the discussion, we present the protocol in two steps. First, we will discuss the basic dual-quorum protocol, a simplified asynchronous protocol, in Section 4.3.1. This protocol allows separate optimizations of read and write quorum, but because it assumes an asynchronous system model, a write can block for an arbitrarily long period of time. Then,

in Section 4.3.2 we describe how we introduce volume leases to improve write availability while retaining good read performance.

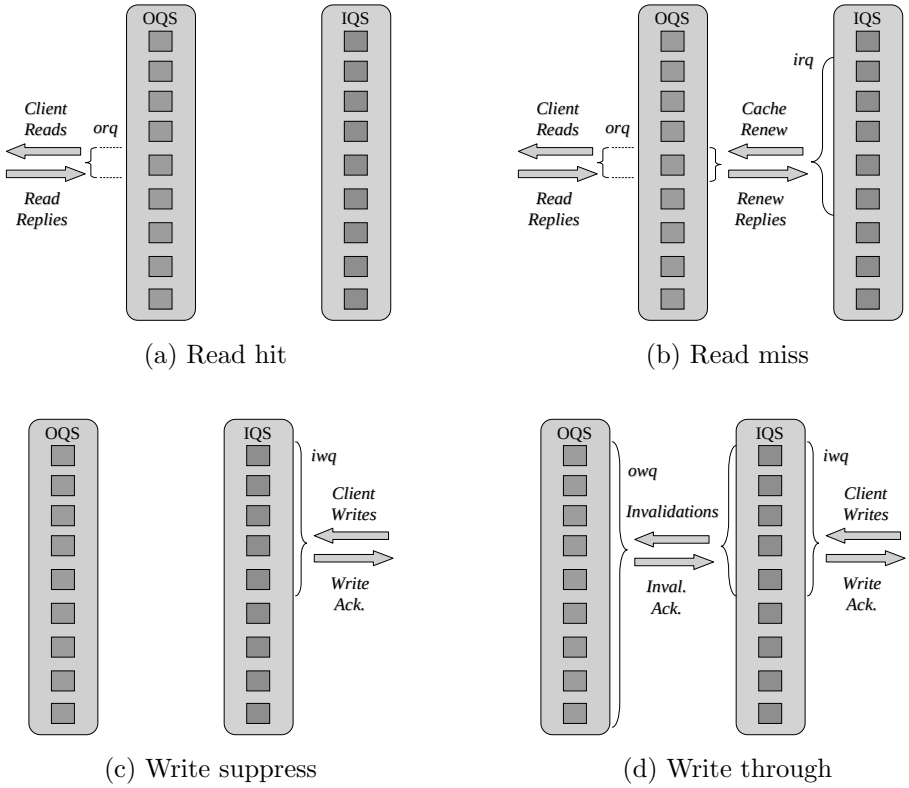
4.3.1 Dual quorum protocol

High level overview The basic idea of the dual quorum replication is to process reads and writes in two different quorum systems, *IQS* and *OQS* , and use a cache invalidation strategy to synchronize the state of objects replicated in *IQS* nodes and cached in *OQS* nodes.

Clients[2] perform similar tasks for reading and writing data as in the conventional quorum based protocols. When a client read arrives in *OQS* , two possible scenarios can happen, as illustrated in Figure 4.3 (a) and (b). In a *read hit* case, the *OQS* read quorum contains a valid cache copy of the requested object, which is immediately sent back to the client. When there is a *read miss*, i.e. the cache copy on the *OQS* read quorum is invalid, the *OQS* read quorum validates the cache copy by querying an *IQS* read quorum for the latest update. Once the cache copy of the *OQS* read quorum is validated, the *OQS* read quorum sends the updated value to the client. There are also two scenarios when processing client writes, as illustrated in Figure 4.3 (c) and (d). In a *write suppress* case, the cache copy in an *OQS* write quorum is already invalid. the *IQS* write quorum can just apply the write to the local object and send the completion acknowledgment to the client. In the case of a *write through*, no *OQS* write quorum holds invalid cache copy. Therefore, the *IQS* write quorum that receives the client write has to invalidate the cache copy on one *OQS* write quorum before the write can complete.

For workloads consisting of read bursts, the first read forces all *OQS* nodes of the read quorum to validate their cached copies. Therefore, all subsequent reads are *read hits*. Once we configure the *OQS* read quorum to contain only one node, reads becomes local. Therefore, the protocol yields optimal read response time and availability for such

[2]Client reads and writes come from edge-server clients of the dual-quorum system not form untrusted services clients.



(a) Read hit (b) Read miss

(c) Write suppress (d) Write through

Figure 4.3: Request processing scenarios

workloads. For workloads consisting of write bursts of the same data, the first write invalidates cached copies in an *OQS* write quorum, making all subsequent writes as *write throughs*. Naturally, we can configure *IQS* as a majority quorum system to provide the optimal write availability for such workloads.

Protocol details The following paragraphs provide the details of the basic dual-quorum protocol by describing the actions taken at individual nodes.

Data structures. Each IQS node maintains the following state for each object o: $lastWriteLC_o$ stores the logical clock of the last write to o, $lastReadLC_o$ stores the value of $lastWriteLC_o$ from the time of the last read of o, $lastAckLC_{o,n}$ stores the logical clock contained in the highest invalidation reply from node n for o, and $value_o$ stores the value of o. Each node in IQS maintains a logical clock $logicalClock$ whose value is always at least as large as the

node's largest $lastWriteLC_o$ for any object o. Each node in OQS maintains the following per-object o per-node n state: $epoch_{o,n}$ indicates the last epoch for which a valid object lease on o from n was held, $logicalClock_{o,n}$ indicates the highest version number (logical clock) of o for which an invalidation or update has been received from n, and $valid_{o,n}$ is true if $logicalClock_{o,n}$ corresponds to an update (false if it corresponds to an invalidate). Finally $value_o$ stores the update body for the highest logical clock received in any update message for o from any node.

Object validity. The system maintains the following key invariant: If node j in OQS has from node i in IQS a valid object o ($j.valid_{o,i}$) then node i in IQS knows node j in OQS has a valid object callback ($i.lastReadLC_o > i.lastAckLC_{o,j}$).

Client read. From the client's point of view, a dual-quorum read is the same as a standard quorum read [45, 46]. *client* sends a read request to the OQS via $QRPC$. After receiving replies from a read quorum in OQS, *client* selects the value with the highest logical clock.

A node j in OQS that receives a client read request checks whether the object o is valid. j performs the check by verifying locally if there exists a read quorum irq in IQS such that j holds a valid object from every node i in this read quorum (i.e. $valid_{o,i} = TRUE$ for all i in a read quorum irq). o is valid if a read quorum irq satisfying the condition can be verified, invalid otherwise. If o is valid, j returns the object's locally-stored logical clock and value. If not, j renews the object by sending object renewal messages to IQS using $QRPC$. After receiving replies R from a read quorum in IQS, j updates its local state ($\forall i,\ s.t.\ i \in R$: if $R.r_{o,i}.lc \geq logicalClock_{o,i}$, then $logicalClock_{o,i} := R.r_{o,i}.lc$ and $valid_{o,i} := true$). Then, j updates $value_o$ with the value in the reply with the highest logical clock and returns both the value $value_o$ and the highest logical clock to the client.

Each IQS server that receives an object renewal message returns to the OQS server $value_o$ and $lastWriteLC_o$ and then updates $lastReadLC_o = max(lastReadLC_o, lastWriteLC_o)$.

Client write. Just like the standard quorum write protocol [45, 46], *client* first queries IQS using $QRPC$ to retrieve the highest logical clock from a read quorum in IQS. Next, *client* advances the logical clock and embeds it in the write request that is then sent to the IQS via $QRPC$. The write completes after *client* receives acknowledgments from a write quorum in IQS.

An IQS server i that receives a client request for the highest logical clock of the last completed write responds with its logical clock $logicalClock$. When i receives a client write whose logical clock is larger than that associated with the last completed write of o on i ($lastWriteLC_o$), i updates $lastWriteLC_o$ and $value_o$ with those in the write. Then, to ensure that a write quorum in OQS is unable to read the old version of the data, i performs one of the following tasks: (a) if no OQS server has renewed since the completion of the last write, (e.g. $\forall j,\ s.t.\ j \in OQS,\ lastReadLC_o < lastAckLC_{o,j}$), i suppresses invalidations to OQS; (b) otherwise, i sends invalidations with the logical clock of the write to OQS using $QRPC$. The write completes after receiving invalidation replies from a write quorum in OQS, at which point i updates $lastAckLC_{o,j}$ for all j in the $QRPC$ reply and returns to the client.

An OQS server j that receives from node i in IQS an invalidation with a logical clock $lc_{o,i}$ compares $lc_{o,i}$ with $logicalClock_{o,i}$. If the invalidation has the higher logical clock, j updates the local state ($logicalClock_{o,i} = lc_{o,i}$ and $valid_{o,i} = false$). Finally, j sends an invalidation acknowledgment back to i.

4.3.2 Dual quorum with volume leases

The basic protocol just described allows one to vary read and write quorum sizes independently. However, our application would benefit from using a read quorum size of 1 so that reads can be serviced locally; any larger read quorum size introduces a network delay to every read and provides qualitatively worse read response time. However, a read quorum size of 1 could lead to unacceptable write availability because it requires a write

```
1   processLCReadRequest(){
2       sendMsg(CLIENT_LC_READ_REPLY, logicalClock);
3   }
4
5   processWriteRequest(Object o, Value v, LogicalClock lc){
6       if(lc > lastWriteLC_o){
7           value_o := v;
8           lastWriteLC_o := lc;
9           // ensure an OQS write quorum is invalidated
10          while(!isOWQInvalid(o, lc)){
11              invalidateOWQ(o, lc); // see text for
                    descriptions
12          }
13      }
14      sendMsg(CLIENT_WRITE_ACK, o, lc);
15  }
16
17  processInvalAck(Object o, Sender j, LogicalClock lc){
18      // update the last inval ack in the record for the
            sender
19      lastAckLC_{o,j} := MAX(lastAckLC_{o,j}, lc);
20  }

20  processVLRenewal(Volume v, Sender j, RequestorTime
        t_{v,0}){
21      expires_{v,j} := L + currentTime;
22      sendMsg(VOLUME_RENEW_REPLY, delayed_{v,j}, L, epoch_{v,j}
            , t_{v,0});
23  }
24
25  processVLRenewalAck(Volume v, Sender j, LogicalC lc)
        {
26      // remove delayed invals already applied at the
            sender
27      ∀k, s.t. inval_{k,j} ∈ delayed_{v,j} {
28          if(lc ≥ inval_{k,j}.lc){
29              delete inval_{k,j};
30          }
31      }
32  }
33
34  processObjRenewal(Object o){
35      // update the last read logical clock
36      lastReadLC_o := lastWriteLCo;
37      sendMsg(OBJECT_RENEW_REPLY, value_o, lastWriteLC_o)
            ;
38  }
```

Figure 4.4: IQS server operations (pseudocode) - Dual quorum with volume leases

to successfully contact all nodes in the OQS to invalidate cached data. We therefore adapt Yin et al.'s volume lease protocol [132] to support very small read quorums in OQS while retaining acceptable availability on writes.

High level overview We group objects into collections called volumes. To process a read, a read quorum in *OQS* must hold both a valid *volume lease* and a valid *object lease* for some read quorum in IQS. A lease represents permission to access some object that expires at some specified time [47]. Similar to the basic dual quorum protocol described in the previous section, when an *OQS* read quorum holds both valid leases, all client reads processed by this read quorum are *read hit*. A *read miss* implies that either or both leases are invalid; they can be renewed by querying from an *IQS* read quorum. Similarly, a *write suppress* is when either or both leases are invalid in at least one *OQS* write quorum. To process a write in the *write through* scenario, the *IQS* write quorum can (a) invalidate the object lease in an *OQS* write quorum or (b) wait for the lease to expire on the volume containing the requested object in an *OQS* write quorum. This second option is what allows the system to function well even with very small *OQS* read quorum.

```
1  processVLRenewReply(Volume v, Sender i, Lease L, Epoch e, DI di, RequestorTime t_{v,0}){
2      expires_{v,i} := MAX(expires_{v,i}, t_{v,0} + L * (1 - maxDrift));
3      epoch_{v,i} := MAX(epoch_{v,i}, e);
4      // apply delayed invals if exist in the reply
5      ∀k, s.t. inval_{k,i} ∈ di {
6          if(inval_{k,i}.lc > logicalClock_{k,i}){
7              logicalClock_{k,i} := inval_{k,i}.lc;
8              valid_{k,i} := false;
9          }
10     }
11     sendMsg(VOLUME_RENEW_REPLY_ACK, v, MAX(di.lc));
12 }
13
14 processInval(Object o, Sender i, LogicalClock lc){
15     // update the local logic clock and object status
16     if(logicalClock_{o,i} < lc){
17         logicalClock_{o,i} := lc;
18         valid_{o,i} := false
19     }
20     sendMsg(INVAL_ACK, lc);
21 }

21 processReadRequest(Object o){
22     // ensure the local object and volume are valid
23     while(!isLocalValid(o)){
24         // if not, renew invalid volume or object or both
25         validateLocal(o);
26     }
27     // once both leases are validated, send reply to cilent
28     lc := MAX_{∀i, s.t. value_{o,i}=true}(logicalClock_{o,i});
29     sendMsg(CLIENT_READ_REPLY, value_o, lc);
30 }
31
32 processRenewReply(Object o, Sender i, Epoch epoch, LogicalClock lc, ObjectValue value){
33     epoch_{o,i} := MAX(epoch_{o,i}, epoch);
34     if(logicalClock_{o,i} ≤ lc){
35         logicalClock_{o,i} := lc;
36         valid_{o,i} := true;
37     }
38     if(valid_{o,i} = true &&
           logicClock_{o,i} ≥ MAX_{∀k, s.t. k∈IQS}(logicalClock_{o,k})){
39         value_o := value;
40     }
41 }
```

Figure 4.5: OQS server operations (pseudocode) - Dual quorum with volume leases

The key benefit of volume leases is that they are of short duration while object leases are of long duration.[3] This combination yields good read response time and high availability for systems with small *OQS* read quorum; nodes in *OQS* can cache objects locally for a long time, and although they must frequently renew volume leases, the cost is amortized across a large number of objects in a volume [132]. At the same time, the combination does not suffer from poor write availability although *OQS* write quorum is large: a write that cannot contact all nodes in an *OQS* write quorum needs only wait for the (short) volume lease to expire.

The key challenge in introducing volume leases is to manage the callback state when invalidations are suppressed at *IQS* when the volume lease expires in an *OQS* write quorum. When an *IQS* write quorum processes a write to o while the lease has expired for the volume v containing o in an *OQS* write quorum, i.e. a *write suppress* scenario, the *IQS* write quorum enqueues the invalidation of o as a *delayed invalidation* [132]. The delayed invalidation of o must be processed by an *OQS* write quorum before v's lease can

[3]For simplicity, we will assume infinite-length object leases or *callbacks* [57]. Generalizing to finite-length object leases is straightforward and can help optimize space and network costs [39].

be renewed so that a callback to *IQS* is installed on an *OQS* write quorum. The callback ensures *OQS* to query from *IQS* for the update to o when o is requested at *OQS* .

A final implementation detail we take from Yin et al. [132] is to bound the size of the list of delayed invalidations for *OQS* using *epochs*. Volume lease renewals are marked with an epoch number, and when this epoch number changes, *OQS* conservatively assumes all object callbacks have been revoked by *IQS* . In this case, *OQS* suspects that all objects under this volume are updated at *IQS* and *OQS* needs to query an *IQS* read quorum to validate the cache copy before sending any object to clients.

Protocol details The protocol details at the node level are similar to the basic dual quorum protocol except that each *IQS* node tracks the volume lease and callback state on all *OQS* nodes in addition to the status of the cache copies. The pseudo-code describing actions at an IQS and an OQS node is shown in Figures 4.4 and 4.5.

Data structures. Each node in IQS maintains a real time clock $currentTime$ (with bounded drift with respect to the other clocks as described in Section 4.2) and a logical clock $logicalClock$. Each IQS node also maintains the following per-volume v, per-OQS-node j state: $expires_{v,j}$ which indicates when v expires at j, $delayed_{v,j}$ which contains a list of delayed invalidations that must be delivered to j before v is renewed, and $epoch_{v,j}$ which indicates j's current epoch number for v. Finally, each IQS node maintains the following per-object o state: $lastWriteLC_o$ stores the logical clock of the last write to o, $lastReadLC_o$ stores the value of $lastWriteLC_o$ from the time of the last read of o, $lastAckLC_{o,j}$ stores the logical clock contained in the highest invalidation reply from node j for o, and $value_o$ stores the value of o.

Each node in OQS maintains a bounded-drift real time clock $currentTime$. In addition, it maintains the following per-volume v per-IQS-node i state: $epoch_{v,i}$ is the highest epoch number for which a valid volume lease from i was held on v and $expires_{v,i}$ is the time when the lease on v from i will expire. And, it maintains the following per-object

o per-IQS-node i state: $epoch_{o,i}$ indicates the last epoch for which a valid object lease on o from i was held, $logicalClock_{o,i}$ indicates the highest version number (logical clock) of o for which an invalidation or update has been received from i, and $valid_{o,i}$ is true if $logicalClock_{o,i}$ corresponds to an update (false if it corresponds to an invalidate). Finally $value_o$ stores the update body for the highest logical clock received in any update message for o from any node.

Volume and object validity. The system maintains the following key invariant: If node j in OQS has from node i in IQS both a valid volume v ($expires_{v,i} > currentTime$) and a valid object o ($epoch_{v,i} = epoch_{o,i}$ && $valid_{o,i}$) then node i in IQS knows node j in OQS has a valid volume lease ($expires_{v,j} > currentTime$) and valid object callback ($lastReadLC_o > lastAckLC_{o,j}$).

Client read. As detailed by **processReadRequest** in the pseudo-code, a node j in OQS processes a client read of object o as follows. j must ensure Condition C: there exists a read quorum irq in IQS such that j holds both a valid volume lease and valid object lease from irq. If C is already true, then j can immediately return the value $value_o$ and the associated logical clock $MAX_{\forall i,\, s.t.\, i \in IQS}(logicalClock_{o,i})$.

If C is not true, then j performs a variation on QRPC. QRPC as defined in Section 4.2 sends and resends a request to different nodes until it receives a quorum of replies. This variation sends *different* requests to different nodes and processes replies until condition C becomes true. In particular, for each target node i selected, j sends one of three things: (a) if the volume from i has expired and the object from i is invalid, it sends a combined volume renewal and object read; (b) if just the volume has expired, it sends a volume renewal; or (c) if just the object is invalid, it sends an object read. As detailed in the pseudo-code **processVLRenewReply**, j processes replies to volume renewal requests from IQS node i by applying the delayed invalidations included in the reply (in the same way as applying normal invalidations as described below) and updating

$expires_{v,i}$ as well as $epoch_{v,i}$. To account for worst-case clock drift, j conservatively sets $expires_{v,i} = t_o + L*(1 - maxDrift)$ where t_o is the time that j *sent* the volume lease renewal request, L is the volume lease length granted in the reply, and *maxDrift* is as defined in Section 4.2. Finally, j sends i a volume lease renewal acknowledgment (which i uses to clear its delayed invalidation queue.) As detailed in the pseudo-code **processRenewReply**, j processes object renewal replies from i by updating $epoch_{o,i}$, $logicalClock_{o,i}$, and $valid_{o,i}$; furthermore, if $valid_{o,i}$ is true and $logicalClock_{o,i}$ exceeds the logical clock of any other *valid* logical clock for this object, j updates $value_o$. The repeated sends and the processing of replies in this QRPC variation ensure that C eventually becomes true, at which point j returns $value_o$ and the associated logical clock ($logicalClock_{o,i_{max}}$) as the result of the read.

On the IQS side, node i in IQS processes volume renewal messages for volume v from node j as described in the pseudo-code **processVLRenewal**: i sends the delayed invalidations $delayed_{v,j}$ and the volume renewal, containing the epoch number $epoch_{v,n}$ and lease length L. i then records the volume expiration time ($expires_{v,j} = L + currentTime$). When i receives a volume lease renewal acknowledgment for volume v and logical clock lc from j, as detailed in the pseudo-code **processVLRenewalAck**, i clears all delayed invalidations with logical clocks up to lc from $delayed_{v,j}$. As **processObjRenewal** indicates, when i in IQS processes a read of object o from OQS node j, it replies with $value_o$ and $lastWriteLC_o$ and updates $lastReadLC_o = lastWriteLC_o$. Note that $lastReadLC_o$, $lastAckLC_{o,j}$, and $lastWriteLC_o$ allow i in IQS to track which nodes j in OQS may hold valid object callbacks. Finally, if an IQS server i wishes to garbage collect delayed invalidation state for j, i advances $epoch_{v,j}$ and and deletes the delayed invalidations $delayed_{v,j}$. Note that if j receives from i a volume lease with a new epoch, then $epoch_{v,i} \neq epoch_{o,i}$ for all o. So all previously valid object leases from i immediately become invalid. Thus, if j misses some object invalidations from i when its volume lease from i has expired, a volume lease renewal from i can resynchronize j's state by either (a) updating $valid_{o,i}$ with the

missing delayed invalidations or (b) advancing $epoch_{v,i}$ by sending a volume renewal with a new epoch number.

Client write. A client first determines the highest logical clock of any completed write by calling IQS's **processLCReadRequest**. A node i in IQS responds to such a call for object o by returning the node's *global* logical clock $logicalClock$. A client then issues the actual write of object o. As detailed in **processWriteRequest** in the pseudo-code, if the write's logical clock exceeds that of the highest write seen so far ($lastWriteLC_o$), node i stores the write's logical clock and value. i then ensures that a write quorum in OQS is unable to read the old version of the data by performing a variation on QRPC that "sends" differently to different nodes depending on whether their volume and object leases are valid. There are three cases for i to consider for node j, object o, and volume v: (a) if i knows o is invalid at j (e.g., $lastReadLC_o < lastAckLC_{o,j}$) then i need take no action for j; (b) otherwise if o is valid at j but v is invalid at j (e.g., $expires_{v,j} < currentTime$) then i enqueues an invalidation in $delayed_{v,j}$ which will be processed at j when it renews its volume; or (c) both the object and volume are valid (e.g., $lastReadLC_o > lastInvalLC_{o,j}$) then j sends an object invalidation containing the write's logical clock ($lastWriteLC_o$) to j. In this last case, if j receives an invalidation from i for object o with logical clock lc, then as the pseudo-code in **processInval** describes, j applies the invalidation: if the invalidation is the newest information about o from i (e.g., $lc > logicalClock_{o,i}$) then update the logical clock and validity information ($\{logicalClock_{o,i} = lc; valid_i = false\}$). Finally, if i receives an invalidation-acknowledgment from j for logical clock lc, then as the pseudo-code in **processClientInvalAck** describes, i updates $lastAckLC_{o,j} = max(lastAckLC_{o,j}, lc)$.

4.3.3 Correctness

In this section, we prove that the dual-quorum protocol provides Pseudo-Regular Semantics [96]:

- Property 1: A read of o that is not concurrent with any writes of o can return only the value and logical clock from the completed write of o with the highest logical clock and
- Property 2: A read of o that is concurrent with one or more writes of o can (a) return the value and logical clock from the completed write of o with the highest logical clock, (b) return the value and logical clock from some concurrent write of o, or (c) keep retrying.

Proof of Dual-Quorum Protocol

We refer to a server in OQS Sr_{OQS} and a server in IQS Sr_{IQS}. When a write to o with timestamp n completes on a Sr_{IQS}, the Sr_{IQS} must have received $lastAckLC_{o,j}$ with timestamp n from every Sr_{OQS} j of an owq.

Lemma 1 After a write with timestamp n completes on a Sr_{IQS}, no subsequent reads return a value with timestamp lower than n.

Proof: Receiving $lastAckLC_{o,j}$ with timestamp n from every server j of an owq implies that at least one server in any orq contains the invalidation with timestamp n. Any subsequent reads will be sent to at least one OQS server holding the invalidation with timestamp n which prevents any value with timestamps lower than n from returning by subsequent reads.

Lemma 2 If $lastAckLC_{o,j} > lastReadLC_o, \forall j \in OQS$ on every server of an iwq and the timestamp of the last completed write is n, no subsequent reads return a value with timestamp lower than n.

Proof: When $lastAckLC_{o,j} > lastReadLC_o, \forall j \in OQS$ on a Sr_{IQS}, it implies that no Sr_{OQS} has renewed from this Sr_{IQS} since the last write completed on this Sr_{IQS}. (1) When $lastAckLC_{o,j} > lastReadLC_o, \forall j \in OQS$ on all Sr_{IQS} of an iwq, it implies that no

Sr_{OQS} has renewed from any Sr_{IQS} of this iwq since the last write, the write with timestamp n, completes on every Sr_{IQS} of this iwq. Because an iwq intersects with all irq, we can also conclude that no Sr_{OQS} has renewed from any irq since the last write completed on this iwq. (2) From the proof of Lemma 1 we know that after any Sr_{IQS} completes the last write, at least one server in any orq contains an invalidation. From (1) and (2) we conclude the following: When $lastAckLC_{o,j} > lastReadLC_o, \forall j \in OQS$ on all Sr_{IQS} of an iwq, at least one server in any orq received an invalidation and has not yet renewed from any irq. Therefore, when a read arrives at an orq, invalid servers (at least one) of this orq will read from an irq. If the last completed write in the system has timestamp n, the irq will not return any value with timestamp lower than n. Consequently, a subsequent read will return a value with timestamp no lower than n. Similarly, subsequent reads arriving at any orq will also return a value with timestamp no lower than n.

Property 1: When a write with timestamp n arrives at an iwq, every server of the iwq performs one of the two tasks: 1. If $lastAckLC_{o,j} > lastReadLC_o, \forall j \in OQS$ is true for any server of the iwq, it applies the write locally and sends a completion acknowledgment to the client. 2. Otherwise, the server attempts to send invalidations to all Sr_{OQS} j of an owq and updates its $lastAckLC_{o,j}$ to n upon receiving the acknowledgment from each j in the owq. If all servers of the iwq perform only task 1, every read after the write with timestamp n completes return values with timestamp no lower than n according to Lemma 2. If any server of the iwq performs task 2, all reads after the write with timestamp n completes return values with timestamp no lower than n according to Lemma 1.

Property 2: Assume the last completed write has timestamp $n - 1$. As a write with timestamp n and a read arrive in the system at the same time, there are two cases to consider. Case 1, if $lastAckLC_{o,j} > lastReadLC_o, \forall j \in OQS$ on every server of an iwq, the read causes at least one Sr_{OQS} to renew from an irq according to the proof of Lemma

2. Then we have a situation where both the renewal and the write are active in the IQS. Because IQS provides Regular Semantics, the renewal could return a value with either $n-1$ or n. Therefore, the read can return either $n-1$ or n to the client. Case 2, if there does not exist an iwq with all its servers satisfying $lastAckLC_{o,j} > lastReadLC_o, \forall j \in OQS$, invalid servers in some orq may have renewed from an irq and received the value with timestamp $n-1$ due to some previous reads. If this concurrent read touches one of those orq validated by previous reads, it returns $n-1$ as previous reads. If this concurrent read touches an orq with at least one invalid Sr_{OQS}, the result is the same as in Case 1. In both cases if an invalid Sr_{OQS} receives the invalidation with timestamp n followed by a renew reply containing the value with timestamp $n-1$, it has to retry from some irq until the value with timestamp n is returned. In conclusion, a read that is concurrent with any write will return the value of the last completed write, the value of the concurrent write, or it retries to get the value of the concurrent write.

Proof of Dual-Quorum Protocol with Volume Leases

Each Sr_{IQS} has a $nonExpSet_v$ that contains all Sr_{OQS} j with valid leases of volume v (i.e. $expires_{v,j} > currentTime$).

Lemma 4 If v's lease has not expired on a Sr_{OQS} j, $expires_{v,j} > currentTime$ on at least one Sr_{IQS} in any iwq.

Proof: When the Sr_{OQS} j obtains a new lease for v, j needs to query an irq. As j sets v's new expiration time to the minimal expiration time returned from the irq, every server i of the irq records the expiration time it sends to j in $expires_{v.j}$ which is equivalent to or larger than the expiration time held by j. Since the irq intersects with all iwq, at least one server, i, of any iwq belongs to the irq. Therefore, as long as v is not expired on j, $expires_{v,j} > currentTime$ on at least one server in any iwq.

Lemma 5 If $nonExpSet_v$ is an empty set on every Sr_{IQS} of an iwq, v must have expired on every server of an owq.

Proof: The counter-positive of Lemma 4 states that if every Sr_{IQS} of an iwq contains $expires_{v.j} <= currentTime$ for the same Sr_{OQS} j, $j'v$ must have expired. When $nonExpSet_v$ is an empty set on every server of the iwq, no Sr_{OQS} contains a non-expired v. Therefore, v must have expired on every Sr_{OQS} of an owq.

Property 1: When a write to o in volume v with timestamp n arrives at an iwq, every Sr_{IQS} of the iwq performs one of the three tasks: 1. If $nonExpSet_v$ is an empty set on a Sr_{IQS}, it applies the write locally, buffers the invalidation with timestamp n on the Sr_{IQS}, and sends a completion acknowledgment to the client. 2. If $lastAckLC_{o,j} > lastReadLC_o$ is true on a Sr_{IQS}, it applies the write locally and sends a completion acknowledgment to the client. 3. Otherwise, the Sr_{IQS} attempts to send invalidations to all Sr_{OQS} j of an owq and updates its $lastAckLC_{o,j}$ to n upon receiving each acknowledgment from each j in the owq. If every Sr_{IQS} performs only task 1, we know that the volume must have expired on every Sr_{OQS} of an owq according to Lemma 5. Because an owq intersects with all orq, any subsequent read arrives on at least one Sr_{OQS} with expired volume lease. The Sr_{OQS} with expired volume lease will renew its volume from an irq. At least one Sr_{IQS} that buffers the invalidation with timestamp n is queried. Therefore, the invalidation with timestamp n is returned to the Sr_{OQS} as it receives its new volume lease. Because the Sr_{OQS} contains an invalidation with timestamp n, it prevents the read from returning a value with timestamp less than n to the client. The rest of the proof is the same as the proof of Property 1 for the Dual-Quorum protocol.

Property 2: The proof of property 2 is similar to that for the dual-quorum protocol. Case 1, the condition changes from "$lastAckLC_{o,j} > lastReadLC_o$" to "$lastAckLC_{o,j} > lastReadLC_o$ or $nonExpSet = \emptyset$" for every server of an iwq. Case 2, the condition

changes from "$lastAckLC_{o,j} > lastReadLC_o$" to "$lastAckLC_{o,j} > lastReadLC_o$ and $nonExpSet = \emptyset$"

4.4 Evaluation

Through both analytical and experimental evaluations, we compare the availability, performance, and communication overhead of DQVL against other popular replication protocols. We show that DQVL yields read performance competitive with ROWA-A epidemic algorithms and overall availability competitive with the majority quorum protocol.

4.4.1 Response time

Analytical evaluation First, we analyze the response time of DQVL and make comparisons with other popular protocols in the context of the edge service environment where every client connects to a nearby edge server via a fast connection, e.g. a LAN-like connection, *lan*, with 6 ms RTT. All edge servers connect to each other through an overlay network, *overlay*, with RTT delays of 80 ms. For a client to connect to servers other than its nearby edge server, it has to go through a WAN-like connection, *wan*, with 86 ms RTT.

To preserve the optimal availability, the IQS is configured as a majority quorum system so that a client needs to contact more than its nearby edge server to complete a write. But the read quorum in OQS can be configured to consist of one node so that a client needs to read only from its nearby server. Therefore, the response time of a *read hit* will only involve *lan* delays. But the response time of a *read miss* is $lan + overlay$ because this closest server needs to renew from other edge servers. The response time of *write suppress* is $wan * 2$, one trip to retrieve the highest timestamp and another trip to perform the actual write. The response time of *write thru* is $wan * 2 + overlay$ because the write has to send invalidations and wait for acknowledgments to come back from a write quorum in OQS in addition to retrieving the highest timestamps and sending the write to be performed. If we assume the workload consists of all consecutive reads followed by

consecutive writes (or all consecutive writes followed by consecutive reads), most reads are *read hit* (except for the first one) and most writes are *write suppress* (except for the first one). And we have the best case average response time for DQVL:

$$resp_{DQ-best} = w * wan * 2 + (1 - w) * lan \tag{4.1}$$

When the workload consists of interleaved reads and writes, most reads are *read miss* and most writes are *write thru*. The average response time in this case is the worst:

$$resp^{w<0.5}_{DQ-worst} = w * (wan + wan + overlay) + (1 - w) * (lan + overlay) \tag{4.2}$$

$$resp^{W\geq 0.5}_{DQ-worst} = w * ((2w - 1)/w * (wan + wan) + (1 - w)/w * (wan + wan + overlay)) + (1 - w) * (lan + overlay) \tag{4.3}$$

The average response time of other protocols are as follows:

$$resp_{ROWA} = w * wan + (1 - w) * lan \tag{4.4}$$

$$resp_{ROWAA} = lan \tag{4.5}$$

$$resp_{Majority}(orresp_{Grid}) = w * wan * 2 + (1 - w) * wan \tag{4.6}$$

Average response times of various protocols are illustrated in Figure 4.6 where we plot the average response times while varying the write ratio and fix the number of replicas to 15. DQVL provides its best case response time when workloads consist of only *read hits* and *write suppresses*. As indicated by the third curve from the bottom, DQVL *read hits* yield performance competitive with ROWA-A epidemic algorithms against read-dominated workloads because they only need to communicate with the closest server. DQVL has the worst case response time against workloads consisting of a large number of *read misses* and *write thrus*. DQVL *read misses* and *write thrus* require communication with distant servers similar to the behaviors of both majority and grid quorum operations. Therefore, they all experience the *wan* delays. Furthermore, because writes in quorum systems (including

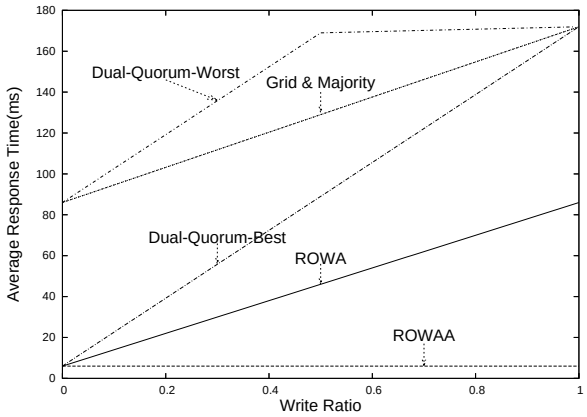
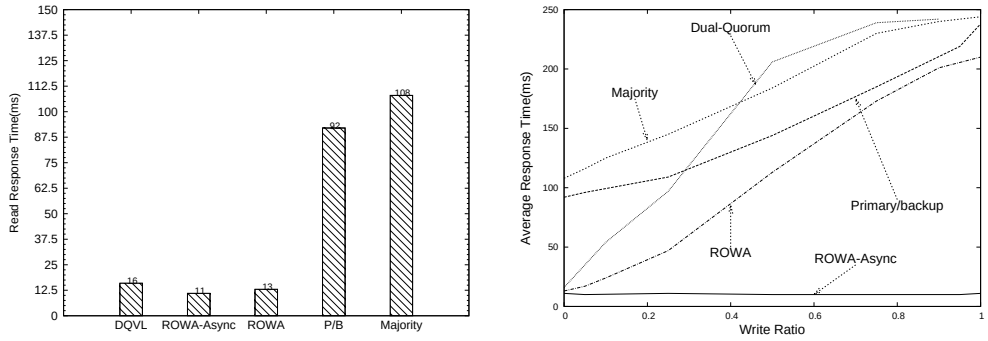


Figure 4.6: Average response time

DQVL) require one *wan* trip to retrieve the highest timestamp and another to perform the actual write, their response time is twice as much as that of ROWAA. *write thrus* require an additional *wan* trip to invalidate a write quorum in OQS. At 50% write ratio, when DQVL has the maximum amount of *write thrus*, the overall response time of DQVL reaches its worst case as indicated by the top most curve.

Experimental evaluation We have also developed replication prototypes for DQVL, primary/backup, majority quorum, ROWA-Async and ROWA protocols. All the prototypes are built in Java and run on eight Emulab nodes. In our prototype experiment, we set the "lan" delay between a client and its closest edge server to 8 ms. The "wan" delay between the client and other edge servers is 86 ms. And the network delay among edge servers is 80 ms.

In the rest of this section, we compare the response time of five protocols under our target workload, the TPC-W workload specified for the user profile. We show that DQVL yields better response time than protocols providing strong consistency guarantees and competitive response time to protocols with relaxed consistency guarantees.



(a) 5% write rate (b) varying write rate

Figure 4.7: Response time vs. write rate

Write ratio We first evaluate the response time by fixing the write rate to 5%, which is the update rate for TPC-W profile object, i.e. a workload with a low update rate and strong access locality. Accesses to the profile object consist of 95% reads on a customer's purchase history, credit information, and addresses and 5% writes on a customer's shipping address when processing an online purchase. When the profile is replicated on edge servers, a customer is routed to the closest edge server to access its profile information.

As illustrated in Figure 4.7 (a), DQVL provides at least six times read response time improvement over primary/backup and majority quorum protocols that are used to provide strong consistency guarantees. DQVL yields almost the same read response time as ROWA and ROWA-Async protocols because it allows most client reads to be processed only at the client's closest replicas with only 8 milliseconds RTT[4] while maintaining the same level of consistency guarantees as both primary/backup and majority quorum protocols by running the dual-quorum invalidation protocol between the closest replica and the rest of replicas in the system.

Figure 4.7 (b) is the sensitivity graph illustrating the overall response time changes as we vary the write rate. As writes dominate the workload, DQVL's response time approximates that of the majority quorum protocol and becomes higher than those of pri-

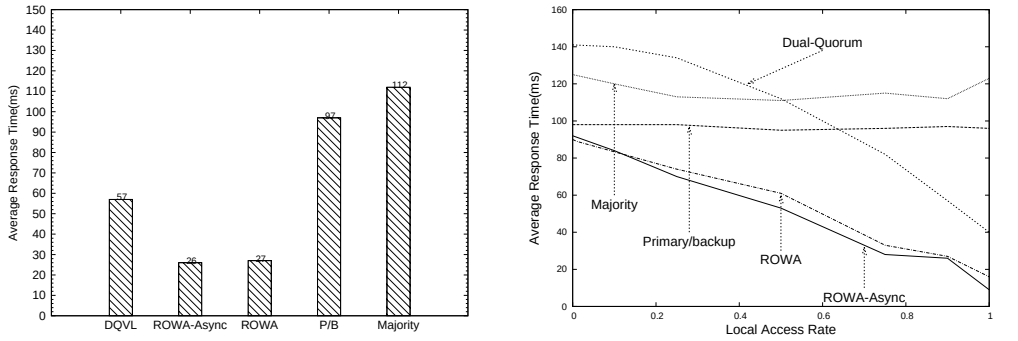
[4]Response times of all prototypes are higher than the underlying minimum network delays due to experimental variation and un-tuned code.

mary/backup and ROWA. The main reason is that DQVL clients, following the same procedure as the majority quorum protocol, need to obtain the latest timestamp from a read quorum before sending the write to a write quorum in IQS . Two round trips are required for both the majority quorum protocol and DQVL while only one round trip is needed for primary/backup and ROWA protocols. The additional trip to obtain the timestamp prior to performing the actual write increases the average response times of both DQVL and the majority quorum protocol compared with ROWA protocol.

Access locality In this subsection, we evaluate response time when some portion of client requests are routed to replicas other than the client's default closest one. Under normal circumstances, requests are routed to the client's closest server. But the unavailability of the closest replica or the geographical movement of the client can sometimes result in the requests being routed to distant replicas.

Figure 4.8 (a) illustrates protocols' response times at our target 5% write rate and 90% access locality (i.e. 10% of client requests are sent to distanced replicas and 90% of client requests are sent to the client's closest replica). The 90% access locality is a pessimistic measure for Internet edge servers given topical network failure rate is below 10% and the majority of end users do not travel frequently. DQVL outperforms both primary/backup and majority quorum protocols for the workload of our interest while preserving the same consistency level even in cases where client requests are directed to distanced replicas. Note that ROWA-Async protocol yields the optimal response time at the cost of serving reads with potentially inconsistent data when requests are directed to the distanced replicas.

In DQVL protocol, the response time of reads at distanced replicas is higher than the normal response time experienced when reading from the closest one. As the access locality varies, the overall response time changes accordingly. Figure 4.8 (b) indicates the relationship between the access locality and the overall response time of five protocols. DQVL suffers at the low access locality because both reads and writes needs to contact



(a) 5% write rate & 90% access locality (b) 5% write rate & varying access locality

Figure 4.8: Average response time vs. access locality

replicas in both input and output quorum systems. But DQVL's response time keeps improving as the access locality becomes higher. The majority quorum and primary/backup protocols are not affected by the access locality because neither protocol is designed to take advantage of the access locality in the edge service environment. This graph suggests that when the access locality is 70% or higher, DQVL should be preferred over primary/backup or majority quorum protocols for replication systems requires low response time and strong consistency guarantees.

4.4.2 Availability

In this section, we provide analytical models to evaluate the availability of the dual quorum protocol in comparison with other popular replication protocols.

We define the availability (av) as the number of client requests successfully processed by the system over the total number of requests submitted to the system during a given time period. A request is rejected by the system when target consistency semantics cannot be satisfied [136] or if insufficient nodes are available to process requests. In the context of this dissertation, systems are required to provide regular semantics [71]. For example, if more than half of the nodes are unavailable in IQS of a dual quorum system or in a majority quorum system, a client write will be rejected because the system can no longer

guarantee that a later read can always retrieve the value of this write. Because ROWA-Async protocol allows reads to return stale data from nodes without the latest update, it does not provide the regular semantics. The availability comparison among protocols is fair only if those protocols offer the same level of consistency. Therefore, to avoid ROWA-Async protocol returning arbitrary value, we design the experiment implementing ROWA-Async protocol to reject client reads returning stale data.

The availability of both *read hit* and *read miss* are $\min(av_{orq}, av_{irq})$. The availability of both *write thru* and *write suppress* are $\min(av_{irq}, av_{iwq})$. Given that the size of a quorum (referring to voting based quorum systems) is qs and the total replication size is N, the availability of the quorum is

$$av_{quroum} = \sum_{i=0}^{N-qs} \frac{N}{qs}(1-p)^{qs+i}p^{n-qs-i} \tag{4.7}$$

The availability of the dual-quorum system can be expressed as

$$av_{DQVL} = (1-w) * min(av_{orq}, av_{irq}) + w * min(av_{iwq}, av_{irq})$$

Noticed that the above model provides the worst-case availability analysis of the dual-quorum system because it under estimates the availabilities of both *read hits* and *write suppresses*. Because a *read hit* requires only a read quorum in OQS and a *write suppress* requires only a write quorum in IQS. Similarity, we derive the available models of other quorum systems as the following:

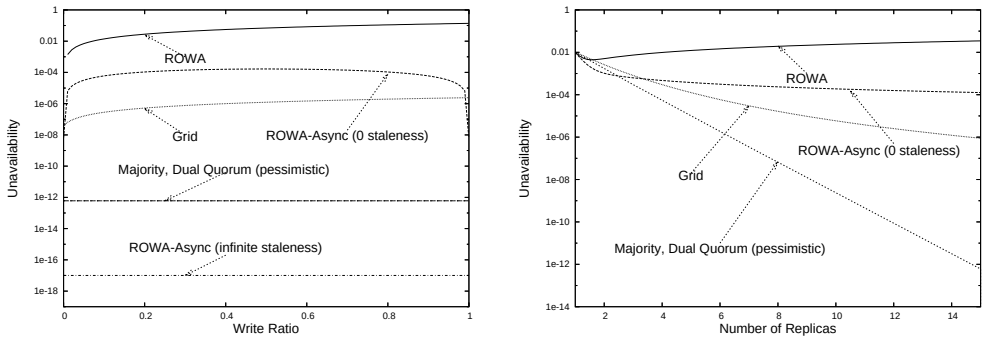
$$av_{ROWA} = (1-w) * (1-p^n) + w * (1-p)^n \tag{4.8}$$

$$av_{ROWAA} = 1 - p^n \tag{4.9}$$

$$av_{Majority} = \sum_{i=1}^{\frac{n-1}{2}+1} \frac{n}{\frac{n-1}{2}+i}(1-p)^{\frac{n-1}{2}+i} * p^{\frac{n-1}{2}+1-i} \tag{4.10}$$

$$av_{Grid} = (1-p^{\sqrt{n}})^{\sqrt{n}} - w * (1-(1-p)^{\sqrt{n}} - p^{\sqrt{n}})^{\sqrt{n}} \tag{4.11}$$

Figure 4.9 illustrates the unavailability of DQVL in comparison with other protocols in log scale. The unavailability is computed as $1-av$. An unavailability of 10^{-i} corresponds



(a) Unavailability vs. write ratio (b) Unavailability vs. number of replicas

Figure 4.9: System unavailability

to the availability of i 9's. Our simple model assumes the failure probability $p = 0.01$ and that failures (including server crashes and network failures) are independent. Read and write rates are defined as $1 - w$ and w.

Figure 4.9 (a) illustrates the systems' unavailability as we vary the write ratio and fix the number of replicas to 15 (in both *IQS* and *OQS*). The key result is that DQVL's availability tracks that of the majority quorum. Note that the DQVL's availability measurement is pessimistic because a read can proceed without contacting any read quorum in IQS if the read quorum in OQS holds valid volume and object leases; this effect may mask some failures that are shorter than the volume lease duration. Note that ROWA-Async protocol provides excellent availability by allowing reads to return arbitrarily stale data to clients. When our experiments allow no stale reads in ROWA-Async protocol, it yields poor availability that is several orders of magnitude worse compared to other quorum based protocols and our DQVL protocol.

Figure 4.9 (b) illustrates systems' unavailability as we vary the number of replicas and fix the write ratio at 25%. It shows that the unavailability of DQVL has similar behavior as the majority quorum system. The availability of quorum based protocols, including DQVL, improves as the total number of nodes increases. The availability of ROWA and ROWA-Async with no stale reads is insensitive to the number of nodes in the

system.

4.4.3 Communication Overhead

This section analyzes DQVL's communication overhead in terms of the number of message exchanges required in processing a client request. To simplify the model, the study assumes weights of all message types are equal. In addition to notation used in the previous analytical study, we introduce $|irq|$ that represents the size of a read quorum in IQS. When a OQS server sends renews an object or an volume lease from a read quorum in IQS, we use $|irq|$ to indicate the number of messages sent by the OQS server (one message to each server of the IQS read quorum). msg_r and msg_w denote numbers of message exchanges when processing a read and a write. Our model targets the average number of message exchanges which is calculated as $msg_r * (1 - w) + msg_w * w$.

A *read hit* requires $2*|orq|$ messages because a client sends to and receives from each server of an OQS read quorum one message. But for a *read miss*, each participating OQS server that needs to renew the volume lease or the object sends a renewal request, receives a renewal reply, and responds with an renewal acknowledgment to a read quorum in IQS, which requires $3*|irq|$ messages in addition to the $2*|orq|$ messages. when all servers of the OQS read quorum need to renew their local volume leases or the object, the total message cost is $2*|orq|+3*|orq|*|irq|$. A *write suppress* requires $2*(|irq+iwq|)$ messages because it retrieves the highest timestamp from an IQS read quorum and performs the write on an IQS write quorum. But a *write thru* requires additional $2*|iwq|*|owq|$ messages because of invalidations and acknowledgments between an IQS write quorum and an OQS write quorum. The total messages required for a *write thru* is $2*(|irq+iwq|+|iwq|*|owq|)$. Therefore, the average number of message exchanges for DQVL when workload consists of only consecutive reads followed by consecutive writes (or vice versa) is:

$$msg_{DQ-best} = w * 2 * (|iwq| + |irq|) + (1 - w) * 2 * |orq| \tag{4.12}$$

When the workload consists of only interleaving reads and writes, the average number of

messages required is:

$$msg_{DQ-worst}^{w<0.5} = w * 2 * (|irq| + |iwq| + |iwq| * |owq|) + (1-w) * (3 * |irq| * |orq| + 2 * |orq|) \quad (4.13)$$

and

$$msg_{DQ-worst}^{w\geq 0.5} = w * (\frac{2w-1}{w} * 2 * (|irq| + |iwq|) + \frac{1-w}{w} * 2 * (|irq| + |iwqq| + |iwq| * |owq|)) + (1-w) * (3 * |irq| * |orq| + 2 * |orq|) \quad (4.14)$$

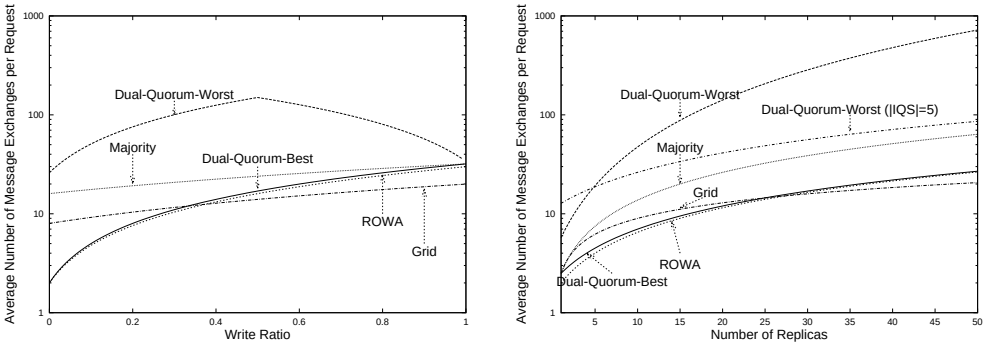
The average number of messages required in other protocols are as the following:

$$msg_{ROWA} = w * 2 * N + (1-w) * 2 * 1 \quad (4.15)$$

$$msg_{Majority}(or\ msg_{Grid}) = w * 2 * (|rq| + |wq|) + (1-w) * 2 * |rq| \quad (4.16)$$

To make the comparison fair, both IQS and OQS systems of DQVL are configured the same as in the previous study, i.e. read and write quorums of IQS include a majority of servers and the read quorum size of OQS is one.

Figure 4.10 shows the average number of messages required to process a client request in log scale. As illustrated in Figure 4.10 (a), in the worst case where the write ratio is at 50%, DQVL can have high communication overhead as reads and writes interleave with each others. In this case, most reads are *read misses* and most writes are *write throughs* which involve both IQS and OQS in processing requests. However, DQVL's overhead should be comparable to other approaches in practice. First, workloads that DQVL is designed to face are dominated by reads. Consecutive reads are likely to benefit from having objects cached on OQS servers, i.e. the target workloads have a large number of *read hits*. Second, the design of DQVL allows us to vary the OQS size to meet read performance goals while varying the IQS size to balance overhead vs. availability goals. As shown in Figure 4.10 (b), once we fix IQS at a moderate size while letting the OQS size



(a) varying write ratio (b) varying the number of replicas

Figure 4.10: Communication overhead

grow, the communication overhead yielded by DQVL is at the same level as the majority quorum without requiring many *read hits* in the workload.

Although the dual quorum protocol is described in terms of two quorum systems, *IQS* and *OQS* , an *IQS* server can physically be on the same node as an *OQS* server. Therefore, the overall communication overhead could be less because some messages become local.

4.5 Summary

This section presents dual-quorum replication, a novel data replication algorithm designed to support Internet edge services. Through both analytical and experimental evaluations, we demonstrate that this replication protocol offers nearly ideal trade-offs among high availability, good performance, and strong consistency under the target workloads.

Several important issues will be addressed in our future work. It will be interesting to configure both IQS and OQS to optimize other metrics. For example, we can configure the read quorum size in OQS to be larger than one to avoid timeouts on invalidations. We can also configure IQS as a grid quorum system [29] to reduce the overall system load. We are also interested in modifying DQVL to provide different consistency semantics (e.g.

atomic semantics [71]) and comparing the cost difference.

Chapter 5

Towards the Unified Replication Architecture

In this chapter, we advance the design methodology of the semantic-aware replication (SAR) by introducing the replication microkernel concept. We will first introduce a novel replication framework, PRACTI replication, that will be used as the "microkernel" in the envisioned new replication architecture. Then we outline the new replication architecture based on the microkernel approach used for designing operating systems [18, 34, 99, 53].

One approach to building new distributed applications is reusing our SAR objects. Although existing objects are designed based on TPC-W specific requirements, those objects capture the abstractions of many important and representative data replication scenarios commonly seen in a broad range of applications. When building new applications, some or all of the objects may be reused by matching the semantics exhibited by individual shared data sets. For example, the inventory object implements the algorithm for solving resource allocation problems. It can be used in other e-commerce, supply-chain, and ticket reservation systems to allocate identical resources on distributed nodes. The order object implements the abstraction of the single-reader/multi-writer scenario that can be used as an administrative tool in the distributed environment to collect statistical

information from multiple nodes. The catalog object implements the abstraction of the multi-reader/single-writer (dissemination) scenario. It can be used for the client-server oriented data replication such as AFS and IBM sports and event information systems.

Relying on the SAR objects, however, has three limitations. First, the scope of new applications that may be built with the existing objects is limited. When data sets within applications of other classes exhibit new semantics not captured by the SAR objects, we have to introduce additional objects to exploit those semantics. Second, it is rather expensive to introduce new objects based on SAR's initial design because SAR objects are built from scratch. The ad-hoc approach currently used to design SAR objects involves development overhead because low-level mechanisms used for exchanging and ordering updates among nodes are repeatly implemented for each object. Third, the current SAR design does not address the cross-object consistency that may be a critical issue in applications of other classes. Reasoning about the cross-object consistency is hard because SAR objects work independently. Leaving this issue open in the distributed TPC-W prototype is acceptable because no consistency requirements are violated. However, the same design approach may not work in building other applications that demand consistency guarantees across objects.

A joint effort with colleagues from the LASR lab investigates the first PRACTI (Partial Replication, Arbitrary Consistency, and Topology Independent) replication framework [36] that can potentially refine the design of SAR to eliminate those limitations. This chapter explores the integration of the semantic-aware methodology and the PRACTI framework. By offering a set of flexible primitives for building data replication systems in WANs, PRACTI simplifies the design of SAR objects by allowing a single framework to subsume a broad range of existing approaches. We can consider PRACTI mechanisms as a replication microkernel and implement SAR objects as domain-specific replication libraries using primitives supported by the microkernel. The replication libraries implement policies based on specific semantics that they exploit. Furthermore, because PRACTI mechanisms

are not entangled with specific policies, they can be used to facilitate the implementation of SAR objects with any point-solution policies embedded in existing mechanisms.

In this chapter, we briefly describe the high-level design and key insights of PRACTI mechanisms. (However, to fully understand PRACTI's detailed design and features, readers need refer to our technical report [36].) Then, we outline the envisioned new SAR architecture with the support from PRACTI primitives, followed by a discussion on how to implement TPC-W objects atop PRACTI primitives. Finally, we demonstrate the advantage of combining the SAR design methodology with the PRACTI mechanisms using two benchmark evaluations.

5.1 Design of PRACTI Replication

Our research on PRACTI replication was motivated by the fact that mechanisms and policies are entangled in existing replication systems. As a result, when building a replication system for a new environment, we must either develop it from scratch or modify existing mechanisms to accommodate new policy trade-offs. PRACTI seeks to define core primitives on which different replication policies can be implemented. In this section, we will briefly describe the overall design of PRACTI and primitives it provides.

5.1.1 Design overview

PRACTI is based on a log-exchange protocol similar to that of Bayou [95] to enable synchronization between two nodes. Basically, nodes exchange portions of their logs to propagate writes through the system. Bayou log-exchange protocol allows an arbitrary node pair to exchange updates and offers the basis for providing a range of consistency guarantees [136]. But a fundamental limit in Bayou log-exchange approach is that the protocol requires full replication in order to ensure the causally-consistent prefix of the system's writes on every node.

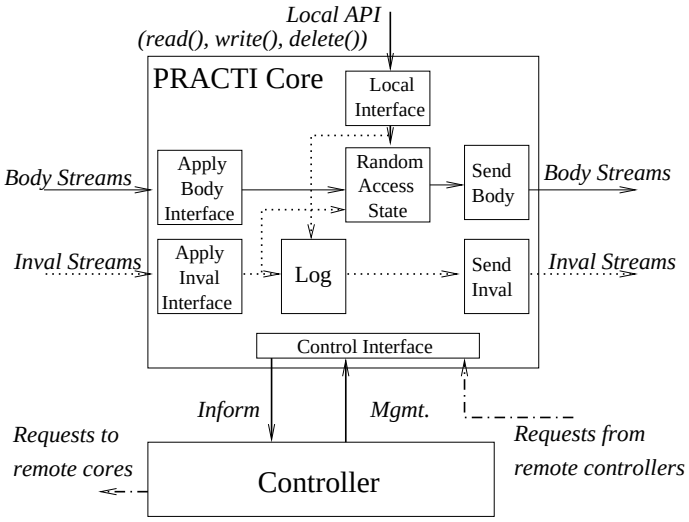


Figure 5.1: High level PRACTI architecture for one node.

To allow any node to replicate a subset of the data, PRACTI mechanisms separate the associated meta data, namely *invalidations*, from the actual updates. A node may replicate a subset of the data and specifies to receive only the invalidations and update bodies of the subset from other nodes. Furthermore, the invalidations that are not associated with the updates to the subset replicated on a node are explicitly encoded via *imprecise invalidations* when received by the node.

Figure 5.1 illustrates the high-level design of the PRACTI framework. Each node has two main components: the replication *core* and the *controller*. The *core* contains the basic PRACTI mechanisms to process incoming messages and maintain a local view of the system's state. The controller embodies a system's *policies* by initiating communication among nodes, similar to the semantic-aware replication policies we have explored in previous chapters. In other words, different controllers can be implemented with specific replication topologies and consistency guarantees. In the rest of this section, we will describe the replication core in detail and leave the discussion of the controller to the next section where we talk about combining the SAR design approach with PRACTI in Section 5.2.

5.1.2 Separate invalidations from bodies

The separation of invalidations from bodies allows partial replication of data. While invalidations are sent to all replicas eventually, bodies are sent to replicas based on specific replication policies. To accomplish this separation, the PRACTI architecture splits *write messages* into two parts: (1) *invalidation messages* identify what objects were written and when the writes occurred and (2) *body messages* contain the bodies of the writes.

PRACTI distributes invalidations using a straightforward variation of Bayou's log exchange protocol that operates on invalidations rather than complete writes. When a node receives a new invalidation message, it applies the message to its log, and when it sends its log to another node, this log contains invalidations rather than complete writes. When a node receives an invalidation, then if the invalidation's logical timestamp exceeds the logical time for the object in the data store, the node updates the object's state in data store by marking it *INVALID* and updating its logical timestamp. PRACTI thus retains three key invariants on invalidations [95]. First the *prefix property* requires that a node's state always reflects a prefix of the sequence of invalidations by each node in the system. I.e., If a node's state reflects the ith invalidation by some node η in the system, then the node's state reflects all earlier invalidations by η. Second, each node's local state always reflects *causally consistent* [58] view of all invalidations that have occurred. This property follows from the prefix property and from the use of Lamport clocks [70] to ensure that once a node has observed the invalidation for write w, all of its subsequent writes' logical timestamps will exceed w's. Third, the system ensures *eventual consistency*: all connected nodes eventually agree on the same total order of all invalidations. Given this basis, we can enforce a broad range of consistency semantics [136].

Although invalidations must be sent and applied in causal, logical timestamp order, PRACTI nodes can distribute bodies according to arbitrary policies, in arbitrary order, across arbitrary topologies. A PRACTI node must therefore synchronize arriving bodies with the invalidation streams before applying bodies to its local state. PRACTI maintains

the invariant that update bodies are not applied to the data store until after the corresponding invalidation message. To ensure this invariant, the core's *applyBody* interface buffers updates until they may be safely applied. When a node finally can apply a body message, if the logical time for the object in the data store has not advanced past the body's logical time, the data store marks the object *VALID* and stores the body.

The separation of bodies from invalidations affects local read requests. The system blocks a local read request until the requested object's status is *VALID*. Of course, to ensure liveness, when an *INVALID* object is read, an implementation should arrange for someone to send the body. A controller can implement any policy for doing this from a static hierarchy (i.e., ask your parent [19] or a central server [57] for the missing data) to a separate, centralized directory, to a DHT-based directory [115], to a hint-based search strategy [103], to a push-all strategy [95] (i.e., "just wait and the data will come.") In addition to distributing bodies in response to demand reads, controllers can also prefetch bodies [50], pre-push bodies [51, 101, 124], or pre-position bodies according to a global placement policy [123].

5.1.3 Imprecise invalidations and interest sets

Where separation of invalidations from bodies supports partial replication of data, imprecise invalidations allow partial replication of meta data: nodes receive precise invalidations for objects they plan to access but receive only summaries of invalidations for other objects. Although each invalidation is small, imprecise invalidations are crucial for large systems. As demonstrated by Mike et al. [36], imprecise invalidations can reduce replication costs by an order of magnitude compared to requiring every node to see every invalidation of every object.

An *imprecise invalidation* is a conservative summary of a group of ordinary invalidations, which we refer to as *precise invalidations.* We use the term *general invalidation* to refer to either a precise or imprecise invalidation. An imprecise invalidation includes a set

of *targets* and a range of logical times defined by some *start* and *end* times, and it denotes that "One or more objects in *targets* was updated between *start* and *end*." An imprecise invalidation must *cover* all summarized invalidations—any invalidation summarized by an imprecise invalidation i must have its target ID included in $i.targets$ and its logical time included between $i.start$ and $i.end$. Notice that a general invalidation's start and end times are partial version vectors with as few as one element (to cover invalidations by one node) and as many as n elements (to cover invalidations by all nodes in the system). Also note that an imprecise invalidation i can be conservative: $i.targets$ can include objects that were not invalidated between $i.start$ and $i.end$. This rule supports concise encodings of large numbers of files (e.g., a list of subdirectories or a Bloom filter of object IDs).

When a node α receives an imprecise invalidation i, α applies i to both its log and its data store. For the log, i serves as a "placeholder" so that if α sends its log to another node, i indicates which precise invalidations are omitted. Logs thus still maintain the prefix property, causal consistency, and eventual consistency invariants. The benefit of imprecision is efficiency: when a controller tells node α to send invalidations to node β, the request indicates subsets of the object ID space for which α should summarize invalidations using imprecise invalidations before sending them.

Imprecise invalidation i must also update the data store. A naive strategy would mark every object covered by $i.target$ as *INVALID* to logical time $i.end$. Such an approach has two problems. The first is performance: the cost to process an imprecise invalidation would be proportional to its *target* set size. The second problem is liveness: each invalidated object o would remain *INVALID* until the node receives a body for o with a logical time at least $i.end$. Note that typically, only one object in $i.targets$ actually was written as late as $i.end$ by the summarized invalidations; for any other object p in $i.target$, there may exist no write in the system that can make p *VALID*.

To address these performance and liveness problems, nodes use a more sophisticated approach: nodes allow portions of the data store to include stale state after an imprecise

invalidation, but they ensure consistency by preventing observation of stale data store entries. To do this, a node partitions its data store into one or more *interest sets*. An *interest set* is a portion of the object ID space that is either *PRECISE* or *IMPRECISE*. An interest set IS is *PRECISE* if and only if the data store reflects all precise invalidations for all objects in IS up to the node's current logical time. For consistency, a local read of an object must block until the enclosing interest set is *PRECISE*; when a read blocks, the controller must initiate sufficient communication in order to make the interest set *PRECISE* and to allow the read to complete. We differ the detailed algorithm to the technical report [36] since the focus of this chapter is on combining the PRACTI mechanisms with SAR design methodology.

5.2 The Replication Microkernel Architecture

In this section, we outline the new replication architecture by leveraging PRACTI mechanisms. Note that we currently do not have a fully implemented SAR replication system that utilizes PRACTI mechanisms. This section outlines our vision to combine the SAR approach, a methodology to design/tune replication policies based on the semantics of the shared data, with the PRACTI mechanisms, a single framework to subsume a broad range of existing replication techniques. The combined approach is similar to the *microkernel* concept used to design operating systems [18, 34, 99, 53]. In the following discussion, we call our envisioned replication architecture the *replication microkernel architecture*.

In the replication microkernel architecture, we aim to use the PRACTI controller to accommodate various SAR policies. We first discuss the PRACTI controller that provides us the interfaces to the PRACTI core. Second, we describe how to implement various SAR policies by utilizing the PRACTI controller. Then, we present the high-level design of the replication microkernel architecture.

5.2.1 PRACTI controller

The PRACTI core's mechanisms enforce their safety properties regardless of what incoming messages they see. Our cores use an asynchronous style of communication in which incoming messages or streams are self-describing—the rules for processing each incoming message are completely defined, and interpreting a message does not require knowledge of what request triggered its transmission. Any machine can therefore send any legal protocol message to any other machine at any time. Each core provides an interface for requesting that the node send invalidations or bodies to other nodes, but these requests can be regarded as hints: the loss of messages or the introduction of extra messages can affect system performance but not the correctness of responses to application read and write requests.

The controller implements policies that focus on liveness (including performance and availability.) The controller's basic job is to ensure that the right cores send useful data at the right times in order to do such things as satisfy a read miss, prefetch data to improve performance, or provision a node's local storage for disconnected operation. Controllers accomplish this by sending requests to trigger communication between cores.

The controller is defined by its interface. Within this interface, different implementations provide different policies. Controllers use three sets of interfaces to accomplish their work: a core calls a controller's *inform* interface to inform the controller of important local events like message arrival or read miss, a controller calls a remote core's *remote request* interface to trigger sends of invalidation streams or bodies, and a controller calls its core's *management* interface for maintenance functions like garbage collection and interest set split/join. Additionally, a set of controllers implementing a specific distributed policy may communicate with one another using policy-specific interfaces.

5.2.2 Disentangle mechanism from policy

Disentangling mechanism from policy is the key to solving the extensibility problem in the current SAR design. As we mentioned in previous discussions, the implementation of SAR objects includes both replication policies and mechanisms. This design limits the extensibility of SAR because similar sets of mechanisms need to be implemented in every object.

The PRACTI techniques cleanly separate mechanism from policy in order to support a broader range of replication, topology, and consistency policies than made available by current techniques. This implementation provides low level mechanisms over which higher-level services that control policy choices can be built. As we adapt PRACTI mechanisms to build SAR objects, the implementation of objects will only consist of replication policies as the higher-level services supported by low level PRACTI mechanisms. Furthermore, the flexibility of PRACTI mechanisms allows for building replication policies that are embedded in existing replication mechanisms.

5.2.3 Replication microkernel architecture

A straightforward way to combine SAR and PRACTI is to implement SAR objects within the PRACTI controller. As we have described earlier in this section, PRACTI controllers are designed to manage when and where an update or an invalidation should be sent to or requested from by using a set of interfaces to the low level mechanisms. The implementations of PRACTI controllers focus on policies, matching the design goal of separating mechanism from policy for extensible SAR objects.

The combined approach is similar to the microkernel approach used for designing operating systems. The PRACTI core provides replication primitives for implementing policies that allow partial-replication, arbitrary-consistency, and topology-independence. I.e. policies can specify replication of any subsets of data on any nodes, tunable consistency guarantees [136], and arbitrary paths for the propagation of updates among nodes. Because

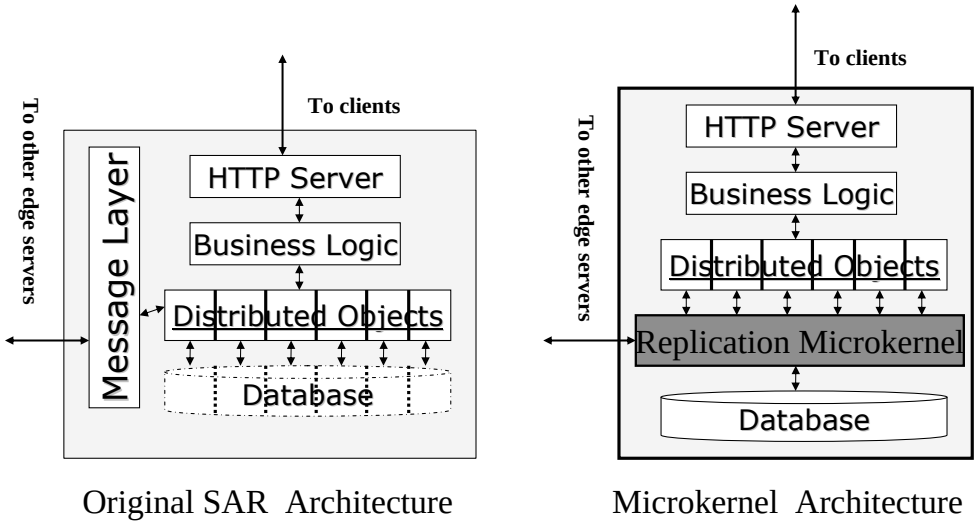


Figure 5.2: Architecture comparison for the edge server

the PRACTI exports the interface to low level mechanisms, we can subclass a semantic-aware controller that runs a set of SAR objects, each managing the replication of some interest sets by using specifically tuned policies. The function of SAR objects is similar to that of the domain-specific libraries built for microkernel operating systems.

Figure 5.2 illustrates the architecture of the new edge server that employs the replication microkernel for WAN data replication in comparison with the architecture of the edge server described in Chapter 3. The architecture of the new edge server breaks the original SAR layer into two layers: *semantic-aware policy layer* and *replication microkernel*. The semantic-aware policy layer include the semantic-aware controller and a set of distributed objects running within the controller. The replication microkernel consists of the PRACTI primitives. Because the PRACTI primitives persistently store the content of the out-bound messages in the data store, the new architecture does not require a persistent messaging layer to survive crashes. But the policies need to include the logic to request for re-connection and re-transmission of any lost messages.

5.3 The PRACTI Implementation of TPC-W Objects

Leveraging PRACTI primitives, the implementations of some TPC-W objects are straightforward while that of others requires additional design efforts. This section focuses on how to utilize and extend PRACTI primitives to implement the TPC-W specific objects discussed in Chapter 3. As described in the previous section, the TPC-W objects need to be implemented by using the PRACTI controller interface.

5.3.1 Topology independence

Two TPC-W objects, *catalog* and *order*, exploit the update topology. Those two objects are straightforward to implement using PRACTI primitives because PRACTI allows any node to exchange updates with any other node in the system.

Catalog object

The *catalog* object implements the abstraction of one-to-many update, i.e. update at one location and read at multiple locations. The catalog object running inside the PRACTI controller manages the propagation of local updates. At a high level, all the nodes are configured to be either the writer-node or a reader-node in the one-to-many scenario. All reader-nodes subscribe for both invalidations and bodies from the writer-node during their bootstraps. And updates will be sent from the writer-node to all reader-nodes in a FIFO order.

To implement this update topology, the catalog object relies on the following PRACTI primitives. During a node's bootstrap, a catalog object is instantiated in the PRACTI controller. Based on the configuration information, the instance of the catalog object recognizes itself as either the writer or a reader. All the writes at the writer-node are bounded, i.e. the PRACTI core does not separate invalidations from the actual writes and propagates writes to other nodes through invalidation streams as bounded invalidations. A reader-node subscribes to the writer-node for the invalidations by using the

subscribeInval interface exported by the controller. Because Bayou's log-exchange protocol offers a casually-consistent pre-fix of the updates, all reader-nodes see invalidations (bounded) in the same order as their associated writes are performed at the writer-node. When a reader-node receives a bounded invalidation, it updates both its log and data store with the content in the bounded invalidation.

Order object

The *order* object implements the abstraction of many-to-one update, i.e. update at multiple locations and read at one location. At a high level, all the nodes are configured to be either a writer-node or the reader-node in the one-to-many scenario. The reader-node subscribes for both invalidations and bodies from all writer-nodes during its bootstrap. Each writer-node sends its local update to the read in a FIFO order.

The PRACTI implementation of the *order* object is similar to that of the *catalog* object. All writer-nodes process local writes as bounded. The instance of the order object on the reader-node initially subscribes to all writer-nodes for the bounded invalidations to the order data. The PRACTI core ensures invalidates from the same writer-node arrive at the reader-node in FIFO order. The reader-node retrieves the update content from the bounded invalidations received.

Discussion

As described above, the *catalog* object is easy to implement using PRACTI. In addition, optimizations of this object, such as self-tuning updates and scalable update topologies, are easier to implement on PRACTI. Although we can modify existing TPC-W implementations to include those optimizations, the new implementations are nearly as complex as the PRACTI core. And some of the mechanisms need to be re-implemented for optimizations of other objects. On the other hand, few design changes are required to implement those optimizations for the catalog object using PRACTI primitives. The actual implementa-

tions using PRACTI are as simple as configuring update subscriptions at bootstraps and issuing demand reads to the appropriate nodes using interfaces to the PRACTI core.

Self-tuning updates A reader-node subscribes for both invalidations and bodies of the *catalog data* from the writer-node by using the *subscribeInval* and *subscribeBody* interfaces exported by the controller. When a local update is performed, the PRACTI core at the writer-node separates the update into an invalidation and a body. The invalidation is sent in the foreground to all reader-nodes using Bayou's log-exchange protocol and bodies are sent in the background using TCP-nice [122]. Because bodies are sent in the background, they may arrive at a reader-node much later after the corresponding invalidation arrives. When the PRACTI core on a reader-node observes that the requested data is invalidated while processing a client read, it notifies the controller about the invalid status. Then the controller passes the notification to the catalog object. The catalog object invokes *demandRead* function in the PRACTI core to retrieve the body from the writer-node. However, *demandRead* may time out due to network partitions. The current PRACTI implementation lacks the ability to notify readers when the local copy cannot be validated. To implement this optimization, we need to extend PRACTI to measure the staleness and propagate the staleness information to clients when necessary.

Scalable update topologies Our existing catalog object assumes a star topology for propagations of both invalidations and bodies. The star topology is not scalable, for example, when the bandwidth at the writer-node is limited. An hierarchical dissemination or aggregation topology can solve this problem by offloading the amount of out-bound traffic at the writer-node. The catalog object can implement a hierarchical dissemination by categorizing nodes in different levels and letting a node subscribe to another node at one level lower. The writer-node is the only node at level zero and all other nodes are at level one or higher. Another alternative is to adjust PRACTI to delay applying an invalidation until the body arrives [87]. This approach yields less staleness and bandwidth

consumptions compared with the design that does not separate invalidations from updates. The next section shows the advantage of using the optimized propagation topologies.

5.3.2 Numeric updates

The implementations of both the *inventory* and *best-seller list* objects are based on the fact that the shared data are of the numeric type. The updates either increase or decrease data's numeric value. Our original design minimizes the communication cost since inventory updates are commutative and can be aggregated. In addition, the design allows for numerical error [136] on local copies, i.e. buffering a certain number of updates (e.g. decrements of the numeric value) on each local node with the total decrements combined on all nodes never exceeding the numeric value.

Inventory object

Recall that the implementation of the *inventory* object divides the total inventory among all nodes by giving each object instance a *localCount* and enforcing the invariant that the sum of all local counts never exceeds the total global inventory count. In the prototype development in Chapter 3, we implement a simple protocol between the backend node and edge nodes for inventory re-distribution. Whenever the inventory difference between a pair of edge nodes exceeds a certain threshold, the backend node requests the inventory re-distribution between the two edge nodes. The backend node gains inventory information on edge nodes by monitoring the purchase orders submitted from those nodes.

To implement the inventory object on PRACTI, we need to introduce a new variable on each edge node, *localSalesCount*. A *localSalesCount* represents the number of a specific items sold at a given edge node. Each edge node still keeps a copy of *localCount* while the backend node keeps copies of both *localCount* and *localSalesCount* on all edge nodes. The backend nodes subscribes to all edge nodes for updates on *localSalesCount* and edge nodes subscribe to the backend node for updates on *localCount*. An inventory re-distribution is

triggered when the difference between *localCount* and *localSalesCount* falls below a certain threshold on a node. All updates are propagated as bounded writes through invalidation streams between the backend node and all edge nodes. During an inventory re-distribution between node a and b with a's *localCount* below the threshold, the backend first updates its local copy of b's *localCount* to a lower value. After ensuring that the update is propagated and applied at b (by issuing a *sync* to b and waiting for it to return from b), the backend node issues a *demandRead* to retrieve b's *localSalesCount* to ensure that the difference between b's *localCount* and *localSalesCount* does not fall below the threshold. If the difference is above the threshold, the backend node update its local copy of a's *localCount* to a higher value and propagate the update to node a. If the difference falls below the threshold, the backend node change back b's *localCount* to its original value and propagate the update to b. The backend node will then choose to take away the local inventory from another node.

Node that the implementation of the best-seller list on PRACTI is similar to that described above.

Discussion

The implementation of the inventory object described above lacks the ability to support commutative updates that are originally designed for the TPC-W inventory object. The original design of the inventory object propagates the update operations, e.g. increment or decrement the local inventory by a certain amount, rather than the updated values. Because additions and subtractions are commutative, they do not need to be propagated to remote nodes in the same order as they are applied on the local node. Based this observation, the inventory object is initially designed to support the propagation of commutative updates, which imposes less constraints compared with Bayou's gossip protocol.

However, PRACTI only has the mechanisms to propagate the updated values because this approach is simpler to implement and works well in most systems. Typically, when a node propagates a local operation on the shared data to other nodes, it has to

include the timestamp of the previous value that the operation modifies. When a remote node receives the update, it uses the timestamp of the previous value in the update to detect conflicting updates and roll back the state of the system as necessary to apply the update. The rollback is necessary because for a operation to yield the same result on a remote node, the remote node has to reflect the same state as when the operation is applied on the original node. PRACTI simplifies the design by eliminating the need for roll-backs. A PRACTI update contains the value as the result of the update operation instead of the actual update operation. Therefore, no roll-backs are necessary when a node observes conflicting updates. Once the conflict resolution rule determines the winning update, the value contained in the winning update is applied to the system.

We can extend PRACTI to support the propagation of operations without the need for system roll-backs. Although the implementations of both *inventory* and *best-seller list* are required to propagate operations, those operations can be applied at remote nodes without having to first read the previous value. The observation implies that the system does not need to roll back to a previous state to apply a certain operation. Operations can be applied incrementally yet commutatively. But a special feature is needed to ensure that the global constraint is not violated (i.e. the sum of all local counts never exceeds the total global inventory count).

Propagating commutative updates To propagate operations on the inventory data, the PRACTI core needs to distinguish inventory data from other shared data. Only the operations are propagated for the inventory (and the best-seller list) updates while only modified values are propagated for updates of other shared data. In other words, the system cannot be allowed to propagate both the operations and modified values of the same shared data. Furthermore, because inventory operations are commutative, we can propagate them in the body stream or a new stream that does not require a casually consistent pre-fix property.

Following the design of the current inventory object, the global inventory is divided

among all nodes. A backend node contains copies of inventory count on all nodes. During the bootstrap, the backend node subscribes for the body streams of all nodes to collect inventory updates from each of them. Each node also subscribes for the body stream of the backend node to receive inventory re-distribution. The inventory re-distribution is done by the backend node that sends a decrement operation to the node with a higher inventory count, and then send a increment operation to the node with a low inventory count. Because all nodes asynchronously send inventory updates to the backend node, the backend node has only approximate views of inventory counts on all nodes. Therefore, during the inventory re-distribution from node a to b, node a may receive a decrement operation from the backend but does not have sufficient local inventory to accommodate the operation. In this case, a's local inventory count becomes negative after processing the decrement operation. The backend node has to issue an increment operation to a to maintain a non-negative count on a. The attempt to take away some inventory from a is considered a failure and the backend will attempt to take away inventory from other nodes to give to b. The backend node will issue an increment operation to b only after it successfully takes away some inventory from a or some other node. To prevent a node's local inventory being negative after processing the decrement operation by the backend, the backend has to query the node's local inventory by issuing a demand read with the same timestamp as the decrement operation.

5.3.3 TPC-W profile object

TPC-W *profile* object described in Chapter 3 is implemented based on Bayou's log-exchange protocol. It uses the protocol to asynchronously propagate profile updates among nodes and ensures the eventual consistency of the profile data. When implemented with PRACTI primitives, local writes are propagated as bounded updates to other nodes, similar to the basic implementations of both the catalog and order objects. But because the profile object handles the multi-writer/multi-reader scenario, any node can read and modify the shared

data. The additional tasks are to detect and resolve conflicting updates on the profile data. PRACTI provides the conflict detection mechanism by including the timestamp of the previous update in the invalidation (in both bounded and unbounded). Two updates conflict when their previous timestamp are the same. To resolve the conflict, PRACTI compares the timestamps (Lamport clocks) of conflicting updates and uses the latest-win strategy to decide the final value.

Discussion

We can also implement self-tuning updates similar to the catalog object by separating invalidations from bodies of the profile updates. But we still need to extend PRACTI to bound the staleness of the data returned to clients as required in the optimization of the catalog object. Furthermore, the conflict detection and resolution have always been challenging tasks in replication systems that use the gossip protocol for asynchronous update propagations. The latest-win strategy implemented in the PRACTI core may not provide sufficient semantics for some applications. For example, the per-field resolution, which is originally designed for the profile object, cannot be implemented using the PRACTI mechanisms. We might be able to extend the mechanism for resolving the byte-range conflicts in file systems to work for the per-field resolution in database systems.

A replication system is much easier to implement when no conflict detection and resolution is necessary. Implementing the dual-quorum with volume leases (DQVL) protocol on PRACTI to replicate the TPC-W profile data is more attractive. DQVL offers the optimized trade-offs among consistency, availability, and response time while avoiding conflicting writes and bounding the staleness of the replicated data as described in Chapter 4.

However, we need to extend PRACTI to implement DQVL. Several high-level components need to be introduced in PRACTI as the building blocks for DQVL: (1) quorum-based operations (read and write), (2) callbacks, and (3) temporal error. A complete DQVL

design on PRACTI is beyond the scope of this dissertation. In the following paragraphs, we discuss how to use extend PRACTI to build the three building blocks.

Quorum-based operations The PRACTI core imposes no policies on where a local update should be propagated and where a demand read should query data from. To process a local update, the PRACTI core generates an invalidation and a body. The invalidation is applied to the log and the body is applied to the data store. Both the invalidation and the body are propagated to remote nodes subscribing for the interest set containing the datum. But no policies specify what remote nodes should subscribe for the update. When the PRACTI core processes a local read, the datum is returned immediately if it is valid. When the core discovers that the target datum is invalid, it blocks the read until the datum is validated. The core notifies the controller when it discovers that the target datum is invalid and depends on the controller to retrieve the body from remote nodes before unblocking the read.

We can implement policies in the controller to enforce quorum operations. Basically, the controller blocks the return of local operations to clients before receiving confirmations from a quorum of nodes. To extend a PRACTI local read to a quorum read, the controller issues demand reads to all nodes (including the local node) upon receiving a local read. After receiving replies from a read quorum of nodes, the controller selects a value with the highest timestamp as the result of the local read. To extend a PRACTI local write to a quorum write, the controller performs the task similar to a quorum read to obtain the highest timestamp from a read quorum of remote nodes. Then a higher timestamp is generated and sent to all nodes along with the write. To confirm that at least a write quorum of nodes has received the body (because a write is propagated as an invalidation and a body with the body propagated in the background), the controller issues *syncBody* calls to all nodes with the timestamp of the write. A *syncBody* call does not return until the target node successfully applies the body with a timestamp no less then the timestamp specified. After *syncBody* calls return from a write quorum of nodes, the controller allows

the local write to return, with the timestamp of the write.

Note that appropriate quorum configurations need to be specified for the controller to correctly perform the corresponding quorum-based operations.

Temporal error In the current PRACTI system, a node cannot bound the staleness of the data returned to clients. There is no information associated with each datum to indicate the time the datum is updated at remote nodes. When processing a local read, the PRACTI core cannot tell the freshness of the local data.

To bound the staleness of data, we need to introduce a vector clock on every node. Each entry of the vector clock represents the wall-clock time when the last invalidation is sent by the corresponding remote node. When a node sends a stream of invalidations to another node as the node is subscribed for invalidations, the wall-clock time of the sender is attached. When the invalidations arrive, the receiver updates the corresponding vector clock entry to the time associated with the last invalidation in the stream. To process a local read, a node checks for the the minimum time in the vector clock. If the difference between the local wall-clock time and the minimum time is greater than the maximum staleness allowed, the node has to subscribe to the remote node corresponding to the entry with the minimum time for both invalidations and bodies. The entry is updated to the new time attached to the last invalidation in the stream. A datum can be returned only when the difference between the current time and the minimum time in the vector clock is no greater than the maximum staleness bound.

Callbacks Existing PRACTI supports inefficient callbacks because a node sends all invalidations to remote nodes regardless the status of the data on the remote nodes. For example, when node b subscribes to node a for invalidations on an interest set is, all invalidations on is are sent to b and invalidations on other interest sets are sent to b as imprecise invalidations. In other words, no invalidations on any data are suppressed by a (invalidations are sent as either precise or imprecise invalidations to b). This approach may

be inefficient because a might send invalidations of a datum that is already invalidated.

But because DQVL is design for workloads with a low write ratio, using the existing PRACTI mechanisms to implement callbacks in DQVL is an acceptable solution. The fact that PRACTI never suppresses any invalidations eliminates the *write suppress* scenario in DQVL. Recall that a *write suppress* is resulted when two consecutive writes on the same data are processed. In this case, the invalidation resulted in the second write will not be sent to remote nodes caching the data because the first write has already invalidated the data on remote nodes. In the workloads targeted by DQVL where writes are infrequent, we do not anticipate too many consecutive writes. Therefore, the overall performance of DQVL will not be significantly degraded when callbacks are implemented using existing PRACTI primitives. And we will explore a more efficient callback implementation using PRACTI primitives in future work.

5.4 Case Study

In this section, we show that PRACTI mechanisms provide a flexible platform to incorporate the SAR design methodology by implementing on PRACIT two benchmarks, *mobile storage* and *WAN-FS*, and comparing their performance with the same benchmarks implemented atop existing replication frameworks. Note that we have not fully re-implemented our TPC-W prototype using the replication microkernel approach. However, we seek to demonstrate the feasibility of PRACTI mechanisms when used in the replication microkernel architecture. In particular, we compare PRACTI with existing replication techniques when used to implement the two benchmarks, which demand partial data replication on distributed nodes in WANs and vary the paths through which updates are propagated among those nodes.

Because PRACTI mechanisms support a wide range of policies, we are able to apply the SAR methodology when implementing two benchmarks, i.e. tune replication policies based on update topologies in building various aspects of two benchmarks. We

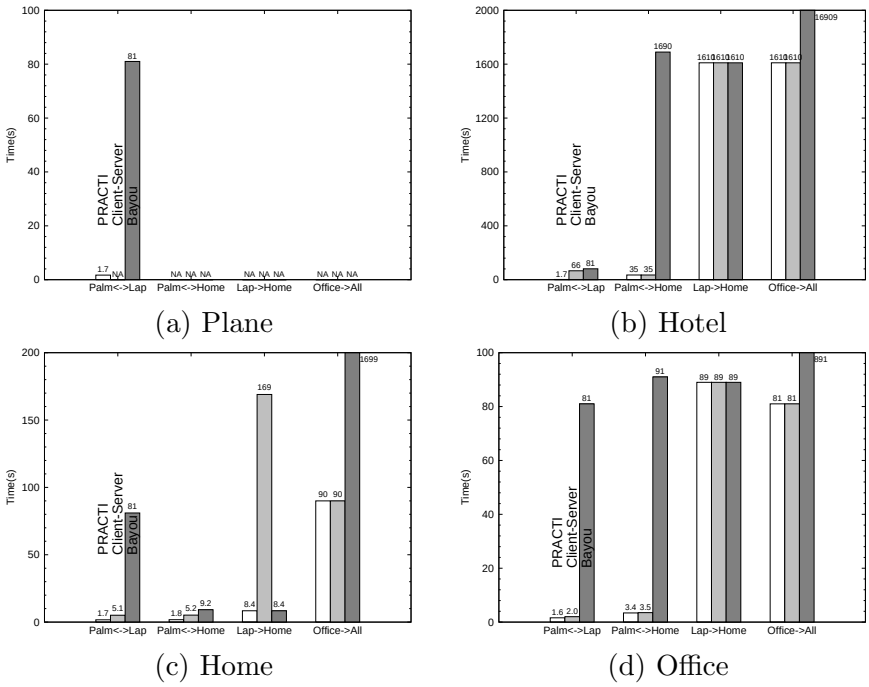


Figure 5.3: Synchronization time among devices for different network topologies and protocols.

anticipate that the PRACTI implementations will outperform the implementations using other replication techniques. The SAR design methodology focuses on constructing replication policies based on the semantics of shared date sets. For these two instances, the semantics that we leverage are the different update topologies. As described in detail in following paragraphs, while replication policies are fixed in other techniques, we are able to implement specific policies to optimize for different update topologies.

5.4.1 Mobile storage

Figure 5.3 evaluates PRACTI in the context of a mobile storage system that distributes data across palmtop, laptop, home desktop, and office server machines. We compare PRACTI to a client-server Coda-like system (that supports partial replication but that

	Storage	Dirty Data	Wireless	Internet
Office server	1000GB	100MB	10Mb/s	100Mb/s
Home desktop	10GB	10MB	10Mb/s	1Mb/s
Laptop	10GB	10MB	10Mb/s 1Mb/s	50Kb/s Hotel only
Palmtop	100MB	100KB	1Mb/s	N/A

Figure 5.4: Configuration for "mobile storage" experiments.

distributes updates via a central server) [64] and to a full-replication Bayou-like system (that can distribute updates directly between interested nodes but that requires full replication) [95]. All three systems are realized by implementing different controller policies over PRACTI.

As summarized in Figure 5.4 our synthetic workload models a department file system that supports mobility: an office server stores data for 100 users, a user's home machine and laptop each store 1% of that data, and a user's palmtop stores 1% of a user's data. Note that due to resource limitations, we store only the "dirty data" on our test machines, and we use desktop-class machines for all nodes; we control the network bandwidth of each scenario using a library that throttles transmission.

Figure 5.3 charts the time to synchronize dirty data among machines in four scenarios: (a) *Plane*—the user is on a plane with no Internet connection, (b) *Hotel*—the user's laptop has a 50Kb/s modem connection to the Internet, (c) *Home*—the user's home machine has a 1Mb/s connection to the Internet, and (d) *Office*—the user's office has a 100Mb/s connection to the Internet. The user carries her laptop and palmtop to each of these locations and co-located machines communicate via wireless network at speeds indicated in Figure 5.4. For each location, we measure time for machines to exchange updates: (1) P↔L: the palmtop and laptop exchange updates, (2) P↔L: the palmtop and home machine exchange updates, (3) L→H: the laptop sends updates to the home machine, (4) O→*: the office server sends updates to all other machines.

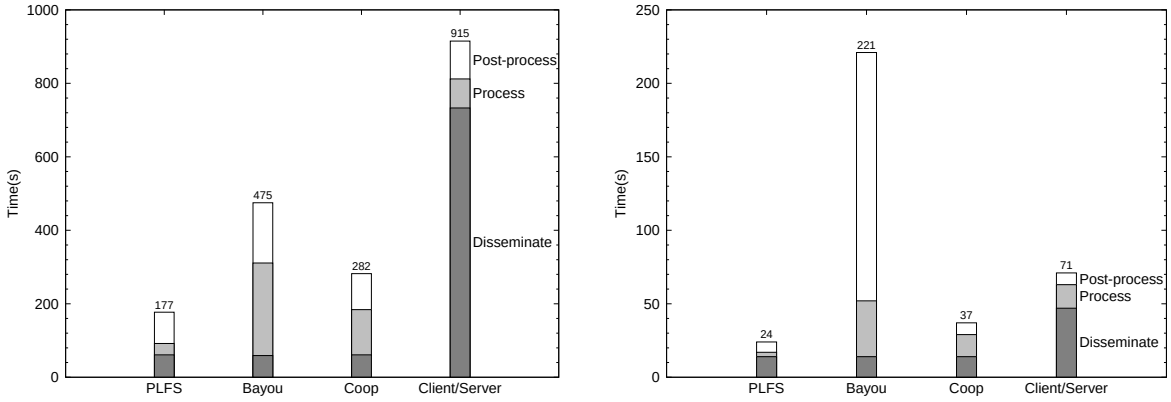
In comparing the optimized PRACTI system to a client-server system, topology independence has significant gains when the machines that need to synchronize are near one another but far from the server: in the isolated *Plane* location, the palmtop and laptop can not synchronize at all in a client-server topology; in the *Hotel* location, direct synchronization between these two devices is an order of magnitude faster than synchronizing via the server (1.7s v. 66s); and in the home location directly synchronizing co-located devices is between 3 and 20 times faster than client-server synchronization.

5.4.2 WAN-FS for Researchers

Figure 5.5 evaluates a wide-area-network file system called PLFS designed for PlanetLab and Grid researchers. The controller for PLFS is simple: for invalidations, PLFS forms a multicast tree to distribute all precise invalidations to all nodes. And, when an *INVALID* file is read, PLFS uses a DHT-based system [130] to find the nearest copy of the file; not only does this approach minimize transfer latency, it effectively forms a multicast tree when multiple concurrent reads of a file occur [6]. Like Shark [6], PLFS is designed to be convenient for allowing a user to export data from her local file system to a collection of remotely running nodes. However, unlike the read-only Shark system, PLFS supports read/write data.

We examine a 3-phase benchmark that represents running an experiment: in phase 1 *Disseminate*, each node fetches 10MB of new executables and input data from the user's home node; in phase 2 *Process*, each node writes 10 files each of 100KB and then reads 10 files from randomly selected peers; in phase 3, *Post-process*, each node writes a 1MB output file and the home node reads all of these output files. We compare PLFS to three systems: a client-server system, client-server with cooperative caching of read-only data (e.g., a Shark-like system [6]), and server-replication (e.g., a Bayou-like system [95]). All 4 systems are implemented over PRACTI.

Figure 5.5 shows performance for an experiment running on (a) 50 distributed nodes



(a) 50 distributed nodes + remote server

(b) 50 cluster nodes + remote server

Figure 5.5: Execution time for the WAN-Experiment benchmark.

each with a 5.6Mb/s connection to the Internet (we emulate this case by throttling bandwidth) and (b) 50 "cluster" nodes at university X with a switched 100Mbit/s network among them and a shared path via Internet2 to the origin server at university Y. PLFS's combination of partial replication and topology independence allows it to dominate the other designs. Compared to client/server, it is faster in both the Dissemination and Process phases due to its multicast transmission and direct data transfer. Compared to full replication, it is faster in the Process and Post-process phases because it only sends the required data. And compared to cooperative caching of read only data, it is faster in the Processing phase because data can be transferred directly between nodes.

5.4.3 Generalized APIs

APIs of SAR objects described in Chapter 3 are simple because the specific semantics that individual SAR objects leverage have been implemented within objects. When we export the SAR objects as a WAN replication library for building other WAN applications, objects will become easier to use by application developers if we modify their APIs to reflect the exact semantics they support. Therefore, by matching the semantics implemented within objects and those exhibited in the target data sets, a developer can effectively select the

appropriate objects to replicate the target data sets.

Each SAR object is extended to include a new constructor that takes three additional arguments, *updateTopology*, *dataType*, and *writeRatio*. The first argument specifies the topology of the update propagation. For example, when replicating the TPC-W catalog data or implementing the dissemination phase of the WAN-FS benchmark, *one-to-many* needs to be used as the *updateTopology*. When replicating the TPC-W order data or implementing the *post-process* phase of the WAN-FS benchmark, *many-to-one* needs to be specified instead. And when the update topology is *many-to-many*, the second and third arguments need to be considered. If the second argument is *numeric*, the inventory object is instantiated; otherwise we need to examine the third argument and decide whether to use the DQVL implementation of the TPC-W profile object. However, when no objects with the similar semantics are found, it is time to introduce new objects into our replication library.

Of course, we can extend APIs to allow scalable dissemination when the replication scale increases. But because this dissertation mainly focuses on optimizing CARP trade-offs, scalability issues are to be considered in future work.

5.5 Discussion

This chapter presents our vision and initial steps to build a replication microkernel architecture by combining the PRACTI mechanisms with the SAR design methodology. However, many open issues remain to be addressed before we can have a complete implementation of the microkernel architecture and use it to re-build the distributed TPC-W prototype.

5.5.1 Support for replicated database

The invalidation schema in the current PRACTI core does not support database systems. The data structures employed in the core support updates and invalidations to byte ranges. The data store in PRACTI manages a file in the form of byte ranges. A file is broken into

multiple ranges of bytes that can be merged or divided based on the update patterns. The data store also reserves an extra storage to store the status of a byte range. When an invalidation is applied in the system, the corresponding range of bytes are marked invalid in the associated extra storage. But this design cannot accommodate the relational database tables because data store cannot represent database tuples in byte ranges.

A potential schema to represent invalidations to database tuples is to add a status column in every writable table. For this schema to work, we need to modify the existing data store to add SQL capability. Instead of reading byte ranges from the disk, data store needs to perform SQL operations to store and retrieve database tuples. Although the status column is a part of the regular database attributes from the point of view of DBMS, the data store depends on this information to perform read and update operations.

5.5.2 Cross-object consistency

Not only could the replication microkernel architecture enhance the extensibility of our SAR approach, it also yields a framework for reasoning about the cross-object consistency. In SAR approach, consistency guarantees of a shared data set is provided by a single distributed object. And because SAR objects do not communicate with each other while managing their own shared data sets, few guarantees are provided across different objects in one application.

In the replication microkernel architecture, a single point (e.g. the PRACTI core) controls the propagation of updates on all data sets in a given application. SAR objects running inside the controller refer to their specific policies to manage the order in which local updates are sent to other nodes and remote update are applied to the local data store. No policies specify how to order between two updates when they target data sets maintained by two different objects. But once those updates are passed to the core, they will be maintained with the causally consistent prefix property as we have described in the previous section.

Chapter 6

Related Work

This chapter describes related work in data replication and systems research on the TPC-W benchmark.

6.1 General Data Replication

Replication is fundamentally difficult. For example Siegel [107] proves what has come to be known as the CAP dilemma [20]: a replication system that provides sequential *C*onsistency cannot simultaneously provide 100% *A*vailability in an environment that can be *P*artitioned. Similarly, Lipton and Sandberg describe fundamental performance limitations for any sequentially consistent algorithms [73]. As a result, systems must make compromises or optimize for specific workloads.

6.1.1 Web caching

The traditional web caching is a simple form of data replication where read-only data are replicated from the central server to client machines or proxies. All updates are performed only at the server. Clients and the server use cache invalidation protocols [47, 132] to manage the consistency of the cached data. But because current HTTP traffic is un-

cachable [38, 129], the traditional web caching provides limited availability and response time improvements to WAN services.

Many studies have addressed the importance of caching dynamic content to improve system performance and availability. Challenger et al. [23] develop an approach for consistently caching dynamic Web data that became a critical component of the 1998 Olympic Winter Games Web site. But it concerns only the single writer case. Arlitt et al. [8] study the scalability of a large online shopping system by performing workload characterization, and they conclude that linear scalability is not always adequate in case of workload bursts. They suggest efficient caching and capacity planning techniques to increase the system scalability and performance. But the fact that dynamic content is always generated by the central server limits availability and performance improvements offered by caching techniques.

6.1.2 Database replication

Replicated database systems offer the flexibility for distributed services to operate on local database replicas. Most commercial databases support data replication with an *eager* or *lazy* consistency model [48]. The eager update model considers updating every replica as part of a single transaction, which may decrease the system availability and response time when used in wide area replication. The *eager* consistency model cannot tolerate network partitions. They are therefore not suitable for WAN data replication. The lazy update model is usually preferred for WAN replication because updates are asynchronously propagated to other replicas. Although general database systems support procedures for resolving conflicts, those procedures are normally defined with database level semantics [91] and are difficult for applications to leverage on.

6.1.3 Replication with eager consistency

There are many variations of the *eager* consistency model that are designed for both distributed database and file systems.

The most obvious protocol in this category is the read-one/write-all (ROWA) protocol. In ROWA the "read-one" property yields excellent read availability and response time. But this protocol has limited write availability and response time because writes can not complete if any of the replicas are unavailable. Protocols with the read-one/write-all-available property (ROWA-A) [13, 14] perform writes synchronously only on the available replicas to improve write availability and response time when some replicas fail. But ROWA-A protocols do not tolerate network partitions, nor communication failures in general.

The primary-backup (or primary-copy) model [4, 111] is simpler to implement compared with ROWA protocol family. A variation of the primary-backup protocol proposed by Oki et al. [90] can tolerate network partitions by combining with a voting-based protocol that detects the existence of a majority partition. Alternatively, group-communication based techniques [17, 79, 83], enable the election of a new primary by actively propagating updates to all group members and constantly running membership protocols to maintain the correct memberships. The new primary can be selected from active members and the change of memberships is also broadcast to all active members as well. This class of techniques has degraded performance in WANs because the membership protocol may always need to run to constantly include/exclude certain replicas when they are mistakenly considered as crashed/recovered due to slow WAN links. In addition, all primary-server based protocols are inflexibly in favor of reads' availability and performance.

Quorum based protocols [1, 29, 45, 46, 76, 92, 116] can tolerate network partitions as long as connected replicas can form a quorum to process reads/writes. However, the reads' response time and availability of most quorum systems are worse than those of ROWA-A or primary-backup based protocols because reads usually need to query a larger

set of servers. Quorum based protocols may not be desirable to handle a read-dominated workload, e.g. a workload from interactive online applications.

Recently, studies on probabilistic quorum systems [78, 134] show that the probabilistic intersection approach increases the availability and reduces the load for quorum systems. But few practical implementations of the probabilistic quorum systems are available to validated the claim.

Some quorum based techniques use light-weight nodes, such as ghosts [93, 120] to help form quorums for processing requests. When propagating a write, a replica only sends to these nodes the timestamp and object ID of the write. Our dual-quorum invalidation protocol shares the idea in terms of replacing writes with invalidations when propagating to some replicas. But our use of invalidations also allows us to reduce the future message propagation to other replicas.

6.1.4 Replication with relaxed consistency

The *relaxed* consistency model is widely used for data replication in WANs. Studies [64, 68, 85, 95, 102, 135] have explored the space of relaxed consistency models. The Coda file system [64] employs a hierarchical architecture for replicating data volumes onto clients (smaller devises) and supports disconnected operations by clients to tolerate network partitions and client mobility. Ladin et al. [68] propose a lazy replication technique to increase the system availability and performance with the guarantee for causal ordering. All updates are processed asynchronously while queries are processed in a sequence that reflects casual ordering with support from both the information in the *label* associated with every query and the *gossip* process among replicas. The Bayou [95] replication framework uses an anti-entropy protocol to guarantee the eventual consistency of the system, and it uses version vectors and application-specific reconciliation to ensure client consistency. Saito et al. [102] focus on minimizing the space and the computation overhead using the optimistic replication approach to provide the eventual consistency. TACT [135] constructs a model

for evaluating the trade-offs between availability and consistency. The system can be tuned to provide availability that is subject to the specified consistency requirements. Both Bayou and TACT provide hooks for application developers to attach specific reconciliation rules to resolve update conflicts [114]. The design of some of our distributed objects make use of these ideas.

The drawback with the relaxed consistency model is the additional programming complexity needed to handle subtle cases. For example, because the relaxed consistency model does not impose a restrict global order on all writes, the system need to detect and resolve any conflicting writes. In some cases, the system need to depend on applications-specific resolution rules to resolve conflicts.

6.2 Semantic-aware replication approaches

While generic algorithms are forced to make compromises when used for data replication in WANs, algorithms designed based on specific semantics (data properties, workload characteristics, and update patterns) can achieve optimized trade-offs.

Existing studies [56, 100, 112] exploit type-specific properties in managing replicated data. Herlihy [56] introduces a replication technique that systematically exploits properties that are specific to data types and derives constraints on building a quorum-based replication system. Herlihy argues that understanding type-specific properties can reduce the constraints on the availability of the replication system. This idea is similar to ours except that we focus on applying such techniques to e-commerce applications and are not restricted to use only quorum-based approaches for building our replication framework.

Sussman et al. [112] use the Bancomat problem as a simple partition-aware application to evaluate the properties provided by different partitionable group membership protocols. In the Bancomat study, authors exploit the properties of both the shared data that are of the numeric type and the associated updates that reduce the numeric value by a certain amount. Because of those properties, updates are commutative and they can also

be aggregated as one update or partitioned into multiple updates as needed by the various partitionable group membership protocols evaluated by authors. Our *order*, *inventory*, and *best-seller-list* objects take advantage of the fact that updates are commutative and can be slightly reordered before the threshold is reached. The value of commutativity for simplifying consistency has also been used in write-anywhere databases [48].

Porcupine [100] is a mail service that runs on a cluster of commodity PCs. This mail service consists of a collection of data structures, each managing the different shared data sets of the mail service using specifically designed policies to achieve the desired overall system manageability, availability, and performance.

Nayate et al. examine data dissemination services with self-tuning, push-based prefetch from the server [86]. In this work, the authors argue for separating invalidations from the actual updates of the shared data. Synchronizing invalidation messages arriving at replicas enforces the system's consistency requirement, and prefetching updates in the background maximizes the hit rate at replicas, minimizes the response time, and maximizes the service availability. We could incorporate this approach to enhance the *catalog* object.

Studies described in this section are examples of semantic-aware replication solutions that achieve the optimized trade-offs for specific replication scenarios that they aim for. This dissertation differs from above studies in that it seeks to provide a framework for exploiting semantics to optimize WAN replication algorithms. Those studies are focus on individual instances of algorithms supported in our framework.

6.3 Object-oriented Replication/Distribution Approaches

Our replication framework uses the encapsulation concept from the object-oriented model to implement semantic-aware algorithms. This approach is similar to those in other systems [49, 74, 106, 121] which propose to encapsulate the complexity of structures and computations in distributed systems with objects. Both Gribble [49] and Litwin et al. [74] use distributed data structures to abstract the implementation details and provide scalable

and efficient data sharing and distribution in a cluster environment. Shapiro [106] proposes to structure a distributed system as a set of services or subsystems, each of which may be made of a number of communicating objects across the network. But this work focuses on low level components such as system processes, services, and resources. Globe objects [121] are application level objects specifically for the WAN data replication. They are similar to our distributed objects in that they both allow application programmers to make use of pre-constructed replication modules to easily invoke standard consistency algorithms with different objects and let programmers exploit application semantics in the design and implementation of individual objects. This distributed objects model provides a flexible and powerful way to build distributed applications in WAN. But to our knowledge, there is little work quantitatively evaluating the benefits of this approach in building data-oriented services, such as e-commerce applications. Our specific implementation differs from Globe in that we do not follow the same uniform internal structure of Globe objects that separate "semantics object" from "replication object". Our initial implementation found it simpler to integrate the code for object semantics and replication consistency. But the replication microkernel architecture outlines the new design where "replication objects" in Globe are integrated into the microkernel consisting of the replication primitives.

6.4 TPC-W benchmark related systems research

We use a variation of TPC-W benchmark to demonstrate the gains in availability and performance when using SAR for WAN data replication. The TPC-W is also used as the evaluation benchmark in studies on WAN data replication and computer architecture.

Walsh et al. build the TPC-W benchmark on top of TACT to demonstrate the feasibility of using TACT as a database middleware for traditional, SQL-based database applications [125]. They evaluate both the performance benefit and consistency costs of continuous consistency for their TPC-W implementation across a variety of replication scenarios and consistency bounds.

Garcia et al. [44] study the TPC-W benchmark, including its architecture, operational procedures for carrying out tests, and the performance metrics it generates. Their experimental results demonstrate that TPC-W is a useful tool for generating a standard metric of the transactional capacity of servers working in e-commerce environments. The PHARM project [118] at the University of Wisconsin focuses on the micro-architectural characterization of the TPC-W defined workload such as branch predictability, caching behaviors, and multiprocessor data sharing patterns. Amza et al. [5] characterize the bottleneck of dynamic web site benchmarks, including the TPC-W online bookstore and auction site. Their study focuses on discovering and explaining the bottleneck resources in each benchmark.

Chapter 7

Future Directions

Data replication and distribution in wide area networks (WANs) are fundamental problems for many important and popular infrastructures besides Internet edge services, such as on-demand computing, grid computing, and large-scale file systems. In addition to our vision of the unified replication architecture described in Chapter 5, the following research topics warrant further exploration.

7.1 Distributed Data Stream Management

Our experience with building distributed TPC-W objects suggests that a peer-to-peer infrastructure is more suitable than a centralized one for highly available and scalable distributed data stream management (DDSM). Distributed data stream management (DDSM) is widely employed in database systems, network monitoring, sensor networks, and manufacturing. DDSM provides a platform for managing and querying distributed data sets that change constantly in the form of stream, such as link bandwidth usage (targeted by network monitoring), chemical concentration (targeted by sensor networks), and inventory planning (targeted by manufacturing). Scalability and availability of existing DDSM systems are fundamentally limited because they rely on a central coordinator to eventually

gather and process streams of changes from distributed sources. Although there have been extensive studies on how to improve the scalability of DDSM by minimizing the monitoring and processing efforts [10, 11, 12, 108], including the communication overhead and CPU cycles, the central coordinator is still the availability and scalability bottleneck. A peer-to-peer approach could solve this bottleneck problem at the cost of reducing the accuracy of the global state information. The challenge in this work is to design the right peer-to-peer infrastructure that can offer trade-offs among availability, scalability, and accuracy of the global state. With given available trade-offs from this infrastructure, we can identify the appropriate DDSM applications.

7.2 Adaptive Replication

Ultimately, the WAN data replication problem will be solved by one data replication solution that can continuously adapt to the dynamic environment. The semantic-aware distributed objects solution works well because we design specific objects to handle the shared data under environments with various workloads (e.g. write ratios and access localities), fault-loads (e.g. failure locations and failure rate), propagation topologies, etc. However, we may also consider a system that can monitor the changes of the environment and dynamically adjust the underlying data replication strategies according to environmental changes. The basic idea is to create a set of cost functions that can estimate the availability, performance, and consistency guarantees of a given replication strategy based on a set of metrics, such as workloads, fault-loads, and update topologies. Then, the distributed objects can use cost functions to determine the most suitable replication protocols and continuously adjust internal replication protocols to adapt to any changes of the environment. Challenges in this work include the fact that too many metrics need to be considered while designing the environment monitor and cost functions. We might discover different sets of important metrics across various application classes. Furthermore, we also need to evaluate the cost and benefit of dynamically switching replication protocols.

7.3 Dynamic Replica Placement

The advantage provided by dynamic replication is that we can effectively respond to the unpredictable geographical shift of the service demand. The unpredictable shift could be caused by link failures at backbones, network congestion, and client access pattern changes. If a system can not respond to this change in demand well, some server replicas might be overloaded. Therefore, the performance and availability of the system will decrease. It is necessary for replication systems to have the ability to capture the access patterns and network failure patterns. Once the system notices a change, it can deploy new replicas at the estimated hot-spots to accommodate the load increase at specific regions.

However, the advantage of dynamic replica placement for the edge service model is unclear. On one hand, it appears that the system could effectively achieve better availability and efficiency by dynamically creating replicas at the Internet hot-spot as the workload and fault-load vary overtime. On the other hand, the cost for creating new replicas, tearing down obsolete replicas, and maintaining consistency across all replicas can potentially offset the benefit gained with the dynamic placement strategy. Therefore, there are two questions to be investigated: 1. How much benefit do we gain with dynamic replication? 2. What is the minimal cost for creating and tearing down replicas on-demand?

Chapter 8

Conclusions

Quantitatively, we have showed the limits of the existing Internet architecture using a trace-based simulator. Using the failure model developed by Dahlin et al. [35], our study shows that using traditional web caching techniques alone cannot cause significant improvement on the end-to-end Internet service availability. Only with a more sophisticated infrastructure, like the edge service infrastructure, which combines overlay routing and more aggressive replication techniques (e.g. server replication and selection), can service providers offer revolutionary end-to-end improvements to their Internet services.

We present the semantic-aware replication (SAR) that offers the optimized CARP (consistency, availability, response time, and partition-resilience) trade-offs when replicating dynamic data in the edge service architecture. Although generic WAN replication approaches have been proven incapable of providing the optimized CARP trade-offs, SAR reduces the constraints in WAN replication by leveraging the semantics of shared data sets and encapsulating the consistency maintenance using the object-oriented approach. SAR solves the replication of dynamic data in WANs for the edge service architecture that distributes both business logic and data to edge servers to minimize access to a central database. In the SAR approach, data shared among the edge servers are encapsulated within distributed objects that are aware of the semantics (i.e. workloads, update topolo-

gies, and data properties) of the underlying shard data. Replication techniques implemented in each object are specially tuned to leverage on the semantics of the corresponding data set. Access to the shared data sets is restricted through narrower interfaces of objects that hide the complexity of WAN data replication. The experimental results confirm our claim that SAR offers the optimized CARP trade-offs that are difficult to achieve by generic WAN replication techniques. For instance, our prototype TPC-W system continues to process requests with the same throughput and response time before, during, and after a 50-second network partition that separates edge servers and the backend server. And the response time of our system is nearly five times better than that of the traditional centralized system, in which end users connect to web servers via slow WAN links.

Then, we discussed the dual-quorum with volume leases (DQVL) protocol to optimize the multi-writer/multi-reader replication scenario that is managed by the *profile* object in SAR. DQVL is a novel data replication algorithm that provides the key missing piece to achieve highly-available, low-latency, and consistent data replication for a range of Internet services. In particular, dual-quorum replication optimizes these properties for data elements that can be both read and written from many locations, but whose reads and writes exhibit locality in two dimensions: (1) at any given time access to a given element tends to come from a single node and (2) reads tend to be followed by other reads and writes tend to be followed by other writes. Our dual-quorum replication protocol combines ideas from volume leases and quorum based techniques. Through both analytical and experimental evaluations, we show that, for the important special case of single-node OQS read quorums, the average read response time of DQVL can approach a node's local read time, making the read performance of this approach competitive with ROWA-A epidemic algorithms such as Bayou. At the same time, the overall availability of DQVL is competitive with the majority quorum protocol for the targeted workloads.

Finally, we outline our vision of the unified replication architecture that applies the SAR design approach atop the PRACTI replication mechanisms. The unified architec-

ture allows the separation of policies from mechanisms. While PRACTI provides flexible mechanisms to implement replication policies that may demand partial-replication, arbitrary consistency, and topology independence, actual policies are specified in forms of distributed objects as in SAR. The unified architecture advances the SAR approach by providing a single environment for leveraging semantics of shared data sets across classes of Internet services. We have a framework for reasoning about cross-object consistency that is difficult to argue when objects are independently built. The development of distributed objects becomes easier and faster because we can reuse mechanisms exported from PRACTI.

Bibliography

[1] D. Agrawal and A. Abbadi. The Tree Quorum Protocol: An Efficient Approach for Managing Replicated Data. In *Proceedings of the Sixteenth International Conference on Very Large Data Bases*, 1990.

[2] Inc. Akamai Technologies. Akamai-The Business Internet - A Predictable Platform for Profitable E-Business. http://www.akamai.com/BusinessInternet/whitepaper_business_internet.pdf, 2004.

[3] Inc. Akamai Technologies. Turbo-Charging Dynamic Web Sites with Akamai EdgeSuite. White paper, Akamai Technologies, Inc., 2004.

[4] P. Alsberg and J. Day. A Principle for Resilient Sharing of Distributed Resources. In *the 2nd Intl. Conference on Software Engineering*, 1976.

[5] C. Amza, E. Cecchet, A. Chanda, A. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel. Bottleneck Characterization of Dynamic Web Site Benchmarks. Technical Report TR02-391, Rice University, Feb 2002.

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

[7] Network Appliance. Internet content adaptation protocol (icap). DS-2326, June 2000.

[8] M. Arlitt, D. Krishnamurthy, and J. Rolia. Characterizing the Scalability of a Large Web-based Shopping System. *ACM Transactions on Internet Technology*, June 2001.

[9] A. Awadallah and M. Rosenblum. The vMatrix: A Network of Virtual Machine Monitors for Dynamic Content Distribution. In *7th International Workshop on Web Content Caching and Distribution*, August 2002.

[10] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and Issues in Data Stream Systems. In *Principles of Database Systems (PODS 2002)*, June 2002.

[11] B. Babcock and C. Olston. Distributed Top-K Monitoring. In *Proceedings of 2003 SIGMOD*, June 2003.

[12] D. Barbara and H. Garcia-Molina. The Demarcation Protocol: A technique for maintaining linear arithmetic constraints in distributed database systems. In *Proceedings of the International Conference on Extending Database Technology*, March 1992.

[13] P. Bernstein and N. Goodman. An algorithm for concurrency control and recovery in replicated distributed databases. In *ACM Transactions on Database Systems*, December 1984.

[14] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control adn Receivery in Database Systems.* Addison Wesley, 1987.

[15] S. Bhattacharjee, M. H. Ammar, E. W. Zegura, N. Shah, and Z. Fei. Application Layer Anycasting. In *IEEE INFOCOM'97*, 1997.

[16] S. Bhattacharjee, K. Calvert, and E. Zegura. Self-organizing wide area network caches. Technical Report GIT-CC-97/31, Georgia Tech, 1997.

[17] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, 1987.

[18] D. Black, D. Golub, D. Julin, R. Rashid, R. Draves, R. Dean, A. Forin, J. Barrera, H. Tokuda, G. Malan, and D. Bohman. Microkernel operating system architecture and Mach. *Journal of Information Processing*, 14(4), March 1992.

[19] M. Blaze and R. Alonso. Dynamic Hierarchical Caching in Large-Scale Distributed File Systems. In *Proceedings of the 12th International Conference on Distributed Computing Systems*, pages 521–528, June 1992.

[20] E. Brewer. Lessons from giant-scale services. In *IEEE Internet Computing*, July/August 2001.

[21] P. Cao, J. Zhang, and K. Beach. Active Cache: Caching Dynamic Contents on the Web. In *Proceedings of Middleware 98*, 1998.

[22] N. Cardwell, S. Savage, and T. Anderson. Modeling TCP Latency. In *Infocom'00*, 2000.

[23] J. Challenger, P. Dantzig, and A. Iyengar. A Scalable and Highly Available System for Serving Dynamic Data at Frequently Accessed Web Sites. In *Proceedings of ACM/IEEE, Supercomputing '98 (SC98)*, November 1998.

[24] J. Challenger, P. Dantzig, and A. Iyengar. A Scalable System for Consistently Caching Dynamic Web Data. In *Proceedings of IEEE Infocom*, March 1999.

[25] J. Challenger, A. Iyengar, K. Witting, C. Ferstat, and P. Reed. A Publishing System for Efficiently Creating Dynamic Web Content. In *Proceedings of IEEE Infocom*, March 2000.

[26] B. Chandra. Web workloads influencing disconnected services access. Master's thesis, University of Texas at Austin, 2001.

[27] B. Chandra, M. Dahlin, L. Gao, A. Khoja, A. Razzaq, and A. Sewani. Resource

management for scalable disconnected access to web services. In *WWW10*, May 2001.

[28] L. Cherkasova, Y. Fu, W. Tang, and A. Vahdat. Measuring and characterizing end-to-end internet service performance. *ACM Trans. Inter. Tech.*, 3(4):347–391, 2003.

[29] S. Cheung, M. Ahamad, and M. Ammar. The grid protocol: a high performance scheme for maintaining replicated data. In *Proceedings of the Sixth International Conference on Data Engineering*, pages 438–445, 1990.

[30] S. Cheung, M. Ahamad, and M. H. Ammar. Optimizing Vote and Quorum Assignments for Reading and Writing Replicated Data. *IEEE Transactions on Knowlegde and Data Engineering*, 1(3):387–397, September 1989.

[31] IBM Corporation. MQSeries: An Introduction to Messaging and Queueing. Technical Report GC33-0805-01, IBM Corporation, July 1995. ftp://ftp.software.ibm.com/software/mqseries/pdf/horaa101.pdf.

[32] Transaction Processing Performance Council. TPC BENCHMARK W. http://www.tpc.org/tpcw/spec/-tpcw_V1.8.pdf, 2002.

[33] C. Cunha, A. Bestavros, and M. Crovella. Characteristics of WWW Traces. Technical Report TR-95-010, Boston University Department of Computer Science, April 1995.

[34] H. Custer. Inside windwos nt. Microsoft Press.

[35] M. Dahlin, B. Chandra, L. Gao, and A. Nayate. End-to-end WAN Service Availability. *IEEE/ACM Transactions on Networking*, April 2003.

[36] M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication for Large-Scale Systems. Technical report, University of Texas at Austin Department of Computer Sciences, 2004.

[37] A. Datta, K. Dutta, H. Thomas, D. VanderMeer, Suresha, and K. Ramamritham. Proxy-Based Acceleration of Dynamically Generated Content on the World Wide Web: An Approach and Implementation. In *SIGMOD Conference*, 2002.

[38] B. Duska, D. Marwood, and M. Feeley. The Measured Access Characteristics of World-Wide-Web Client Proxy Caches. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, December 1997.

[39] V. Duvvuri, P. Shenoy, and R. Tewari. Adaptive Lease: A Strong Consistency Mechanism for the World Wide Web. In *Proceedings of IEEE Infocom*, March 2000.

[40] Z. Fei, S. Bhattacharjee, E. Zegura, and M. Ammar. A Novel Server Selection Technique for Improving the Response Time of a Replicated Service. In *Proceedings of IEEE Infocom*, March 1998.

[41] S. Floyd. Connections with Multiple Congested Gateways in Packet-Switched Networks Part 1: One-way Traffic. *Computer Communications Review*, 21(5), October 1991.

[42] M. Frigo. The Weakest Reasonable Memory Model. Master's thesis, MIT, 1988.

[43] L. Gao, M. Dahlin, A. Nayate, J. Zheng, and A. Iyengar. Improving Availability and Performance with Application-Specific Data Replication. *IEEE Transactions on Knowledge and Data Engineering*, March 2005.

[44] D. Garcia and J. Garcia. TPC-W E-Commerce Benchmark Evaluation. *IEEE Computer*, February 2003.

[45] H. Garcia-Molina and D. Barbara. How to Assign Votes in a Distributed System. In *Journal of the ACM 32 (4)*, 1985.

[46] D. Gifford. Weighted voting for replicated data. In *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*, December 1979.

[47] C. Gray and D. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 202–210, 1989.

[48] J. Gray, P.Helland, P. E. O'Neil, and D. Shasha. Dangers of Replication and a Solution. In *Proceedings of SIGMOD*, pages 173–182, 1996.

[49] S. Gribble, E. Brewer, J. Hellerstein, and D. Culler. Scalable, Distributed Data Structures for Internet Service Construction. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, October 2000.

[50] J. Griffioen and R. Appleton. Reducing File System Latency Using A Predictive Approach. In *Proceedings of the Summer 1994 USENIX Conference*, June 1994.

[51] J. Gwertzman and M. Seltzer. The case for geographical pushcaching. In *HOTOS95*, pages 51–55, May 1995.

[52] J. Gwertzman and M. Seltzer. World-Wide Web Cache Consistency. In *Proceedings of the 1996 USENIX Technical Conference*, January 1996.

[53] G. Hamilton and P. Kougiouris. The spring nucleus: A microkernel for objects. Technical report, Sun Microsystems Laboratories, Inc., 1993.

[54] N. Harvey, M. Jones, S. Saroiu, M. Theimer, and A. Wolman. Skipnet: A scalable overlay network with practical locality properties. In *4th USENIX Symposium on Internet Technologies and Systems (USITS '03)*, March 2003.

[55] J. Hennessy and D. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 2nd edition, 1996.

[56] M. Herlihy. A Quorum-Consensus Replication Method for Abstract Data Types. *ACM Transactions on Computer Systems*, 4(1):32–53, February 1986.

[57] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[58] P. Hutto and M. Ahamad. Slow memory: Weakening consistency to enhance concurrency in distributed shared memories. In *Proceedings of the Tenth International Conference on Distributed Computing Systems*, 1990.

[59] Java Message Service (JMS). http://java.sun.com/products/jms.

[60] JORAM. http://www.objectweb.org/joram.

[61] A. Joseph, A. deLespinasse, J. Tauber, D. Gifford, and M. Kaashoek. Rover: A Toolkit for Mobile Information Access. In *Proceedings of the Fifteenth ACMSymposium on Operating Systems Principles*, December 1995.

[62] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proceedings of the Twenty-ninth ACM Symposium on Theory of Computing*, 1997.

[63] Web performance index. Internet World, August 1999 – April 2000 1999-2000.

[64] J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems*, 10(1):3–25, February 1992.

[65] M. Korupolu and M. Dahlin. Coordinated Placement and Replacement for Large-Scale Distributed Caches. In *Proceedings of the 1999 IEEE Workshop on Internet Applications*, June 1999.

[66] B. Krishnamurthy and C. Wills. Analyzing factors that influence end-to-end Web performance. *Computer Networks (Amsterdam, Netherlands: 1999)*, 33(1–6):17–32, 2000.

[67] A. Kumar. Comparative Performance Analysis of Versions of TCP in a Local Networkwith a Lossy Link. *IEEE/ACM Transactions on Networking*, 6(4), August 1998.

[68] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing High Availability Using Lazy Replication. *ACM Transactions on Computer Systems*, 10(4):360–391, November 1992.

[69] T. Lakshman and U. Madhow. The Performance of TCP/IP for Networks with High Bandwidth-delay Products and Random Loss. *IEEE/ACM Transactions on Networking*, June 1997.

[70] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), July 1978.

[71] L. Lamport. On interprocess communications. *Distributed Computing*, pages 77–101, 1986.

[72] D. Li and D. Chariton. Scalable Web Caching of Frequently Updated Objects Using Reliable Multicast. In *Proceedings of the Second USENIX Symposium on Internet Technologies and Systems*, pages 1–12, Oct 1999.

[73] R. Lipton and J. Sandberg. PRAM: A Scalable Shared Memory. Technical Report CS-TR-180-88, Princeton, 1988.

[74] W. Litwin, M-A. Neimat, and D. Schneider. LH* - A Scalable, Distributed Data Structure. In *ACM Transactions on Database Systems*, December 1996.

[75] C. Liu and P. Cao. Maintaining Strong Cache Consistency in the World-Wide Web. In *Proceedings of the Seventeenth International Conference on Distributed Computing Systems*, May 1997.

[76] M. Maekawa. A Algorithm for Mutual Exclusion in Decentralized Systems. In *ACM Transactions on Computing Systems 3 (2)*, 1985.

[77] D. Malkhi and M. Reiter. An Architecture for Survivable Coordination in Large Distributed Systems. *IEEE Transactions on Knowledge and Data Engineering*, pages 187–202, March 2000.

[78] D. Malkhi, M. Reiter, A. Wool, and R. Wright. Probabilistic quorum systems. *Inf. Comput.*, 170(2):184–206, 2001.

[79] D. Malki, K. Birman, A. Schiper, and A. Ricciardi. Uniform Actions in Asynchronous Distributed Systems. In *ACM SIGOPS-SIGACT*, August 1994.

[80] M. Mathis, J. Semke, J. Mahdavi, and T. Ott. The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm. *Computer Communications Review*, 27(3), July 1997.

[81] J. Mogul. A Design for Caching in HTTP 1.1 Preliminary Draft. Technical report, Internet Engineering Task Force (IETF), January 1996. Work in Progress.

[82] A. Moissis. SYBASE replication server: A practical architechture for distributing and sharing corporate information. Technical report, SYBASE Inc, March 1994.

[83] D L. Moser, Y. Amir, P. Melliar-Smith, and D. Agarwal. Extended virtual synchrony. In *Proceedings of the Fourteenth International Conference on Distributed Computing Systems*, June 1994.

[84] L. Mummert, M. Ebling, and M. Satyanarayanan. Exploiting weak connectivity for mobile file access. In *Proceedings of the Fifteenth ACMSymposium on Operating Systems Principles*, December 1995.

[85] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, December 2002.

[86] A. Nayate, M. Dahlin, and A. Iyengar. Data Invalidation and Prefetching for Transparent Edge-Service Replication. Technical Report TR-03-44, University of Texas at Austin Department of Computer Sciences, Nov 2002.

[87] A. Nayate, M. Dahlin, and A. Iyengar. Transparent Information Dissemination. In *ACM/IFIP/USENIX 5th International Middleware Conference*, October 2004.

[88] NIST Net Home Page. http://snad.ncsl.nist.gov/itg/nistnet/.

[89] B. Noble, M. Satyanarayanan, D. Narayanan, J. Tilton, J. Flinn, and K. Walker. Agile Application-Aware Adaptation for Mobility. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, October 1997.

[90] B. Oki and B. liskov. Viewstamped replication: A general primary copy method to support highly available distributed systems. In *Proceedings of the Seventh Symposium on the Principles of Distributed Computing*, August 1998.

[91] Oracle7 Server Distributed Systems: Replicated Data. http://www.oracle.com/products/oracle7/server/whitepapers/ replication/html/index, 1994.

[92] J. Paris and D. Long. Efficient Dynamic Voting Algorithms. In *Int'l Conference on Data Engineering*, 1988.

[93] JF Paris and D. Long. Voting with Regenerable Volatile Witnesses. In *Proceedings of the Seventh International Conference on Data Engineering*, pages 112–119, 1991.

[94] V. Paxson. *Measurements and Analysis of End-to-End Internet Dynamics*. PhD thesis, University of California, Berkeley, April 1997.

[95] K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and A. Demers. Flexible Update Propagation for Weakly Consistent Replication. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, October 1997.

[96] E. Pierce and L. Alvisi. A Framework for Semantic Reasoning about Byzantine Quorum Systems. In *Proceedings of the Twentieth Symposium on the Principles of Distributed Computing*, pages 317–319, August 2001.

[97] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *2001 ACM SIGCOMM Conference*, 2001.

[98] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware*, November 2001.

[99] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrman, C. Kaiser, S. Langloisand P. Léonard, and W. Neuhauser. Overview of the Chorus distributed operating system. In *Workshop on Micro-Kernels and Other Kernel Architectures*, pages 39–70, April 1992.

[100] Y. Saito, B. Bershad, and H. Levy. Manageability, Availability and Performance in Porcupine: A Highly Scalable, Cluster-based Mail Service. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, December 1999.

[101] Y. Saito, C. Karamanolis, M. Karlsson, and M. Mahalingam. Taming aggressive replication in the pangaea wide-area file system. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, December 2002.

[102] Y. Saito and H. Levy. Optimistic Replication for Internet Data Services. In *Proceedings of the Fourteenth International Conference on Distributed Computing*, October 2000.

[103] P. Sarkar and J. Hartman. Efficient Cooperative Caching using Hints. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pages 35–46, October 1996.

[104] M. Satyanarayanan, J. Kistler, P. Kumar, M. Okasaki, E. Siegel, and D. Steere. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.

[105] S. Savage, A. Collins, E. Hoffman, J. Snell, and T. Anderson. The End-to-end Effects of Internet Path Selection. In *Proceedings of the ACM SIGCOMM '99 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 289–299, September 1999.

[106] Marc Shapiro. Structure and Encapsulation in Distributed Systems: the Proxy Principle. In *Proceedings of the Sixth International Conference on Distributed Computing Systems*, May 1986.

[107] A. Siegel. *Performance in Flexible Distributed File Systems*. PhD thesis, Cornell, 1992.

[108] N. Soparkar and A. Silberschatz. Data-value partitioning and virtual messages. In *Proceeding of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, April 1990.

[109] Charles Sterling. Programming Best Practices with Microsoft Message Queuing Services (MSMQ). http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnmqqc/html/msmqbest.asp.

[110] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *2001 ACM SIGCOMM Conference*, 2001.

[111] M. Stonebraker and E. Neuhold. Concurrency control and consistency of multiple copies of data in distributed ingres. *IEEE Transactions on Software Engineering*, 3(3):188–194, May 1979.

[112] J. Sussman and K. Marzullo. The Bancomat Problem: An Example of Resource Allocation in a Partitionable Asynchronous System. In *International Symposium on Distributed Computing*, pages 363–377, 1998.

[113] A. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms*, chapter Consistency and Replication. Prentice Hall, 2002.

[114] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the Fifteenth ACMSymposium on Operating Systems Principles*, pages 172–183, December 1995.

[115] R. Tewari, M. Dahlin, H. Vin, and J. Kay. Design Considerations for Distributed Caching on the Internet. In *Proceedings of the Nineteenth International Conference on Distributed Computing Systems*, May 1999.

[116] R. Thomas. A Majority Consensus Approach to Concurrency Control for Multiple Copy Database. In *ACM Transactions on Database Systems*, pages 180–209, June 1979.

[117] G. Tomlinson, H. Orman, M. Condry, J. Kempf, and D. Farber. Extensible proxy services framework. IETF-Draft draft-tomlinson-epsfw-00.txt, IETF, July 2000. Expires January 11, 2001.

[118] The PHARM Project at the University of Wisconsin. http://www.ece.wisc.edu/ pharm/tpcw/.

[119] A. Vahdat, M. Dahlin, T. Anderson, and A. Aggarwal. Active Naming: Flexible Location and Transport of Wide-Area Resources. In *The Second USENIX Symposium on Internet Technologies and Systems*, October 1999.

[120] R. van Renesse and A. Tanenbaum. Voting with Ghosts. In *Proceedings of the Eighth International Conference on Distributed Computing Systems*, pages 456–462, 1988.

[121] M. van Steen, P. Homburg, and S. Tanenbaum. Globe: A Wide-Area Distributed System. Technical report, Vrije Universiteit, March 1999.

[122] A. Venkataramani, R. Kokku, and M. Dahlin. TCP-Nice: A mechanism for background transfers. In *OSDI02*, December 2002.

[123] A. Venkataramani, P. Weidmann, and M. Dahlin. Bandwidth constrained placement in a wan. In *Proceedings of the Twentieth Symposium on the Principles of Distributed Computing*, August 2001.

[124] A. Venkataramani, P. Yalagandula, R. Kokku, S. Sharif, and M. Dahlin. "potential costs and benefits of long-term prefetching for content-distribution". In *Proceedings of the 2001 Web Caching and Content Distribution Workshop*, June 2001.

[125] K. Walsh, A. Vahdat, and J. Yang. Enabling Wide-Area Replication of Database Services with Continuous Consistency. Unpublished Manuscript.

[126] J. Wang, Y. Zhang, and S. Keshav. Understanding End-to-End Performance: Testbed and Primary Results. In *IEEE Global Internet Symposium*, 2001.

[127] A. Whitaker, M. Shaw, and S. Gribble. Scale and Performance in the Denali Isolation Kernel. In *OSDI02*, December 2002.

[128] TPC-W performance result in price/performance . http://www.tpc.org/tpcw/results/tpcw_price_perf_results.asp.

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

[130] P. Yalagandula and M. Dahlin. A Scalable Distributed Information Management System. In *Proceedings of the ACM SIGCOMM '04 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, August 2004.

[131] J. Yin, L. Alvisi, M. Dahlin, and A. Iyengar. Engineering server-driven consistency for large scale dynamic web services. In *Proceedings of the Tenth International World Wide Web Conference*, May 2001.

[132] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Volume Leases to Support Consistency in Large-Scale Systems. *IEEE Transactions on Knowledge and Data Engineering*, February 1999.

[133] C. Yoshikawa, B. Chun, P. Eastham, A. Vahdat, T. Anderson, and D. Culler. Using Smart Clients to Build Scalable Services. In *Proceedings of the 1997 USENIX Technical Conference*, January 1997.

[134] H. Yu. Signed quorum systems. In *PODC '04: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 246–255, 2004.

[135] H. Yu and A. Vahdat. The Costs and Limits of Availability for Replicated Services. In *Proceedings of the Eightteenth ACM Symposium on Operating Systems Principles*, 2001.

[136] H. Yu and A. Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Transactions on Computer Systems*, pages 239–282, August 2002.

[137] H. Yu and A. Vahdat. Minimal Cost Replication for Availability. In *Proceedings of the Twenty-First Symposium on the Principles of Distributed Computing*, 2002.

[138] Y. Zhang, V. Paxson, and S. Shenkar. The Stationarity of Internet Path Properties: Routing, Loss, and Throughput. Technical report, AT&T Center for Internet Research at ICSI, http://www.aciri.org/, May 2000.

[139] B. Zhao, L. Huang, J. Stribling, S. Rhea, A. Joseph, and J. Kubiatowicz. Tapestry: A Global-scale Overlay for Rapid Service Deployment. *IEEE Journal on Selected Areas in Communications*, 2003. Special Issue on Service Overlay Networks.

Vita

Lei Gao was born in Kunming, Yunnan China on April 8, 1977, the son of Zhen Gao and Ruihua Gao. After completing his second school year at Yunnan No. 1 High School in 1994, he came to the United State as an exchange student. One year later, he entered Western Michigan University in Kalamazoo, Michigan. In the spring of 1997, he transfered to the University of Texas at Austin in Texas. He received the degree of Bachelor of Science from the University of Texas at Austin in August 1998. During the following years he worked as a software analyst at J.D. Edwards & Company in Denver, Colorado. In the spring of 2000, he entered the Graduate School of the University of Texas at Austin and received the degree of Master of Art in December 2001.

Permanent Address: 187 Xinying Rd.
S. Weilong Complex
Building 6, #301
Kunming, Yunnan P.R. China 650223

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay and

James A. Bednar.