UNIVERSITY of CALIFORNIA
Santa Barbara

Clustering, Resource Management, and Replication Support for Scalable Network Services

A dissertation submitted in partial satisfaction of the
requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Kai Shen

Committee in charge:

Professor Tao Yang, Chair
Professor Anurag Acharya
Professor Linda R. Petzold
Professor Klaus E. Schauser

September 2002

The dissertation of Kai Shen is approved:

Anurag Acharya

Linda R. Petzold

Klaus E. Schauser

Tao Yang, Committee Chair

July 2002

Clustering, Resource Management, and Replication Support for Scalable Network Services

by

Kai Shen

Acknowledgements

I want to take this opportunity to thank my advisor Professor Tao Yang for his invaluable advice and instructions to me on selecting interesting and challenging research topics, identifying specific problems for each research phase, as well as preparing for my future career.

I also want to thank members of my research group, Lingkun Chu, and Hong Tang, for their incessant support in the forms of technical discussion, system co-development, cluster administration, and other research activities.

Finally, I want to thank my dissertation committee members, Professor Anurag Acharya, Professor Linda R. Petzold, and Professor Klaus E. Schauser for their insightful comments over my dissertation. I also want to thank Professor Acharya and Professor Schauser for their invaluable advice and help during my career search.

This dissertation study was supported in part by NSF CCR-9702640, NSF ITR/ACS-0082666, and NSF ITR/ACS-0086061.

Curriculum Vitæ

Kai Shen

Personal

Name	Kai Shen
Date of Birth	September 19, 1976
Place of Birth	Nanjing, China
Nationality	China
Email	kshen@cs.ucsb.edu

Education

1992–1996	B.S. in Computer Science and Engineering, Shanghai Jiaotong University, Shanghai, China.

Publications

- Kai Shen, Hong Tang, Tao Yang, and Lingkun Chu, Integrated Resource Management for Cluster-based Internet Services. To appear in Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI'02), December 2002.

- Kai Shen, Tao Yang, and Lingkun Chu, Cluster Load Balancing for Fine-grain Network Services. In Proceedings of International Parallel and Distributed Processing Symposium (IPDPS'02), April 2002.
- Kai Shen, Tao Yang, Lingkun Chu, JoAnne L. Holliday, Douglas A. Kuschner, Huican Zhu, Neptune: Scalable Replica Management and Programming Support for Cluster-based Network Services. In Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS'01), pages 197-208, March 2001.
- Hong Tang, Kai Shen, and Tao Yang, Program Transformation and Runtime Support for Threaded MPI Execution on Shared Memory Machines. ACM Transactions on Programming Languages and Systems, Vol. 22, No. 4, pages 673-700, 2001.
- Kai Shen, Xiangmin Jiao and Tao Yang, S^+: Efficient 2D Sparse LU Factorization on Parallel Machines. In SIAM Journal on Matrix Analysis and Applications (SIMAX), Vol. 22, No. 1, pages 282-305, 2000.
- Kai Shen, Hong Tang, and Tao Yang, Adaptive Two-level Thread Management for Fast MPI Execution on Shared Memory Machines. In Proceedings of IEEE/ACM High Performance Networking and Computing

Conference (SC'99), November, 1999.

- Hong Tang, Kai Shen, and Tao Yang, Compile/Run-time Support for Threaded MPI Execution on Multiprogrammed Shared Memory Machines. In Proceedings of 7th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'99), pages 107-118, May 1999.

- Bin Jiang, Steven Richman, Kai Shen, and Tao Yang, Fast Sparse LU Factorization with Lazy Space Allocation. In Proceedings of SIAM Parallel Processing Conference on Scientific Computing, March 1999.

- Anurag Acharya, Huican Zhu, and Kai Shen, Adaptive Algorithm for Cache-efficient Trie Search. In ACM/SIAM Workshop on Algorithm Engineering and Experimentation (ALENEX'99), January 1999.

- Kai Shen, Xiangmin Jiao and Tao Yang, Elimination Forest Guided 2D Sparse LU Factorization. In Proceedings of 10th ACM Symposium on Parallel Architectures and Algorithms (SPAA'98), pages 5-15, June 1998.

Abstract

Clustering, Resource Management, and Replication Support for Scalable Network Services

by

Kai Shen

With the increasing demand of providing highly scalable, available and easy-to-manage services, the deployment of large-scale complex server clusters has been rapidly emerging in which service components are usually partitioned, replicated, and aggregated. This dissertation investigates techniques in building a middleware system, called *Neptune*, that provides clustering support for scalable network services. In particular, Neptune addresses three specific aspects in support of network service clustering: 1) the development of a flexible and scalable clustering architecture with efficient load balancing support for fine-grain services; 2) the design and implementation of an integrated resource management framework that combines the "response time"-based service quality, overall resource utilization efficiency, and service differentiation support; and 3) the design and implementation of a service replication framework fo-

cusing on providing flexible replica consistency, performance scalability, and failure recovery support. Neptune has been implemented on Linux and Solaris clusters and a number of applications have been successfully deployed on Neptune platforms, including a large-scale document search engine.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Statement and Motivation

The ubiquity of the Internet and various intranets has resulted in the widespread availability of services and applications accessible through the network. Examples include document search engines, online digital libraries, discussion forums, and electronic commerce [3, 13, 33, 42, 48]. These network services are exploding in popularity for several reasons. First, they are easier to manage and evolve than their offline counterparts by eliminating the need for software distribution and compatibility issues with multiple platforms and versions. Secondly, they make it economical to deliver resource-intensive applications and services to a large number of users in real-time. The growth of popularity for network services, however, poses challenges for the design of server systems in terms of scalability, availability, and manageability. *Scalability* requires that the increase in hardware resources can maintain a corresponding increase in system throughput and storage capacity. *Availability* requires the services to be operational despite transient hardware or software failures. *Manageability* calls for the ease of extending, reconfiguring, and maintaining the service infrastructure.

Because of the recent advances in clustering technologies and its cost-effectiveness in achieving high availability and incremental scalability, computer clusters are increasingly recognized as the architecture of choice for scalable network services, especially when the system experiences high growth in service evolution and user demands [9, 40, 67, 85]. Within a large-scale complex service cluster, service components are usually partitioned, replicated, and aggregated. *Partitioning* is introduced when the service processing requirement or data volume exceeds the capacity of a single server node. Service *replication* is commonly employed to improve the system availability and provide load sharing. In addition, the service logic itself may be too complicated such that it needs to be partitioned into multiple service components. Partial results may need to be *aggregated* across multiple data partitions or multiple service modules, and then delivered to external users.

Despite its importance, service clustering to achieve high scalability, availability, and manageability remains a challenging task for service authors in designing and deploying services. And this is especially true for services with frequently updated persistent service data. In recognizing this importance and the associated challenges, my thesis is that

> *It is possible to build a flexible, scalable, and highly available infrastructure supporting service clustering, load balancing, resource management, and service replication for clustered-based network services with highly concurrent request traffic and frequently updated persistent service data.*

In other words, the main goal of this study is to answer the following question. Given a service application running on a single machine with a modest amount of service data, how can such a service be expanded quickly to run on a cluster environment for handling a large volume of concurrent request traffic with large-scale persistent service data? This dissertation investigates techniques in building a middleware system, called *Neptune*, that provides clustering support for scalable network services [69, 70, 71]. In particular, it contains the following contributions to establish my thesis:

- The development of a flexible and scalable clustering architecture with efficient load balancing support for fine-grain services.

- The design and implementation of an integrated resource management framework that combines the "response time"-based service quality, overall system resource utilization efficiency, and service differentiation support.

- The design and implementation of a service replication framework that focuses on providing flexible replica consistency, performance scalability, and failure recovery support to large-scale network services based on a variety of underlying data management mechanisms.

- Successful deployment of a number of applications on the proposed and developed system, including a document search engine, a scientific data mining application, and three other services involving frequently updated persistent service data.

1.1.1 Neptune Clustering Architecture with Load Balancing Support

The goal of Neptune clustering architecture is to propose a simple, flexible yet efficient model in aggregating and replicating network service modules. Neptune employs a loosely-connected and functionally-symmetrical clustering architecture for high scalability and availability. This architecture also allows complex multi-tier services to be easily supported. In addition, Neptune provides simple programming interfaces that make it easy to deploy existing applications and shield application programmers from clustering complexities. Overall, this architecture serves as the basis for further clustering supports including data replication, service discovery, load balancing, resource management, failure detection and recovery.

Previous research has proposed and evaluated various load balancing policies for cluster-based distributed systems [9, 15, 28, 32, 43, 57, 61, 82, 83]. These studies are mainly focused on coarse-grain computation and they often ignore fine-grain jobs by simply processing them locally. In the context of network services, with the trend toward delivering more feature-rich services in real time, large number of fine-grain sub-services need to be aggregated within a short period of time. This dissertation investigates cluster load balancing techniques with the focus on fine-grain services. Based on simulation and experimental studies, this dissertation finds that the random polling policy with small poll sizes are well-suited for fine-grain network services. And discarding slow-responding polls can further improve system performance.

1.1.2 Quality-Aware Resource Management

Previous studies show that the client request rates for Internet services tend to be bursty and fluctuate dramatically from time to time [13, 23, 27]. For example, the daily peak-to-average load ratio at Internet search service Ask Jeeves (`www.ask.com`) is typically 3:1 and it can be much higher and unpredictable in the presence of extraordinary events. Over-provisioning system resources for a service site to accommodate the potential peak will not be cost-effective. As a consequence, it is desirable to achieve efficient resource utilization for those services under a wide range of load conditions.

Network clients typically seek services interactively and maintaining reasonable response times is imperative. In addition, providing differentiated service qualities and resource allocation to multiple service classes can also be desirable at times, especially when the system is reaching its capacity limit and cannot provide interactive responses to all the requests. Quality of service (QoS) support and service differentiation have been studied extensively in network packet switching with respect to packet delay and connection bandwidth [18, 29, 58, 74]. It is equally important to extend network-level QoS support to endpoint systems where service fulfillment and content generation take place. Those issues are especially critical for cluster-based network services in which contents are dynamically generated and aggregated [13, 40, 47, 67].

This dissertation presents the design and implementation of an integrated quality-aware resource management framework for cluster-based services. Although cluster-based network services have been widely deployed, we have seen limited research in the literature on comprehensive resource management with service differentiation support. Recent studies on endpoint resource management and QoS support have been mostly focused on single-host systems [1, 6, 14, 17, 20, 59, 79] or clustered systems serving static HTTP content [10, 65]. In comparison, this dissertation study is intended for clustered services with dynamic service fulfillment or content generation. In particular, it addresses the inadequacy of the previous studies and complements them in the following three aspects. First, it allows quality-aware resource management objectives which combine the individual service response times with the overall system resource utilization efficiency. Secondly, it employs a functionally symmetrical architecture that does not rely on any centralized components for high scalability and availability. Thirdly, it uses an adaptive scheduling policy that achieves efficient resource utilization at a wide range of load conditions.

1.1.3 Service Replication

Replication of persistent data is crucial to load sharing and achieving high availability. Previous work has shown that synchronous replication based on *eager* update propagations does not deliver scalable solutions [8, 44]. Various asynchronous models have been proposed for wide-area or wireless distributed systems [5, 8, 44, 66, 81]. However, these studies have not explicitly address the high scalability/availability demand and potentially weak consistency requirement of large-scale network services. Additional studies are needed to investigate the replica consistency and fail-over support for large-scale cluster-based Internet services.

This dissertation study is built upon a large body of previous research in network service clustering, fault-tolerance, and data replication. The goal of this work is to provide flexible and efficient service replication support for network services with frequently updated persistent data. This model should make it simple to deploy existing applications and shield application programmers from the complexities of replica consistency and fail-over support. It also needs to have the flexibility to accommodate a variety of data management mechanisms that network services typically rely on. Under the above consideration, our system is designed to support multiple levels of replica consistency depending on application characteristics. The objective is to provide the desired level of replica consistency for large-scale Internet services with the emphasis on performance scalability and fail-over support.

1.2 Overview

The rest of this dissertation is organized as follows. Chapter 2 presents the overall Neptune clustering architecture and programming interfaces. Chapter 3 describes cluster load balancing support with the focus on fine-grain services. Chapter 4 discusses an integrated resource management framework for scalable network services. Chapter 5 describes a study on multi-level service replication support. Chapter 6 concludes this dissertation and briefly discusses some potential future work.

Chapter 2

Neptune Clustering Architecture

2.1 Introduction

Large-scale cluster-based network services are increasingly emerging to deliver highly scalable, available, and feature-rich user experiences. Within a large-scale complex service cluster, service components are usually partitioned, replicated, and aggregated. Partitioning is introduced when the service processing requirement or data volume exceeds the capacity of a single server node. Service replication is commonly employed to improve the system availability and provide load sharing. In addition, the service logic itself may be too complicated such that it needs to be partitioned into multiple service components. Partial results may need to be aggregated across multiple data partitions or multiple service modules, and then delivered to external users.

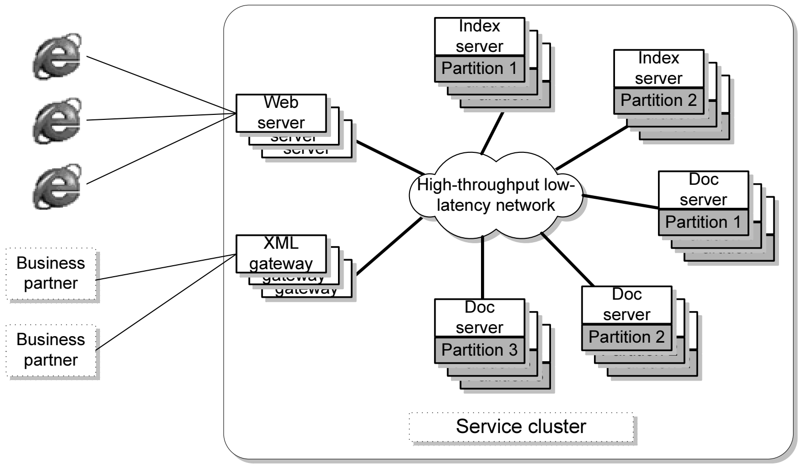

Figure 2.1: Service cluster architecture for a document search engine.

Figure 2.1 illustrates such a clustering architecture for a document search engine [13, 42]. In this example, the service cluster delivers search services to consumers and business partners through Web servers and XML gateways. Inside the cluster, the main search tasks are performed on a set of index servers and document servers, both partitioned and replicated. Each search query first arrives at one of the protocol gateways. Then some index servers are contacted to retrieve the identifications of top ranked Web pages matching the search query. Subsequently some document servers are mobilized to retrieve a short description of these pages and the final results are returned through the original protocol gateway.

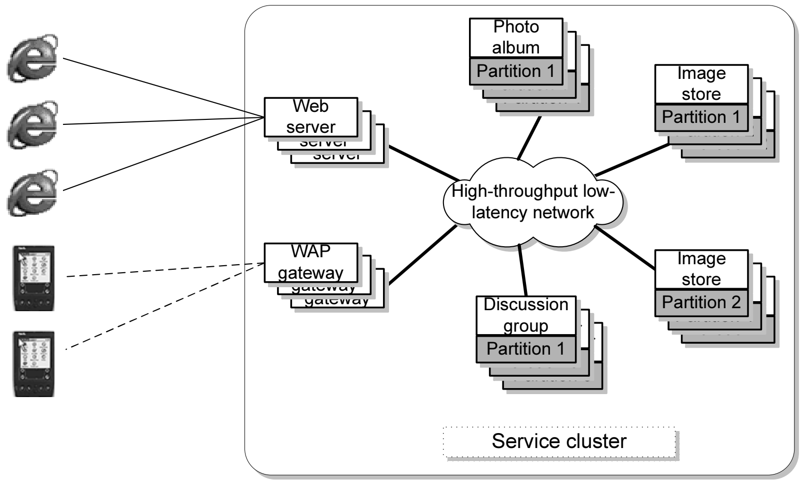

Figure 2.2: Service cluster architecture for a discussion group and photo album service.

As another example, Figure 2.2 illustrates the service cluster architecture for a discussion group and photo album service, similar to MSN Groups [48]. We call this service the *Groups* service. In this case, the service cluster delivers a discussion group and a photo album service to wide-area browsers and wireless clients through Web servers and WAP gateways. The discussion group service is delivered independently while the photo album service relies on an internal image store service. All the components (including protocol gateways) are replicated. In addition, the image store service is partitioned into two partition groups. The Groups service differs from the document search in two ways. First, all service components are directly invoked by the protocol gateways in document search. In the Groups service, however, the service invocation is hierarchical in that the image store is only invoked by the photo album service components. Secondly, external service requests are read-only toward persistent service data for document search while data updates can be triggered by external requests for the Groups service.

Given a service application running on a single machine with a modest amount of persistent data, how can such a service be expanded quickly to run on a cluster environment for handling a large volume of concurrent request traffic with large-scale persistent service data? *Scalability* and *availability* are two main goals that need to be considered. Scalability demands that the increase in hardware resources can maintain a corresponding increase in system throughput and storage capacity. Availability requires the systems to be operational despite transient hardware or software failures. Previous work has recognized the importance of providing software infrastructures for cluster-based network services. For example, the TACC and MultiSpace projects have addressed the fail-over support, component reusability and extensibility for cluster-based services [40, 47]. These systems do not provide explicit support for services with large-scale persistent service data. Recently the DDS project tackles such an issue with a carefully built data management layer that encapsulates scalable replica consistency and fail-over support [46]. While this approach is demonstrated for services with simple processing logic like distributed hash tables, constructing such a data management layer tends to be difficult for applications with complex data management logic, including many database applications. One such example is the previously described discussion group and photo album service.

This dissertation investigates techniques in building a middleware system, called *Neptune*, that provides clustering support for scalable network services. Neptune employs a loosely-connected and functionally-symmetrical approach in constructing the service cluster. This architecture allows Neptune service infrastructure to operate smoothly in the presence of transient failures and through service evolution. In addition, Neptune provides simple programming interfaces that make it easy to deploy existing applications and shield application programmers from clustering complexities. Generally speaking, providing standard system components to achieve scalability and availability tends to decrease the flexibility of service construction. Neptune demonstrates that it is possible to achieve these goals by targeting partitionable network services. The rest of this chapter is organized as follows. Section 2.2 presents the overall Neptune clustering architecture. Section 2.3 describes the programming interfaces. Section 2.4 discusses the system implementation and a number of applications that have been successfully deployed. Section 2.5 examines related work and Section 2.6 concludes this chapter.

2.2 Overall Clustering Architecture

Neptune's design takes advantage of the following characteristics existing in many Internet services: 1) *Information independence*. Network services tend to host a large amount of information addressing different and independent categories. For example, an auction site hosts different categories of items. Every bid only accesses data concerning a single item, thus providing an intuitive way to partition the data. 2) *User independence*. Information accessed by different users tends to be independent. Therefore, data may also be partitioned according to user accounts. Email service and Web page hosting are two examples of this. With these characteristics in mind, Neptune is targeted at *partitionable* network services in the sense that persistent data manipulated by such a service can be divided into a large number of independent partitions and each service access can be delivered independently on a single partition; or each access is an aggregate of a set of sub-accesses each of which can be completed independently on a single partition. With fast growing and wide-spread usage of Internet applications, partitionable network services are increasingly common.

Neptune encapsulates an application-level network service through a service access interface which contains several RPC-like access methods. Each service access through one of these methods can be fulfilled exclusively on one data partition. Neptune employs a *functionally symmetrical* approach in constructing the server cluster. Every node in the cluster contains the same Neptune components and is equipped with the same clustering capabilities. Each cluster node can elect to host service modules with certain data partitions and it can also access service modules hosted at other nodes in the cluster. Within a Neptune cluster, all the nodes are loosely connected through a well-known publish/subscribe channel. This channel can be implemented using IP multicast or through a highly available well-known central directory.

Figure 2.3 illustrates the architecture of a Neptune node. Again, since

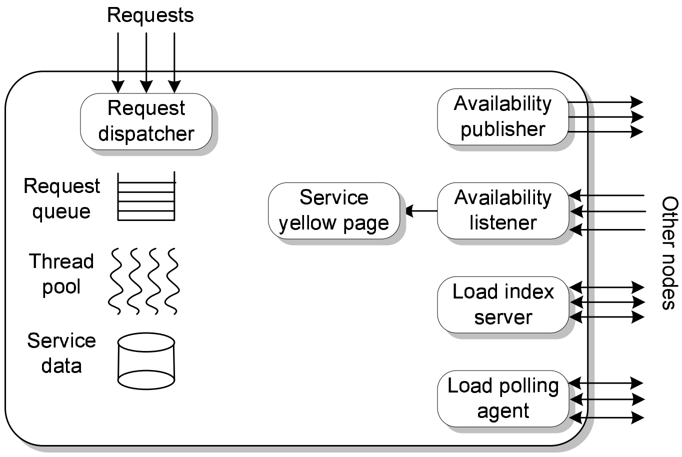

Figure 2.3: Architecture of a Neptune server node.

Neptune service cluster is functionally symmetrical, all the nodes are based on the same architecture. The key components are described as follows. The *request dispatcher* directs an incoming request to the hosted service module which contains a request queue and a pool of worker threads or processes. When all worker threads or processes are busy, subsequent requests will be queued. This scheme allows a Neptune server to gracefully handle spikes in the request volume while maintaining a desired level of concurrency. The *availability publisher* periodically announces the locally hosted service module, the data partitions, and the access interface to other nodes in the cluster. The *availability listener* monitors those announcements from other nodes and maintains a local *service yellow page* containing available service modules and their locations in the cluster. The availability information in the yellow page is kept as soft state such that it has to be refreshed repeatedly to stay alive. This allows node failures and system upgrades to be handled smoothly without complex protocol handshakes. When one node needs to access a service module hosted in the cluster, it first contacts the local service yellow page to acquire the list of nodes in the cluster able to provide the requested service. Then it randomly chooses a certain number of them and polls for their load indexes through a *load polling agent*. The polls are responded by *load index server*s at the polled servers. The requesting node finally forwards the request to the server responding with the lightest load index.

Figure 2.4 illustrates the use of Neptune in application clustering. The request dispatcher, request queue, thread/process pool, availability publisher, and load index server can be considered as server-side components in a service invocation, and they are in aggregate called *Neptune server module*. Similarly, the availability listener, service yellow page, and the load polling agent are together called *Neptune client module*. In the Groups service shown in Figure 2.2, each photo album service instance locates and accesses an image store service instance through the local Neptune client module. In addition, each gateway

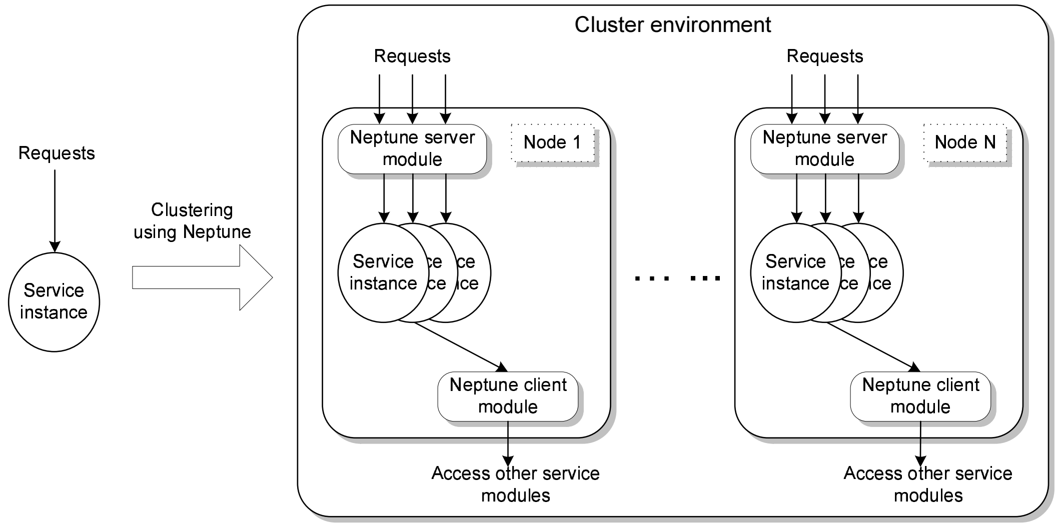

Figure 2.4: Clustering using Neptune.

node relies on the same Neptune client module to export the discussion group and photo album service to external users. Overall, the loosely-connected and functionally-symmetrical architecture allows Neptune service infrastructure to operate smoothly in the presence of transient failures and through service evolution.

In a Neptune-enabled service cluster, the application-level service programmer only needs to implement the stand-alone application modules. The aggregation of multiple data partitions or application modules as well as data replication are supported transparently by Neptune clustering modules. Data replication introduces the complexities of load balancing and consistency enforcement. We will examine cluster load balancing in Chapter 3 and replica consistency support will be the main focus of Chapter 5.

2.3 Neptune Programming Interfaces

Neptune supports two communication schemes between service clients and service instances inside the service cluster: a request/response scheme and a stream-based scheme. In the request/response scheme, the service client and the server instance communicate with each other through a request message and a response message. For the stream-based scheme, Neptune sets up a bidirectional stream between the client and the service instance as a result of the service invocation. Stream-based communication can be used for asynchronous service invocation and it also allows multiple rounds of interaction between the client and the service instance. Currently Neptune only supports stream-based communication for read-only service accesses because of the complication in replicating and logging streams. The rest of this section focuses on the programming interfaces for the request/response communication scheme. The model for stream-based communications can be easily inferred.

For the simplicity of the following discussion, we classify a service access as a *read access* (or *read* in short) if it does not change the persistent service data, or as a *write access* (or *write* in short) otherwise.

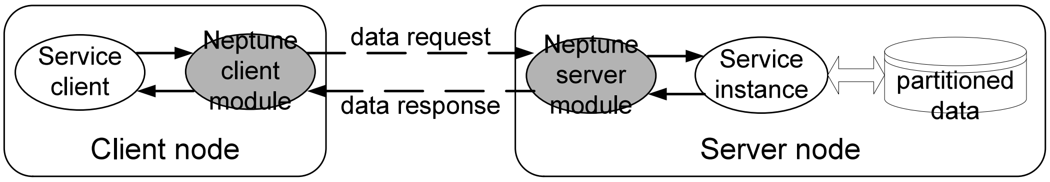

Figure 2.5: Interaction among service modules and Neptune modules during a request/response service invocation.

Figure 2.5 illustrates the interaction among service modules and Neptune modules during a request/response service invocation. Basically, each service access request is made by the client with a service name, a data partition ID, a service method name, and a read/write access mode, as discussed below on the client interface. Then the Neptune client module transparently selects a service node based on the service/partition availability, access mode, consistency requirement, runtime workload, and the load balancing policy. Upon receiving the request, the Neptune server module in the chosen node spawns a service instance to serve the request and return the response message when it completes. Further request propagations may occur in the case of service replication, which will be discussed in detail in Chapter 5.

We discuss below the interfaces between Neptune and service modules at both the client and the server sides for the request/response communication scheme:

- At the client side, Neptune provides a unified interface to service clients for seeking location-transparent request/response service access. It is shown below in a language-neutral format:

  ```
  NeptuneRequest (NeptuneHandle, ServiceName, PartitionID,
            ServiceMethod, AccessMode, RequestMsg, ResponseMsg);
  ```

  A `NeptuneHandle` should be used in every service request that a client invokes. It maintains the information related to each client session. The meanings of other parameters are straightforward.

- At the server side, all the service method implementations need to be registered at the service deployment phase. This allows the Neptune server module to invoke the corresponding service instance when a service request is received.

2.4 System Implementation and Service Deployments

Neptune has been implemented on Linux and Solaris clusters. The publish/subscribe channel is implemented using IP multicast. Each multicast message contains the service announcement and node runtime CPU and I/O workload, acquired through Linux `/proc` file system. Note that this information is only used for monitoring purpose and it is not meant for load balancing. We try to limit the size of each multicast packet to be within an Ethernet maximum transmission unit (`MTU`) in order to minimize the multicast overhead. We let each node send the multicast message once every second and the published information is kept as soft state, expiring in five seconds. That means a faulty node will be detected when five of its multicast messages are not heard in a row. This "soft state"-based node aliveness information can be inaccurate at times, especially for servers that keep going up and down. As a result, service connection setup may fail due to false node aliveness information. Neptune attempts three retries in these cases, after which failing nodes are excluded from local service yellow page. Those failing nodes will be added back when future availability announcements are heard.

Inside each service node, the service instance could be compiled into a dynamically linked library. They are linked into Neptune process space at runtime and run as threads. Note that different service modules can be compiled into different library binaries to simplify partial service upgrade. Alternatively, each service instance could run as a separate OS process, which would provide better fault isolation and resource control at the cost of degraded performance. Running service instances as separate processes also allow existing application binaries being deployed in a Neptune cluster without the need of recompilation. In choosing between thread and process-based deployment, the rule of thumb is to pick threads for simple and short-running services while using processes for complex and large service modules.

Cross-platform compatibility is a major goal of this implementation. All inter-node protocol messages are compatible at the binary level. All multibyte numbers are ordered to the big-endian before sending out and they are transformed to the host order after being received. Neptune has been deployed in service clusters containing nodes of both Linux/Intel and Solaris/SPARC architectures. This experience demonstrates Neptune's cross-platform compatibility.

Overall, this implementation incurs moderate overhead. For instance, we implemented an echo service on top of a Neptune-enable service cluster. The echo service simply responds to the client with a message identical to the request message. The cluster is comprised of nodes with dual 400 Mhz Pentium II processors running Linux 2.2.15. All the nodes are connected with a 100 Mb/s switched Ethernet. The response time of such an echo service is measured at 1128 us excluding the polling overhead incurred in service load balancing. Excluding the TCP roundtrip time with connection setup and teardown which is measured at 1031 us, Neptune is responsible for an overhead of 97 us in each service invocation.

Service Deployments

So far we have deployed a document search engine, a scientific data mining application, and three demonstration services on Neptune clusters, described as follows.

- *Document search.* This service takes in a group of encoded query words; checks a memory mapped index database; and returns the identifications of the list of documents matching the input query words. An earlier version of Neptune has been deployed in Ask Jeeves search (`www.ask.com`) for Web document searching. This Neptune-enabled Solaris cluster hosts an index of about 200 million Web documents and is capable of serving over 30 million requests a day (the actual daily traffic varies).

- *Nucleotide and protein sequence similarity matching.* We have also deployed the nucleotide and protein sequence similarity matching application *BLAST* [7] from National Center for Biotechnology Information on a Linux cluster. Unlike other deployed services that service instances run as threads to gain efficiency, service instances in this case run as OS processes such that the original application binaries can be used directly without the need of recompilation. Currently we have maintained a nucleotide and protein sequence database with around 6.4 GB data, acquired from GenBank [16].

- *Online discussion group*, *auction* and *persistent cache.* These three demonstration services are developed on a Linux cluster. Here we describe the discussion group service in more detail. Other services will be explained when they are used in later chapters. The discussion group handles three types of requests for each discussion topic: viewing the list of message headers (`ViewHeaders`), viewing the content of a message (`ViewMsg`), and adding a new message (`AddMsg`). Both `ViewHeaders` and `ViewMsg` are read-only requests. The messages are maintained and displayed in a hierarchical format according to the reply-to relationships among them. The discussion group relies on underlying MySQL databases to store and retrieve messages and discussion topics.

2.5 Related Work

Software infrastructure for clustered services. Previous work has addressed providing clustering support for network service transformation, aggregation, and composition [9, 40, 84]. For instance, the TACC project employs a two-tier architecture and the service components called "workers" run on different back-ends while accesses to workers are controlled by front-ends which dispatch incoming requests to back-ends [40]. Most of these studies propose two-tier clustering architecture and rely on central components to maintain the server runtime workload information. Neptune employs a loosely-connected and functionally-symmetrical architecture that allows complex multi-tier services to be easily supported. In addition, this architecture does not contain any centralized components so that component failures do not cause system-wide catastrophe.

Fail-over and service migration support. Providing service fail-over and migration support for off-the-shelf applications is addressed in the SunSCALR project [72] and the Microsoft cluster service (MSCS) [78]. SunSCLAR relies on IP fail-over to provide failure detection and automatic reconfiguration. MSCS offers an execution environment where existing server applications can operate reliably on an NT cluster. These studies take advantage of the low-level network protocol standard or operating system features to provide maximum transparency to application services in achieving service reliability. In comparison, Neptune provides a controlled runtime execution environment for application services which allows complex cluster load balancing, resource management, and service replication support to be efficiently incorporated into one framework.

Component-based programming architecture. Component-based inter-operable programming architectures are supported in standards like CORBA [26], COM/DCOM [25], and Java [54]. For example, CORBA provides standard interfaces for service component interactions and it supports basic functionalities including naming service, load balancing, and persistent state services. While sharing the same goal of supporting inter-operable service components, this dissertation study is focused on supporting highly concurrent services in tightly-coupled cluster environments and addressing the scalability and availability issues in such a context.

2.6 Concluding Remarks

In this chapter, we present the overall Neptune clustering architecture, its programming interfaces, and a number of applications that have been successfully deployed in Neptune-based service clusters. It should be noted that providing clustering support for scalable network applications has been extensively studied in recent years. Neptune distinguishes itself mainly in two aspects. First, it employs a loosely-connected and functionally-symmetrical architecture in constructing the service cluster, which allows the service infrastructure to operate smoothly in the presence of transient failures and through service evolution. Secondly, Neptune provides flexible interfaces that allow existing applications to be easily deployed, even for binary applications without recompilation [7]. Overall, the clustering architecture presented in this chapter serves as the basis for the studies on cluster load balancing, resource management, and service replication support described in later parts of this dissertation.

Chapter 3

Cluster Load Balancing

3.1 Introduction

While previous research has addressed the issues of providing clustering support for network service transformation, aggregation, and composition [9, 40, 47, 77], there is still a lack of comprehensive study on load balancing support in this context. This chapter investigates the techniques of providing efficient load balancing support for accessing replicated services inside the service cluster.

A large amount of work has been done by the industry and research community to optimize HTTP request distribution among a cluster of Web servers [11, 12, 21, 38, 53, 64]. Most load balancing policies proposed in such a context rely on the premise that all network packets go through a single front-end dispatcher or a TCP-aware (layer 4 or above) switch so that TCP level connection-based statistics can be accurately maintained. In contrast, clients and servers inside the service cluster are often connected by high-throughput, low-latency Ethernet (layer 2) or IP (layer 3) switches, which do not provide any TCP level traffic statistics. This constraint calls for more complex load information dissemination schemes.

Previous research has proposed and evaluated various load balancing policies for cluster-based distributed systems [9, 15, 28, 32, 43, 57, 61, 82, 83]. Load balancing techniques in these studies are valuable in general, but not all of them can be applied for cluster-based network services. This is because they focus on coarse-grain computation and often ignore fine-grain jobs by simply processing them locally. For example, the job trace used in a previous trace-driven simulation study has a mean job execution time of 1.492 seconds [83]. In the context of network services, with the trend toward delivering more feature-rich services in real time, large number of fine-grain sub-services need to be aggregated within a short period of time. For example, a distributed hash table lookup for keyword search usually only takes a couple of milliseconds.

Fine-grain services introduce additional challenges because server workload can fluctuate rapidly for those services. This dissertation investigates cluster load balancing support for scalable network services with the emphasis on fine-grain services. The study is based on simulations as well as experiments with an implementation on a Linux cluster.

Almost all load balancing policies can be characterized as being either centralized or distributed. The distinction is whether the policy relies on any centralized components to maintain system-wide workload information and/or make load balancing decisions. Centralized approaches tend to be less scalable and available because any central component can become single point of failure or performance bottleneck. While acknowledging that those issues can be addressed to a certain degree through hot backups and coordination of a set of load balancers [2], we choose to focus on fully distributed load balancing policies in this study.

The rest of this chapter is organized as follows. Section 3.1.1 describes the service traces and synthetic workloads that are used in this study. Section 3.2 presents the simulation studies on load balancing policies and the impact of various parameters. Section 3.3 describes the system implementation on a Linux cluster with a proposed optimization. Section 3.4 evaluates the system performance based on this implementation. Section 3.5 discusses related work and Section 3.6 concludes this chapter.

3.1.1 Evaluation Workload

Workload	Arrival interval		Service time	
	Mean	Std-dev	Mean	Std-dev
Medium-Grain trace	341.5ms	321.1ms	208.9ms	62.9ms
Fine-Grain trace	436.7ms	349.4ms	22.2ms	10.0ms

Table 3.1: Statistics of evaluation traces. Medium-Grain trace contains 1,055,359 accesses with 126,518 of them in the peak portion. Fine-Grain trace contains 1,171,838 accesses with 98,933 of them in the peak portion.

We collected the traces of two internal service cluster components from search engine *Teoma* [76] and their statistics are listed in Table 3.1. Both traces were collected across an one-week time span in late July 2001. One of the services provides the translation between query words and their internal representations. The other service supports a similar translation between Web page descriptions and their internal representations. Both services support multiple translations in one access. The first trace has a mean service time of 22.2 ms and we call it the Fine-Grain trace. The second trace has a mean service time of 208.9 ms and we call it the Medium-Grain trace. We use a peak time portion (early afternoon hours of three consecutive weekdays) from each trace in our study. Most system resources are well under-utilized during non-peak times, therefore load balancing is less critical during those times. Note that the arrival intervals of those two traces may be scaled when necessary to generate workloads at various demand levels during our evaluation.

In addition to the traces, we also include a synthetic workload with Poisson process arrivals and exponentially distributed service times. We call this workload Poisson/Exp in the rest of this chapter. Several previous studies on Internet connections and workstation clusters suggested that both the HTTP inter-arrival time distribution and the service time distribution exhibit high variance, thus are better modeled by Lognormal, Weibull, or Pareto distributions [35, 49]. We choose Poisson/Exp workload in our study for the following reasons. First, a primary cause for the high variance of HTTP arrival intervals is the proximity of the HTTP request for the main page and subsequent requests for embedded objects or images. However, if we only consider resource-intensive service requests which require dynamic content generation, HTTP requests for embedded objects are not counted. Secondly, the service time distribution tends to have a low variance for services of the same type. Our analysis on the Teoma traces show that those distributions have similar variances as an exponentially distributed sample would have. This observation is further confirmed by larger traces we acquired later at Ask Jeeves search.

3.2 Simulation Studies

In this section, we present the results of our simulation studies. As mentioned before, we confine our study to fully distributed load balancing policies that do not contain any single point of failure. We will first examine the load information inaccuracy caused by its dissemination delay. This delay is generally insignificant for coarse-grain jobs but it can be critical for fine-grain services. We will then move on to study two distributed load balancing policies: 1) the *broadcast* policy in which load information is propagated through server-initiated pushing; and 2) the *random polling* policy in which load information is propagated through client-initiated pulling. We choose them because they represent two broad categories of policies in terms of how load information is propagated from the servers to the clients. In addition, they are both shown to be competitive in a previous trace-driven simulation study [83].

In our simulation model, each server contains a non-preemptive processing unit and a FIFO service queue. The network latency of sending a service request and receiving a service response is set to be half a TCP roundtrip latency with connection setup and teardown, which is measured at 516 us on a switched 100 Mb/s Linux cluster. Our model does not consider the impact of memory locality on the service processing time. For cluster-based data-intensive applications, the service data is typically partitioned such that the critical working set [13] or the whole service data [42] can fit into the system memory. This study on load balancing is geared toward replicated service nodes of each data partition.

We choose the mean service response time as the *performance index* to measure and compare the effectiveness of various policies. We believe this is a better choice than system throughput for evaluating load balancing policies because system throughput is tightly related to the admission control, which is beyond the scope of this study.

3.2.1 Accuracy of Load Information

Almost all load balancing policies use some sort of *load index*es to measure server load levels. Prior studies have suggested a linear combination of the resource queue lengths can be an excellent predictor of service response time [36, 82]. We use the total number of active service accesses, i.e. the queue length, on each server as the server load index. In most distributed policies, load indexes are typically propagated from server side to client side in some way and then each client uses acquired information to direct service accesses to lightly loaded servers. Accuracy of the load index is crucial for clients to make effective load balancing decision. However, the load index tends to be stale due to the delay between the moment it is being measured at the server and the moment it is being used at the client. We define the *load index inaccuracy* for a certain delay Δt as the statistical mean of the queue length difference measured at arbitrary time t and $t + \Delta t$. Figure 3.1 illustrates the impact of this delay (normalized to mean service time) on the load index inaccuracy for a single server through simulations on all three workloads. We also show the upperbound for Poisson/Exp in a straight line. With the assumption that the inaccuracy monotonically increases with the increase of Δt, the upperbound is the statistical mean of the queue length difference measured at any two arbitrary time t_1 and t_2. Let term ρ be defined as the mean service time divided by the mean arrival interval, which reflects the level of server load. For a Poisson/Exp workload, since the limiting probability that a single server system has a queue length of k is $(1-\rho)\rho^k$ [56], the upperbound can be calculated as:

$$\sum_{i,j=0}^{\infty}(1-\rho)^2\rho^{i+j}|i-j| = \frac{2\rho}{1-\rho^2} \tag{3.2.1}$$

Calculation: We start with defining

$$UB = \sum_{i,j=0}^{\infty} (1-\rho)^2 \rho^{i+j} |i-j| \tag{3.2.2}$$

Let $n = i + j$, then we have

$$UB = (1-\rho)^2 \sum_{n=0}^{\infty} \rho^n \sum_{i=0}^{n} |n-2i| \tag{3.2.3}$$

We define $F(n) = \sum_{i=0}^{n} |n-2i|$. When n is an even number such that $n = 2k$, we have

$$\begin{aligned} F(n) &= \sum_{i=0}^{2k} |2k-2i| \\ &= 2\sum_{i=0}^{k-1} (2k-2i) = 2k(k+1) \end{aligned} \tag{3.2.4}$$

When n is an odd number such that $n = 2k+1$, we have

$$\begin{aligned} F(n) &= \sum_{i=0}^{2k+1} |2k+1-2i| \\ &= 2\sum_{i=0}^{k} (2k+1-2i) = 2(k+1)^2 \end{aligned} \tag{3.2.5}$$

Merge (3.2.4) and (3.2.5) into (3.2.3), we have

$$\begin{aligned} UB &= (1-\rho)^2 \sum_{k=0}^{\infty} (\rho^{2k} 2k(k+1) + \rho^{2k+1} 2(k+1)^2) \\ &= 2(1-\rho)^2 \sum_{k=0}^{\infty} ((1+\rho)k^2\rho^{2k} + (1+2\rho)k\rho^{2k} + \rho^{2k+1}) \end{aligned} \tag{3.2.6}$$

Since we know

$$\sum_{k=0}^{\infty} \rho^{2k} = \frac{1}{1-\rho^2}; \tag{3.2.7}$$

$$\sum_{k=0}^{\infty} k\rho^{2k} = \frac{\rho^2}{(1-\rho^2)^2}; \tag{3.2.8}$$

$$\text{and} \sum_{k=0}^{\infty} k^2\rho^{2k} = \frac{\rho^2(1+\rho^2)}{(1-\rho^2)^3}. \tag{3.2.9}$$

Merge them all into (3.2.6), we have

$$UB = \frac{2\rho}{1-\rho^2} \tag{3.2.10}$$

■

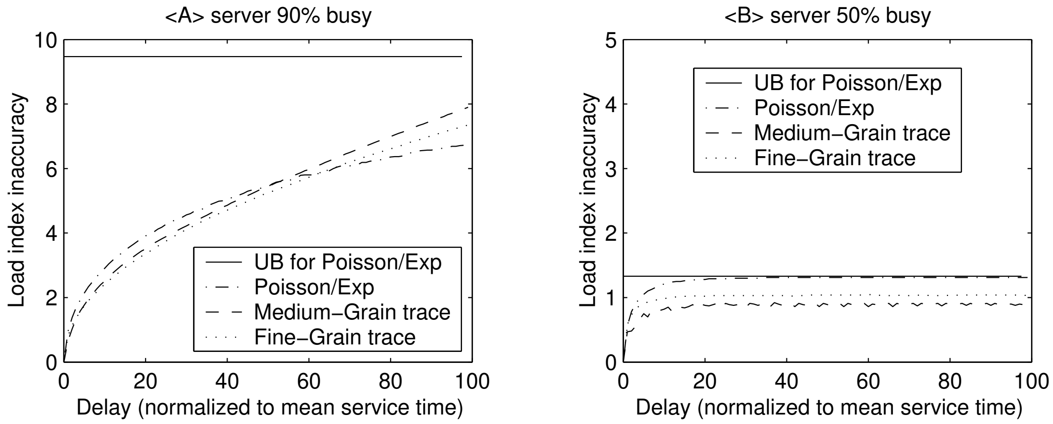

Figure 3.1: Impact of delay on load index inaccuracy with 1 server (simulation).

We can make the following observations from the results in Figure 3.1. When the server is moderately busy (50%), the load index inaccuracy quickly reaches the upperbound (1.33 for Poisson/Exp) when delay increases, but the inaccuracy is moderate even under high delay. This means a random approach is likely to work well when servers are only moderately busy and fancier policies do not improve much. When the server is very busy (90%), the load index inaccuracy is much more significant and it can cause an error of around 3 in the load index when the delay is around 10 times the mean service time. This analysis reveals that when servers are busy, fine-grain services require small dissemination delays in order to have accurate load information on the client side.

3.2.2 Broadcast Policy

In the broadcast policy, an agent is deployed at each server which collects the server load information and announces it through a broadcast channel at various intervals. It is important to have non-fixed broadcast intervals to avoid the system self-synchronization [37]. The intervals we use are evenly distributed between 0.5 and 1.5 times the mean value. Each client listens at this broadcast channel and maintains the server load information locally. Then every service request is made to a server with the lightest workload. Since the server load information maintained at the client side is acquired through periodical server broadcasts, this information becomes stale between consecutive broadcasts and the staleness is in large part determined by the broadcast frequency. Figure 3.2 illustrates the impact of broadcast frequency through simulations. A 50 ms mean service time is used for Poisson/Exp workload. Sixteen servers are used in the simulation. The mean response time shown in Figure 3.2 is normalized to the mean response time under an *ideal* approach, in which all server load indexes can be accurately acquired on the client side free-of-cost whenever a service request is to be made.

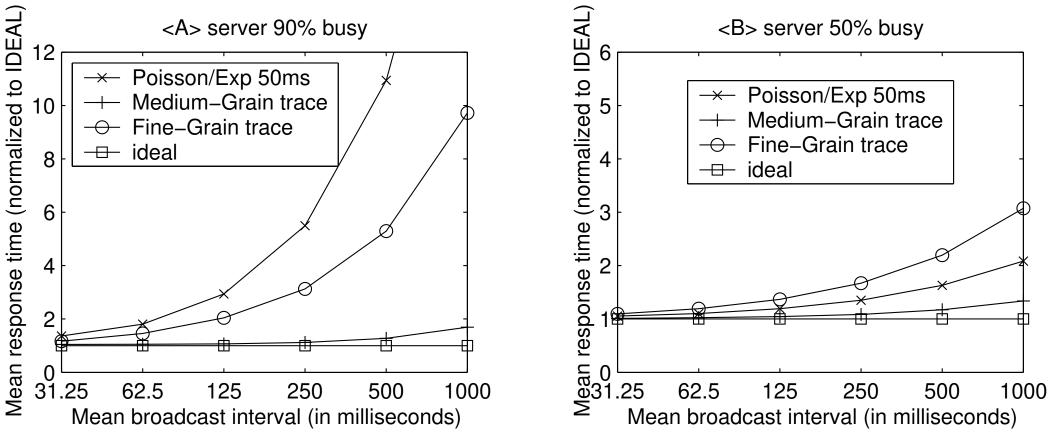

Figure 3.2: Impact of broadcast frequency with 16 servers (simulation).

When servers are 90% busy, we observe that the performance for broadcast policy with 1 second mean broadcast interval could be an order of magnitude slower than the ideal scenario for fine-grain services (Poisson/Exp and Fine-

Grain trace). The degradation is less severe (up to 3 times) when servers are 50% busy, but it is still significant. This problem is originally caused by the staleness of load information due to low broadcast frequency. The staleness is further severely aggravated by the *flocking* effect of the broadcast policy, i.e. all service requests tend to flock to a single server (the one with the lowest perceived queue length) between consecutive broadcasts. The performance under low broadcast interval, e.g. 31.25 ms, is close to the ideal scenario. However, we believe the overhead will be prohibitive under such high frequency, e.g. a sixteen server system with 31.25 ms mean broadcast interval will make each client to process a broadcast message every 2 ms.

3.2.3 Random Polling Policy

For every service access, the random polling policy requires a client to randomly poll several servers for load information and then direct the service access to the most lightly loaded server according to the polling results. An important parameter for a random polling policy is the poll size. Mitzenmacher demonstrated through analytical models that a poll size of two leads to an exponential improvement over pure random policy, but a poll size larger than two leads to much less additional improvement [61]. Figure 3.3 illustrates our simulation results on the impact of poll size using all three workloads. Policies with the poll size of 2, 3, 4, and 8 are compared with the random and ideal approach in a sixteen server system. A 50 ms mean service time is used for Poisson/Exp workload.

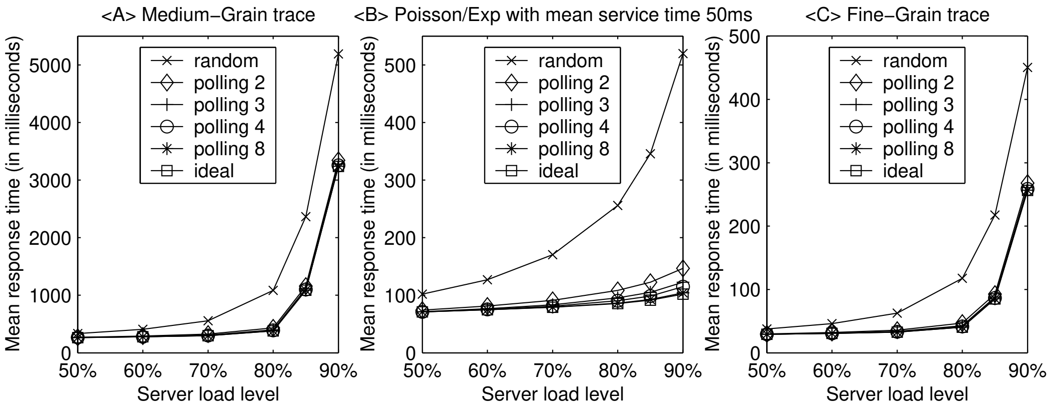

Figure 3.3: Impact of poll size with 16 servers (simulation).

This result echoes Mitzenmacher's analytical results in the sense that a poll size of two leads to exponential improvement over the pure random policy while a larger poll size does not provide much additional benefit [61]. Our simulation also suggests that this result is consistent across all service granularity and all server load level, which makes it a very robust policy. Quantitatively, the mean response time under the random polling policy with a poll size of three is within 20% from the performance of the ideal case for all studied workloads under 90% server load level. We believe the random polling policy is well-suited for fine-grain services because the just-in-time polling always guarantees very little staleness on the load information.

3.2.4 Summary of Simulation Studies

First, our simulation study shows that a long delay between the load index measurement time at the server and its time of usage at the client can yield significant inaccuracy. This load index inaccuracy tends to be more severe for finer-grain services and busier servers. Then we go on to study two representative policies, broadcast and random polling. Our results show that random polling based load balancing policies deliver competitive performance across all service granularities and all server load levels. In particular, the performance of the random polling policy with a poll size of three is within 20% from the performance of the ideal scenario under 90% server load level. As for the broadcast policy, we identify the difficulty of choosing a proper broadcast frequency for fine-grain services. A low broadcast frequency results in severe load index inaccuracy, and in turn degrades the system performance significantly. A high broadcast frequency, on the other hand, introduces high broadcast overhead. Ideally, the broadcast frequency should linearly scale with the system load level to cope with rapid system state fluctuation. This creates a scalability problem because the number of messages under a broadcast policy would linearly scale with three factors: 1) the system load level; 2) the number of servers; and 3) the number of clients. In contrast, the number of messages under the random polling policy only scale with the server load level and the number of servers.

3.3 System Design and Implementation

The major limitation of simulation studies is that it cannot accurately capture various system overhead. For example, a TCP roundtrip with connection setup and teardown costs around 1 ms on our Linux cluster; and a context switch can cost anywhere between tens of microseconds to tens of milliseconds depending on the size of memory footprint. Those costs could be very significant for a 20 ms service access. In recognizing this limitation, We have developed an implementation of the random polling policy on top of a cluster-based service infrastructure. The simulation results in Section 3.2 favor random polling policy so strongly that we do not consider the broadcast policy in this implementation.

3.3.1 System Architecture

This implementation is a continuation of the Neptune clustering architecture. Neptune allows services ranging from read-only to frequently updated be replicated and aggregated in a cluster environment. Neptune encapsulates an application-level network service through a service access interface which contains several RPC-like access methods. Each service access through one of these methods can be fulfilled exclusively on one data partition. Neptune employs a functionally symmetrical architecture in constructing the service network infrastructure. A node can elect to provide services and it can also access services provided by other nodes. It serves as an internal *server* or *client* in each context respectively.

Conceptually, for each service access, the client first acquires the set of available server nodes able to provide the requested service through a *service availability subsystem*. Then it chooses one node from the available set through a *load balancing subsystem* before sending the service request. The service availability subsystem is maintained around a well-known publish/subscribe channel, which can be implemented using IP multicast or a highly available well-known central directory. Each cluster node can elect to provide services through repeatedly publishing the service type, the data partitions it hosts, and the access interface. Each client node subscribes to the well-known channel and maintains a service yellow page.

We implemented a random polling policy for the load balancing subsystem. For each service access, the client randomly chooses a certain number of servers out of the available set returned from the service availability subsystem. Then it sends out load inquiry requests to those servers through connected UDP sockets and asynchronously collects the responses using `select` system call. Finally the client directs the request to the service node responding with the lightest load index. Figure 3.4 illustrates the client/server architecture in Neptune service infrastructure.

3.3.2 Discarding Slow-responding Polls

On top of the basic random polling implementation, we also made an enhancement by discarding slow-responding polls. Through a ping-pong test on two idle machines within our Linux cluster, we measured that a UDP roundtrip cost is around 290 us. However, it may take much longer than that for a busy server to respond a UDP request. We profiled a typical run under a poll size of 3, a server load index of 90%, and 16 server nodes. The profiling shows that 8.1% of the polls are not completed within 10 ms and 5.6% of them are not

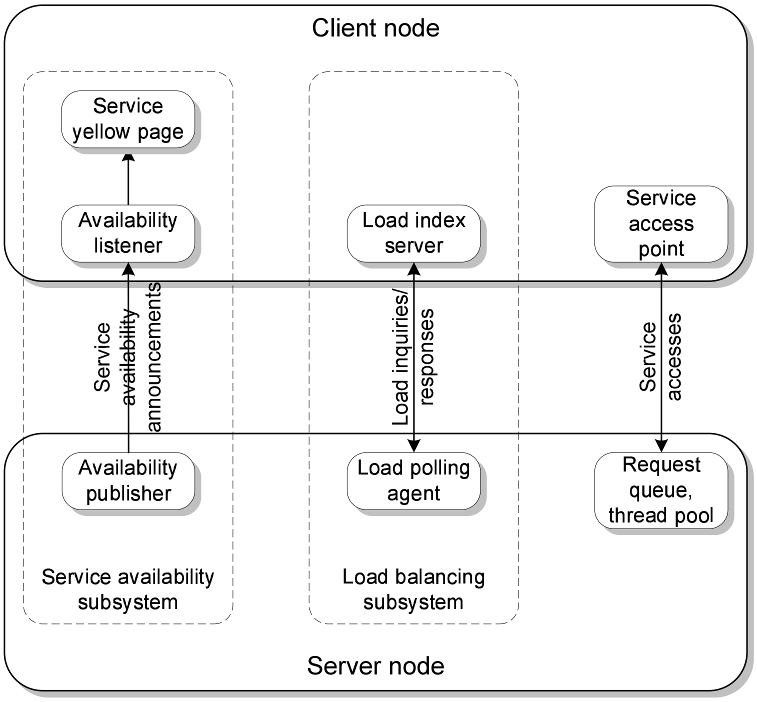

Figure 3.4: The client/server architecture in Neptune service infrastructure.

completed within 20 ms. With this in mind, we enhanced the basic polling policy by discarding polls not responded within 10 ms. Intuitively, this results in a tradeoff between consuming less polling time and acquiring more load information. However, we also realize that long polls result in inaccurate load information due to long delay. Discarding those long polls can avoid using stale load information, which is an additional advantage. And this tends to be more substantial for fine-grain services.

3.4 Experimental Evaluations

All the evaluations in this section were conducted on a rack-mounted Linux cluster with around 30 dual 400 Mhz Pentium II nodes, each of which contains either 512 MB or 1 GB memory. Each node runs Linux 2.2.15 and has two 100 Mb/s Ethernet interfaces. The cluster is connected by a Lucent P550 Ethernet switch with 22 Gb/s backplane bandwidth. All the experiments presented in this section use 16 server nodes and up to 6 client nodes.

Due to various system overhead, we realize that the server load level cannot simply be the mean service time divided by the mean arrival interval. For each workload on a single-server setting, we consider the server reach full load (100%) when around 98% of client requests were successfully completed within two seconds. Then we use this as the basis to calculate the client request rate for various server load levels. The service processing on the server side is emulated using a CPU-spinning micro-benchmark that consumes the same amount of CPU time as the intended service time. The ideal scenario in our simulation study is achieved when all server load indexes can be accurately acquired on the client side free-of-cost whenever a service request is to be made. For the purpose of comparison, we emulate a corresponding ideal scenario in the evaluations. This is achieved through a centralized load index manager which keeps track of all server load indexes. Each client contacts the load index manager whenever a service access is to be made. The load index manager returns the server with the shortest service queue and increments that queue length by one. Upon finishing one service access, each client is required to contact the load index manager again so that the corresponding server queue length can be properly decremented. This approach closely emulates the actual ideal scenario with a delay of around one TCP roundtrip without connection setup and teardown (around 339 us on our Linux cluster).

3.4.1 Evaluation on Poll Size

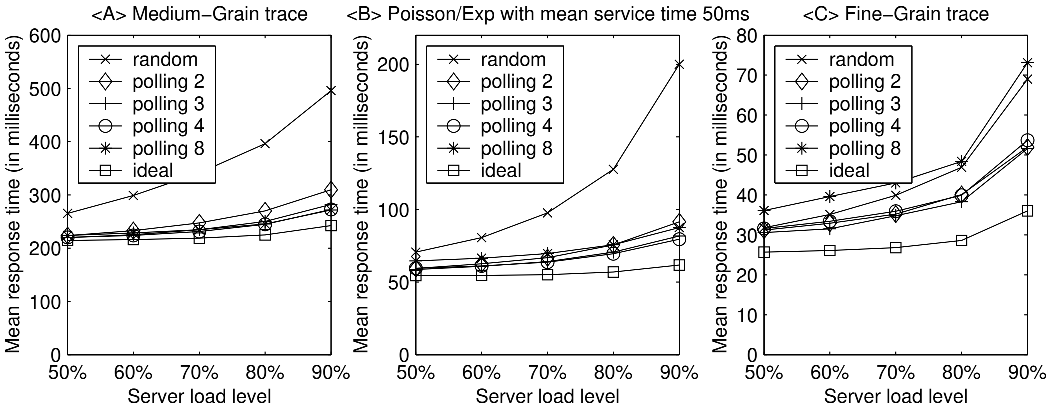

Figure 3.5: Impact of poll size based on experiments with 16 servers.

Figure 3.5 shows our experimental results on the impact of poll size using all three workloads. We observe that the results for Medium-Grain trace and Poisson/Exp workload largely confirm the simulation results in Section 3.2. However, for the Fine-Grain trace with very fine-grain service accesses, we notice that a poll size of 8 exhibits far worse performance than policies with smaller poll sizes and it is even slightly worse than the pure random policy. This is caused by excessive polling overhead coming from two sources: 1) longer polling delays resulting from larger poll size; 2) less accurate server load index due to longer polling delay. And those overheads are more severe for fine-grain services. Our conclusion is that a small poll size (e.g. 2 or 3) provides sufficient information for load balancing. And an excessively large poll size may even degrade the performance due to polling overhead, especially for fine-grain services.

3.4.2 Improvement of Discarding Slow-responding Polls

Table 3.2 shows the overall improvement and the improvement excluding polling time for discarding slow-responding polls. The experiments are conducted with a poll size of 3 and a server load index of 90%. The experiment on Medium-Grain trace shows a slight performance degradation due to the

Workload	Mean response time (mean polling time) Original	Optimized
Medium-Grain trace	282.1ms (2.6ms)	283.1ms (1.0ms)
Poisson/Exp	81.8ms (2.7ms)	79.2ms (1.1ms)
Fine-Grain trace	51.6ms (2.7ms)	47.3ms (1.1ms)

Table 3.2: Performance improvement of discarding slow-responding polls with poll size 3 and server 90% busy.

loss of load information. However, the results on both Fine-Grain trace and Poisson/Exp workload exhibit sizable improvement in addition to the reduction of polling time and this additional improvement is a result of avoiding the use of stale load information. Overall, the enhancement of discarding slow-responding polls can improve the load balancing performance by 8.3% for Fine-Grain trace. The improvement is 5.2% if the polling time is excluded. Note that the performance results shown in Figure 3.5 are not with discarding slow-responding polls.

3.5 Related Work

HTTP load balancing. A large body of work has been done to optimize HTTP request distribution among a cluster of Web servers [11, 12, 21, 38, 53, 64]. Most load balancing policies proposed in such a context rely on the premise that all network packets go through a single front-end dispatcher or a TCP-aware (layer 4 or above) switch so that TCP level connection-based statistics can be accurately maintained. However, clients and servers inside the service cluster are often connected by high-throughput, low-latency Ethernet (layer 2) or IP (layer 3) switches, which do not provide any TCP level traffic statistics. This dissertation study shows that an optimized random polling policy that does not require centralized statistics can deliver competitive performance based on an implementation on a Linux cluster.

Load balancing for distributed systems. Previous research has proposed and evaluated various load balancing policies for cluster-based distributed systems [15, 28, 32, 43, 57, 61, 82, 83]. Those studies mostly deal with coarse-grain distributed computation and often ignore fine-grain jobs by simply processing them locally. We put our focus on fine-grain network services by examining the sensitivity of the load information dissemination delay and its overhead. Both are minor issues for coarse-grain jobs but they are critical for fine-grain services.

This study has focused on load distribution policies initiated from service clients. Server-initiated load balancing and work stealing among servers have been studied in previous research [19, 31, 83]. In particular, Eager et al.

show that client-initiated policies perform better than server-initiated policies under light to moderate system loads. The performance comparison under high system loads depends on the job transfer cost, with high transfer cost favoring client-initiated policies [31]. In a different context, work stealing has been successfully employed in multi-threaded runtime systems like Cilk [19] due to the low job transfer cost in shared memory multi-processors.

Data locality. This study on cluster load balancing does not consider the impact of memory locality on the service processing time. The service data for cluster-based data-intensive applications is typically partitioned such that the critical working set [13] or the whole service data [42] can fit into the system memory. This study on load balancing is geared toward replicated service nodes for each data partition. On the other hand, for systems without such a careful partitioning, locality-based request distribution often results in greater performance impact than load balancing. The exact impact of data locality in these cases tends to be application-specific and a previous study has addressed this issue for cluster-based HTTP servers [64].

Impact of high-performance networks. Several recent studies show that network servers based on Virtual Interface (VI) Architecture provide significant performance benefits over standard server networking interfaces [21, 68]. Generally speaking, the advance in network performance improves the effectiveness of all load balancing policies. In particular, such an advance has certain impact on our results. First, a high-performance network layer may allow efficient and high frequency server broadcasts, which improves the feasibility of the broadcast policy. However, the flocking effect and the scalability issue we raised in Section 3.2 remain to be solved. Secondly, a reduction in network overhead might change some quantitative results of our experimental evaluations. For instance, the overhead of the random polling policy with a large poll size might not be as severe as those shown in our experiments. Those issues should be addressed when advanced network standards become more widespread.

3.6 Concluding Remarks

In this chapter, we study load balancing policies for cluster-based network services with the emphases on fine-grain services. Our evaluation is based on a synthetic workload and two traces we acquired from an online search engine. In addition to simulations, we also conducted experimental evaluations with an implementation on a Linux cluster. Our study and evaluations identify techniques that are effective for fine-grain services and lead us to make several conclusions. First, this study shows that the broadcast policies are ill-suited for fine-grain services while random polling based load-balancing policies perform much better. In terms of random polling policy, our simulation study echoes the previous analytical result that a poll size of two leads to exponential improvement over the pure random policy, but a poll size of larger than two leads to much less additional improvement [61]. Quantitatively, our evaluation shows that the mean response time under the random polling policy with a poll size of three is within 20% from the performance of the ideal case under 90% server load level. In addition, our experiments demonstrate that an excessively large poll size can even degrade the performance substantially due to polling overhead. Finally, this study shows that an optimization of discarding slow-responding polls can further improve the performance by up to 8.3%.

Chapter 4

Quality-Aware Resource Management

4.1 Introduction

Previous studies show that the client request rates for Internet services tend to be bursty and fluctuate dramatically from time to time [13, 23, 27]. For instance, the daily peak-to-average load ratio at Internet search service Ask Jeeves (`www.ask.com`) is typically 3:1 and it can be much higher and unpredictable when the system experiences astronomical growth or in the presence of extraordinary events. As another vivid example, the online site of Encyclopedia Britannica (www.britannica.com) was taken offline 24 hours after its initial launch in 1999 due to a site overload. Over-provisioning system resources for a service site to accommodate the potential peak will not be cost-effective. As a consequence, it is desirable to achieve efficient resource utilization for those services under a wide range of load conditions.

Network clients typically seek services interactively and maintaining reasonable response times is imperative. In addition, providing differentiated service qualities and resource allocation to multiple service classes can also be desirable at times, especially when the system is reaching its capacity limit and cannot provide interactive responses to all the requests. Quality of service (QoS) support and service differentiation have been studied extensively in network packet switching with respect to packet delay and connection bandwidth [18, 29, 58, 74]. It is equally important to extend network-level QoS support to endpoint systems where service fulfillment and content generation take place. Those issues are especially critical for cluster-based network services in which contents are dynamically generated and aggregated [13, 40, 47, 67].

This chapter presents the design and implementation of an integrated resource management framework for cluster-based services. This framework is part of the Neptune system: a cluster-based software infrastructure for aggre-

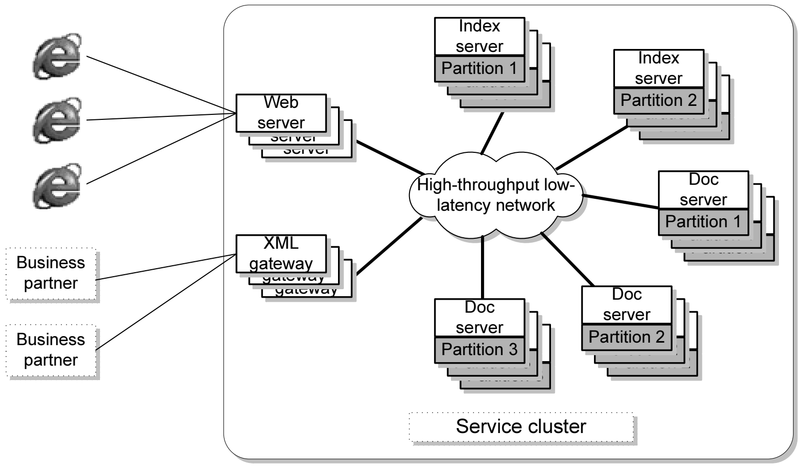

Figure 4.1: Service cluster architecture for a document search engine.

gating and replicating partitionable network services. Figure 4.1 illustrates the clustering architecture for a document search engine [13, 42] which has been described in Chapter 2. In this example, the service cluster delivers search services to consumers and business partners through Web servers and XML gateways. Inside the cluster, the main search tasks are performed on a set of index servers and document servers, both partitioned and replicated. Typically, the service data for cluster-based data-intensive applications is carefully partitioned such that the critical working set [13] or the whole service data [42] can fit into the system memory. This study of resource management is geared toward replicated service nodes for each such data partition, e.g. the replicas for partition 1 of the index servers in the document search engine.

Although cluster-based network services have been widely deployed, we have seen limited research in the literature on comprehensive resource management with service differentiation support. Recent studies on endpoint resource management and QoS support have been mostly focused on single-host systems [1, 6, 14, 17, 20, 59, 79] or clustered systems serving static HTTP content [10, 65]. In comparison, Neptune is intended for clustered services with dynamic service fulfillment or content generation. This dissertation study on resource management for cluster-based network services addresses the inadequacy of the previous studies and complements them in the following three aspects.

- **Flexible resource management objectives.** Most previous studies have been using a monolithic metric to measure resource utilization and define QoS constraints. Commonly used ones include system throughput, mean response time, mean stretch factor [85], or the tail distribution of the response time [60]. Neptune introduces a unified metric that links the overall system efficiency with individual service response time. To be more specific, we consider the fulfillment of a service request produces certain *quality-aware service yield* depending on the response time, which can be linked to the amount of economical benefit or social reach resulting from serving this request. The overall goal of the system is to maximize the aggregate service yield resulting from all requests. As an additional goal, Neptune supports service differentiation for multiple service classes through two means: 1) service classes can acquire differentiated service support by being configured to produce different service yield; 2) each service class can also be guaranteed to receive a certain proportion of system resources, if so requested.

- **Fully decentralized clustering architecture.** Scalability and availability are always overriding concerns for large-scale cluster-based services. Several prior studies have been relying on centralized components to manage resources for a cluster of replicated servers [10, 23, 65, 85]. In contrast, our framework employs a functionally symmetrical architecture that does not rely on any centralized components. Such a design not only eliminates potential single point of failure in the system, it is also crucial to ensuring a smooth and prompt response to demand spikes and server failures.

- **Efficient resource utilization under quality constraints.** Neptune achieves efficient resource utilization through a two-level request distribution and scheduling scheme. At the cluster level, requests for each service class are evenly distributed to all replicated service nodes without explicit partitioning. Inside each service node, an adaptive scheduling policy adjusts to the runtime load condition and seeks high aggregate service yield at a wide range of load levels. When desired, the service scheduler also provides proportional resource allocation guarantee for specified service classes.

The rest of this chapter is organized as follows. Section 4.2 describes the multi-fold objective of our resource management framework. Section 4.3 presents Neptune's two-level request distribution and scheduling architecture. Section 4.4 illustrates the service scheduling inside each service node. Section 4.5 presents the system implementation and experimental evaluations based on traces and service components from a commercial search engine. Section 4.6 discusses related work and Section 4.7 concludes this chapter.

4.2 Resource Management Objective

In this section, we illustrate Neptune's multi-fold resource management objective. We first introduce the concepts of quality-aware service yield and service yield functions. Through these concepts, service providers can express a variety of quality constraints based on the service response time. Furthermore, using service yield functions and resource allocation guarantees, our framework allows service providers to determine the desired level of service differentiation among multiple service classes.

4.2.1 Quality-aware Resource Utilization

Most previous studies have been using a monolithic metric such as system throughput, mean response time, mean stretch factor [85], or the tail distribution of the response time [60] to measure the efficiency of system resource management. We use a more comprehensive metric by conceiving that the fulfillment of a service request provides certain yield depending the response time. This yield, we call *quality-aware service yield*, can be linked to the amount of economical benefit or social reach resulting from serving this request in a timely fashion. Both goals of provisioning QoS and efficient resource utilization can be naturally combined as producing high aggregate yield. Furthermore, Neptune considers the service yield resulting from serving each request to be a function of the service response time. The service yield function is normally determined by service providers to give them flexibility in expressing desired service qualities. Let $r_1, r_2, \cdots, r_k$ be the response times of the k service accesses completed in an operation period. Let $Y_i()$ represent the service yield function for the ith service access. The goal of our system is to maximize the aggregate yield, i.e.

$$\text{maximize} \sum_{i=1}^{k} Y_i(r_i). \tag{4.2.1}$$

We can also illustrate the concept of quality-aware service yield using an economical model. Basically we consider the fulfillment of each service request results in certain economical revenue for the service providers, the exact value of which depends on the service response time. The goal of the system, in this case, is to maximize the aggregate economical revenue resulting from all served requests.

The concept of service quality in this model refers to only the service response time. We do realize that service quality can have various additional dimensions, oftentimes application-specific. For instance, the partial failure in an partitioned search database results in a loss of *harvest* [39]. Neptune focuses on the service response time because it is one of the most general service qualities so that the proposed techniques can be effective for a large number of applications. In addition, we believe the current framework can be enhanced to support application-specific service qualities like harvest in a partitioned search engine.

In general, the service yield function can be any monotonically non-increasing function that returns non-negative numbers with non-negative inputs. We give a few examples to illustrate how service providers can use yield functions to express desired service qualities. For instance, the system with the yield function $Y_{\text{throughput}}$ depicted in Equation (4.2.2) is intended to achieve high system throughput with a deadline D. In other words, the goal of such a system is to complete as many service accesses as possible with the response time $\leq D$. Similarly, the system with the yield function Y_{resptime} in Equation (4.2.3) is designed to achieve low mean response time. Note that the traditional concept of mean response time does not count dropped requests. Y_{resptime} enhances that concept by considering dropped requests as if they are completed in D.

$$Y_{\text{throughput}}(r) = \begin{cases} C & \text{if } 0 \le r \le D, \\ 0 & \text{if } r > D. \end{cases} \tag{4.2.2}$$

$$Y_{\text{resptime}}(r) = \begin{cases} C(1 - \frac{r}{D}) & \text{if } 0 \le r \le D, \\ 0 & \text{if } r > D. \end{cases} \tag{4.2.3}$$

We notice that $Y_{\text{throughput}}$ does not care about the exact response time of each service access as long as it is completed within the deadline. In contrast, Y_{resptime} always reports higher yield for accesses completed faster. As a hybrid version of these two, Y_{hybrid} in Equation (4.2.4) produces full yield when the response time is within a pre-deadline D', and the yield decreases linearly thereafter. The yield finally declines to a *drop penalty* C' when the response time reaches the deadline D. The drop penalty can be used to reflect the importance of requests such that a large value of C' discourages dropping requests when their deadlines are close.

$$Y_{\text{hybrid}}(r) = \begin{cases} C & \text{if } 0 \le r \le D', \\ C - (C - C')\frac{r - D'}{D - D'} & \text{if } D' < r \le D, \\ 0 & \text{if } r > D. \end{cases} \tag{4.2.4}$$

This hybrid version corresponds to the real world scenario that users are generally comfortable as long as a service request is completed in D'. They get more or less annoyed when the service takes longer and they most likely abandon the service after waiting for D. C represents the full yield resulting from a prompt response and the drop penalty C' represents the loss when the service is not completed within the final deadline D. Figure 4.2 gives the illustration of these three functions. We want to point out that $Y_{\text{throughput}}$ is a special case of Y_{hybrid} when $D' = D$; and Y_{resptime} is also a special case of Y_{hybrid} when $D' = 0$ and $C' = 0$. Figure 4.2 gives an illustration of these three types of yield functions.

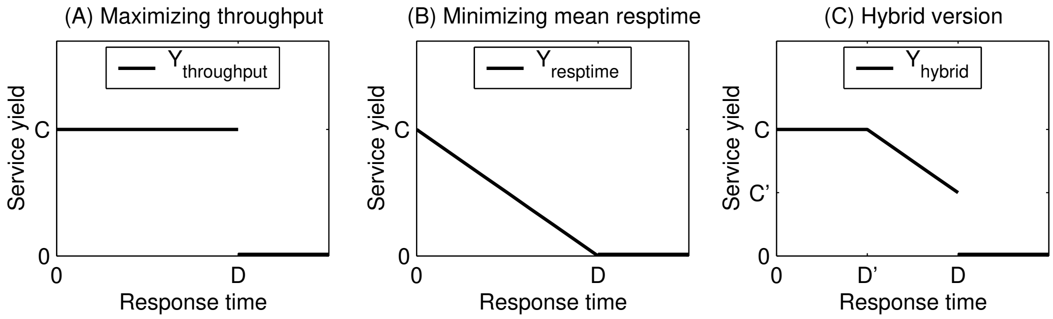

Figure 4.2: Illustration of service yield functions.

In certain sense, our definition of service yield is similar to the concept of *value* in value-based real-time database systems [50, 52]. One major distinction is that real-time systems are usually sized to handle transient heavy load [50]. For Internet services, however, the client request rates tend to be bursty and over-provisioning system resources for a service site to accommodate the potential peak will not be cost-effective [13, 23, 27]. More detailed discussions on scheduling schemes to achieve high aggregate service yield are given in Section 4.4.2.

4.2.2 Service Differentiation

Service differentiation is another goal of our multi-fold resource management objective. Service differentiation is based on the concept of service classes. A *service class* is defined as a category of service accesses that obtain the same level of service support. On the other hand, service accesses belonging to different service classes may receive differentiated QoS support. Service classes can be defined based on client identities. For instance, a special group of clients may be configured to receive preferential service support or a guaranteed share of system resources. Service classes can also be defined based on service types or data partitions. For example, a order placement transaction is typically considered more important than a catalog-browsing request. A previous research on dynamic page caching and invalidation has proposed a flexible service class specification language and how it can be implemented efficiently [86]. In this dissertation study, we simply use these results for request classification.

Our framework provides differentiated services to different service classes on two fronts. First, service classes can acquire differentiated service support by specifying different yield functions. For instance, serving a VIP-class client can be configured to produce higher service yield than serving a regular client. Secondly, each service class can be guaranteed to receive a certain portion of system resources. Most previous service differentiation studies have focused on one of the above two means of QoS support [17, 55, 63, 80]. We believe a combination of them provide two benefits when system is overloaded: 1) the resource allocation is biased toward high-yield classes for efficient resource utilization; 2) a certain portion of system resources can be guaranteed for each service class, if needed. The second benefit is crucial to preventing starvation for low-priority service classes.

4.3 Two-level Request Distribution and Scheduling

In our framework, each external service request enters the service cluster through one of the protocol gateways and it is classified into one of the service classes according to rules specified by service providers. The gateway node then accesses one or more (in the case of service aggregation) internal services to fulfill the request. Inside the service cluster, each service can be made available at multiple nodes through replication. In this section, we discuss the cluster-level request distribution on the replicated servers for a single service on a single data partition or a partition group.

The dynamic partitioning approach proposed in a previous study adaptively partitions all replicas for each service into several groups and each group is assigned to handle requests from one service class [85]. We believe such a scheme has a number of drawbacks. First, a cluster-wide scheduler is required to make server partitioning decisions, which is not only a single-point of failure, but also a potential performance bottleneck. Secondly, cluster-wide server groups cannot be repartitioned very frequently, which makes it difficult to respond promptly to changing resource demand. In order to address these problems, Neptune does not explicitly partition server groups. Instead, we use a symmetrical and decentralized two-level request distribution and scheduling architecture illustrated in Figure 4.3.

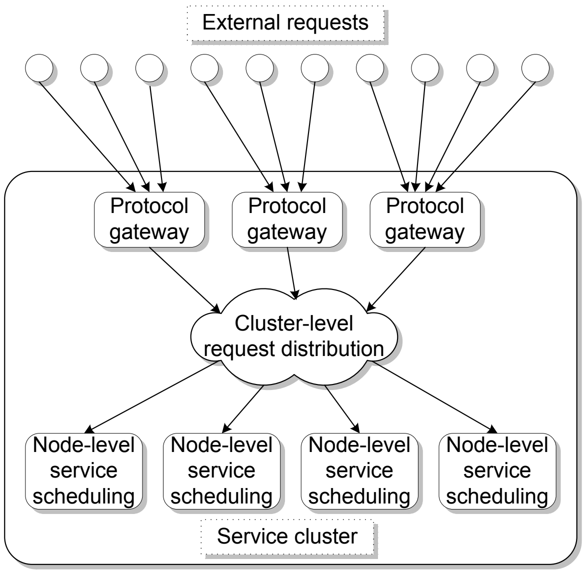

Figure 4.3: Two-level request distribution and scheduling.

Each service node in this architecture can process requests from all service classes. The resource management decision is essentially made at two levels. First, each service request is directed to one of the replicated service node through a cluster-level request distribution. Upon arriving at the service node, it is then subject to a node-level service scheduling. At the cluster level, Neptune employs a class-aware load balancing scheme to evenly distribute requests for each class to all servers. Our load balancing scheme uses a random polling policy that discards slow-responding polls. Under this policy, whenever a client is about to seek a service for a particular service class, it polls a certain number of randomly selected service nodes to obtain the load information. Then it directs the service request to the node with the smallest number of active and queued requests. Polls not responded within a deadline are discarded. This strategy also helps excluding faulty nodes from request distribution. We use a poll size of 3 and a polling deadline of 10 ms in our system. Our study in Chapter 3 shows that such a policy is scalable and well suited for services of a large spectrum of granularities. Inside each service node, our approach must also deal with the resource allocation across multiple service classes. This is handled by a node-level class-aware scheduling scheme, which will be discussed in Section 4.4. Note that the node-level class-aware scheduling is not necessary for the server partitioning approach because every node is configured to serve a single service class under that approach.

An Alternative Approach for Comparison. For the purpose of comparison, we also design a request distribution scheme based on server partitioning [85]. Server partitioning is adjusted periodically at fixed intervals. This scheme uses the past resource usage to predict the future resource demand and makes different partitioning decisions during system under-load and overload situations.

- When the aggregate demand does not exceed the total system resources, every service class acquires their demanded resource allocation. The remaining resources will be allocated to all classes proportional to their demand.

- When the system is overloaded, in the first round we allocate to each class its resource demand or its resource allocation guarantee, whichever is smaller. Then the remaining resources are allocated to all classes under a priority order. The priority order is sorted by the full yield divided by the mean resource consumption for each class, which can be acquired through offline profiling.

Fractional server allocations are allowed in this scheme. All servers are partitioned into two pools, a dedicated pool and a shared pool. A service class with 2.4 server allocation, for instance, will get two servers from the dedicated pool and acquire 0.4 server allocation from the shared pool through sharing with other classes with fractional allocations.

The length of the adjustment interval should be chosen carefully so that it is not too small to avoid excessive repartitioning overhead and maintain system stability, nor is it too large to promptly respond to demand changes. We choose the interval to be 10 seconds in this study. Within each allocation interval, service requests are randomly directed to one of the servers allocated to the corresponding service class according to the load balancing policy.

4.4 Node-level Service Scheduling

Neptune employs a multi-queue (one per service class) scheduler inside each node. Whenever a service request arrives, it enters the appropriate queue for the service class it belongs to. When resources become available, the scheduler dequeues a request from one of the queues for service. Figure 4.4 illustrates such a runtime environment of a service node. This scheduling framework does not directly consider dependencies or associations among service requests like those belonging to one client session. However, we believe features like providing preferential service qualities for particular clients can be indirectly supported through the proper configuration of service classes.

For a service node hosting N service classes: $C_1, C_2, \cdots, C_N$, each class C_k is configured with a service yield function Y_k and optionally a minimum

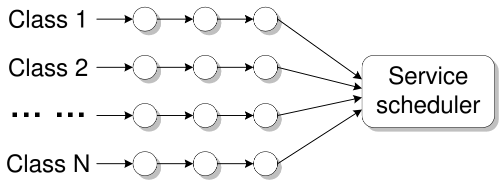

Figure 4.4: Runtime environment of a service node.

system resource share guarantee g_k, which is expressed as a percentage of total system resources ($\sum_1^N g_k \leq 1$). The goal of the scheduling scheme is to provide the guaranteed system resources for all service classes and schedule the remaining resources to achieve high aggregate service yield. Note that when $\sum_{k=1}^N g_k = 1$, our system falls back to a static resource partitioning scheme. Figure 4.5 illustrates the framework of our service scheduling algorithm at each scheduling point. In the rest of this section, we will discuss two aspects of the scheduling algorithm: 1) maintaining resource allocation guarantee; and 2) achieving high aggregate service yield.

4.4.1 Estimating Resource Consumption for Allocation Guarantee

In order to maintain resource allocation guarantee, we need to estimate resource consumption for each service class at each scheduling time. This

1. Drop from each queue head those requests that are likely to generate zero or very small yield according to the request arrival time, expected service time and the yield function.

2. Search for the service classes with non-empty request queues that have an estimated resource consumption of less than the guaranteed share.

   (a) If found, schedule the one with the largest gap between the resource consumption and the guaranteed share.

   (b) Otherwise, schedule a queued request that is likely to produce high aggregate service yield.

Figure 4.5: The node-level service scheduling algorithm.

estimation should be biased toward recent usage to stabilize quickly when the actual resource consumption jumps from one level to another. It should not be too shortsighted either in order to avoid oscillations or over-reactions to short-term spikes. Among many possible functions that exhibit those properties, we define the resource consumption for class C_k at time t to be the weighted summation of the resource usage for all class C_k requests completed no later than t. The weight is chosen to decrease exponentially with regard to the elapsed time since the request completion. For each request r, let $ct(r)$ be its completion time and $s(r)$ be its actual resource usage (we will discuss how to measure it in the end of this sub-section), which is known after its completion. Equation 4.4.1 defines $u_k(t)$ to be the resource consumption for class C_k at time t. Note that the time in all the following equations is denominated in *seconds*.

$$u_k(t) = \sum_{\{r|r \in C_k \text{ and } ct(r) \leq t\}} \beta^{t-ct(r)} s(r), \qquad 0 < \beta < 1 \tag{4.4.1}$$

Another reason for which we choose this function is that it can be incrementally calculated without maintaining the entire service scheduling history. Let t' be the previous calculation time, the resource consumption at time t can be calculated incrementally through Equation 4.4.2.

$$u_k(t) = \beta^{t-t'} u_k(t') + \sum_{\{r|r \in C_k \text{ and } t' < ct(r) \leq t\}} \beta^{t-ct(r)} s(r) \tag{4.4.2}$$

If we adjust $u_k(t)$ at the completion of every request, or, in other words, there is only one request completed between t' and t, then Equation 4.4.2 becomes Equation 4.4.3 where r is the most recent request.

$$u_k(t) = \beta^{t-t'} u_k(t') + \beta^{t-ct(r)} s(r) \tag{4.4.3}$$

The selection of β should be careful to maintain the smooth and stable reaction for both short-term spikes and long-term consumption changes. In this study we empirically choose β to be 0.95. Since we use *second* as the unit of time in those equations, this means a service request completed one second ago carries 95% the weight of a service request completed right now.

With the definition of $u_k(t)$, the proportional resource consumption of class C_k can be represented by $\frac{u_k(t)}{\sum_{k=1}^{N} u_k(t)}$. In step 2 of the service scheduling, this proportional consumption is compared with the guaranteed share to search for under-allocated service classes.

Our estimation scheme is related to the exponentially-weighted moving average (EWMA) filter used as the round-trip time predictor in TCP [34]. It differs from the original EWMA filter in that the weight in our scheme decreases exponentially with regard to the elapsed time instead of the elapsed number of measurement samples. This is more appropriate for estimating resource consumption due to its time-decaying nature.

The detailed measurement of resource consumption $s(r)$ for each request r is application-dependent. Generally speaking, each request can involve mixed CPU and I/O activities and it is difficult to define a generic formula for all applications. Our approach is to let application developers decide how the resource consumption should be accounted. Large-scale service clusters are typically composed of multiple sub-clusters of replicated service components [13]. Each sub-cluster typically hosts a single type of service for modularity and easier management. Thus requests in the same sub-cluster tend to share similar resource characteristics in terms of I/O and CPU demand and it is not hard in practice to identify a suitable way to account resource consumptions. In the current implementation, we use the accumulated CPU consumption for a thread or process acquired through Linux `/proc` filesystem. This accounting is fairly effective in our evaluations even though one of the evaluation benchmarks involves significant disk I/O.

4.4.2 Achieving High Aggregate Yield

In this section, we examine the policies employed in step 2b of the service scheduling in Figure 4.5 to achieve high aggregated yield. In general, the optimization problem specified in Equation 4.2.1 is difficult to solve given the fact that it relies on the future knowledge of the response time of pending requests. Various priority-based scheduling policies were proposed in real-time database systems to maximize aggregate realized value [50, 52]. Typical policies considered in those systems include Earliest Deadline First scheduling

(*EDF*) and Yield or Value-Inflated Deadline scheduling (*YID*). EDF always schedules the queued request with the closest deadline. YID schedules the queued request with the smallest inflated deadline, defined as the relative deadline divided by the expected yield if the request is being scheduled.

Both EDF and YID are designed to avoid or minimize the amount of lost yield. They work well when the system resources are sized to handle transient heavy load [50]. For Internet services, however, the client request rates tend to be bursty and fluctuate dramatically from time to time [13, 23, 27]. Over-provisioning system resources for a service site to accommodate the potential peak will not be cost-effective. During load spikes when systems are facing sustained arrival demand exceeding the available resources, missed deadlines become unavoidable and the resource management should instead focus on utilizing resources in the most efficient way. This leads us to design a *Greedy* scheduling policy that schedules the request with the lowest resource consumption per unit of expected yield.

In order to have a scheduling policy that works well at a wide range of load conditions, we further design an *Adaptive* policy that dynamically switches between YID and Greedy scheduling depending on the runtime load condition. The scheduler maintains a 30-second window of recent request dropping statistics. If more than 5% of incoming requests are dropped in the watched window, the system is considered as overload and the Greedy scheduling is employed. Otherwise, the YID scheduling is used.

All the above scheduling polices are priority-based scheduling with different definition of priorities. Table 4.1 summarizes the priority metrics of the four policies.

Policy	**Priority** (the smaller the higher)
EDF	Relative deadline
YID	Relative deadline divided by expected yield
Greedy	Expected resource consumption divided by expected yield
Adaptive	Dynamically switch between YID (in under-load) and Greedy (in overload)

Table 4.1: Summary of scheduling policies.

Three of above policies require a predicted response time and resource consumption for each request at the scheduling time. For the response time, we use an exponentially-weighted moving average of the response time of past requests belonging to the same service class. An accurate prediction of the resource consumption typically demands service-specific knowledges, including the resource usage pattern of the application-level services and the input parameter for individual service accesses. In our current implementation, such a prediction is based on an exponentially-weighted moving average of the CPU consumptions of past requests belonging to the same service class. Such an approximation does not affect the applicability of the proposed scheduling policies as our evaluation in Section 4.5 demonstrates.

4.5 System Implementation and Experimental Evaluations

The proposed resource management framework has been implemented on top of Neptune clustering architecture. Each external service request is assigned a service class ID upon arriving at any of the gateways. Those requests are directed to one of the replicated service nodes according to the class-aware load balancing scheme. Each server node maintains multiple request queues (one per service class) and a thread pool. To process each service request, a thread is dispatched to invoke the application service module through dynamically-linked libraries. The size of the thread pool is chosen to strike the balance between concurrency and efficiency depending on the application characteristics. The aggregate services are exported to external clients through protocol gateways.

The first goal of the performance evaluation is to examine the system performance of our service scheduling schemes over a wide range of load conditions. Secondly, we will study the performance and scalability of the proposed cluster-level request distribution scheme. Our third goal is to investigate the system behavior in terms of service differentiation during demand spikes and server failures. All the evaluations were conducted on a rack-mounted Linux cluster with around 30 dual 400 MHz Pentium II nodes, each of which contains either 512 MB or 1 GB memory. Each node runs Linux 2.2.15 and has two 100 Mb/s Ethernet interfaces. The cluster is connected by a Lucent P550 Ethernet switch with 22 Gb/s backplane bandwidth.

4.5.1 Evaluation Workloads

Our evaluation studies are based on two service workloads. The first service is a *Differentiated Search* service based on an index search component from Ask Jeeves search. This service takes in a group of encoded query words; checks an memory mapped index database; and returns the identifications of the list of documents matching input query words. The index database size is around 2.5 GB at each node and it cannot completely fit in memory. The mean service time for this service is around 250 ms in our testbed when each request is served in a dedicated environment.

Differentiated Search distinguishes three classes of clients, representing Gold, Silver and Bronze memberships. We let the request composition for these three classes be 10%, 30%, 60% respectively. The yield functions of these service classes can be one of the three forms that we described in Section 4.2.1, i.e. $Y_{\text{throughput}}()$, $Y_{\text{resptime}}()$, or $Y_{\text{hybrid}}()$. In each case, the shapes of the yield functions for three service classes are the same other than the magnitude. We determine the ratio of such magnitudes to be 4:2:1. The deadline D is set to be 2 seconds. In the case of $Y_{\text{hybrid}}()$, the drop penalty C' is set to be half of the full yield and the pre-deadline D' is set to be half of the absolute deadline D. Figure 4.6 illustrates the yield functions when they are in each one of the three forms.

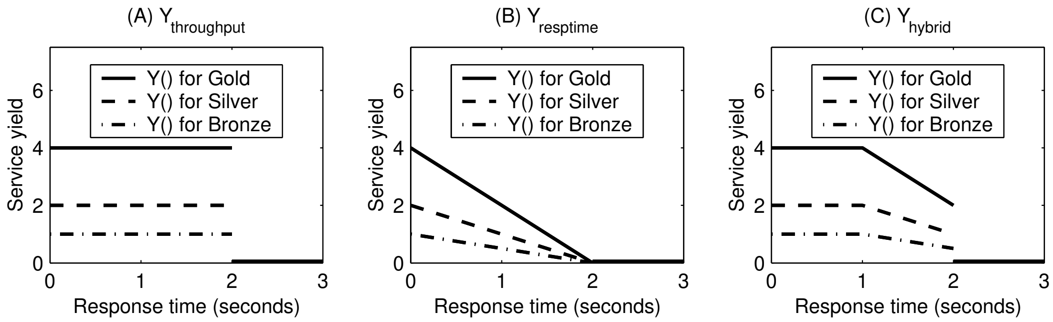

Figure 4.6: Service yield functions in evaluation workloads.

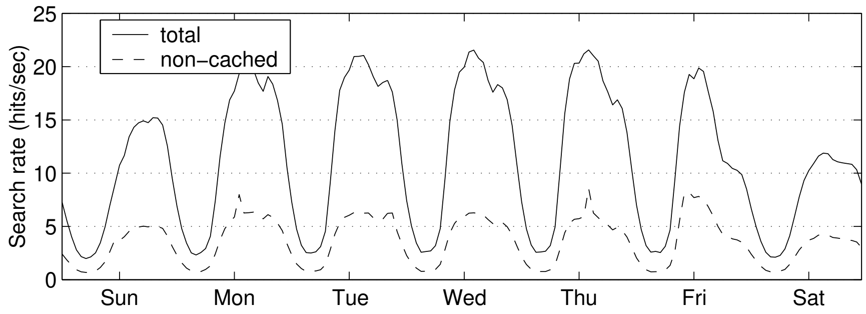

Figure 4.7: Search requests to Ask Jeeves search via one of its edge Web servers (January 6-12, 2002).

The request arrival intervals and the query words for the three Differentiated Search classes are based on a one-week trace we collected at Ask Jeeves search via one of its edge Web servers. Figure 4.7 shows the total and non-cached search rate of this trace. The search engine employs a query cache to directly serve those queries that have already been served before and cached. The cached requests are of little interests in our evaluation because they consume very little system resources. We use the peak-time portion of Tuesday, Wednesday, and Thursday's traces to drive the workload for Gold, Silver, and Bronze classes respectively. For each day, the peak-time portion we choose is a 7-hour period from 11am to 6pm EST. The statistics of these three traces are listed in Table 4.2. Note that the arrival intervals of these traces may be scaled when necessary to generate workloads at various demand levels during our evaluation.

	Number of accesses		Arrival interval		Service time	
	Total	Non-cached	Mean	Std-dev	Mean	Std-dev
Gold	507,202	154,466	161.3ms	164.3ms	247.9ms	218.2ms
Silver	512,227	151,827	166.0ms	169.5ms	249.7ms	219.3ms
Bronze	517,116	156,214	161.3ms	164.7ms	245.1ms	215.7ms

Table 4.2: Statistics of evaluation traces.

The three service classes in Differentiated Search are based on the same service type and thus have the same average resource consumption. The second service we constructed for the evaluation is designed to have different resource consumption for each service class, representing services differentiated on their types. This service, we call *Micro-benchmark*, is based on a CPU-spinning micro-benchmark. It contains three service classes with the same yield functions as the Differentiated Search service. The mean service times of the three classes are 400 ms, 200 ms, and 100 ms respectively. We use Poisson process arrivals and exponentially distributed service times for the Micro-benchmark service.

4.5.2 Evaluation on Node-level Scheduling and Service Differentiation

In this section, we study the performance of four service scheduling policies (EDF, YID, Greedy and Adaptive) and their impact on service differentiation. The performance metric we use in this study is *LossPercent* [50], which is computed as

$$\text{LossPercent} = \frac{\text{OfferedYield} - \text{RealizedYield}}{\text{OfferedYield}} \times 100\%$$

OfferedYield is the aggregated full yield of all arrived requests and *RealizedYield* is the amount of yield realized by the system.

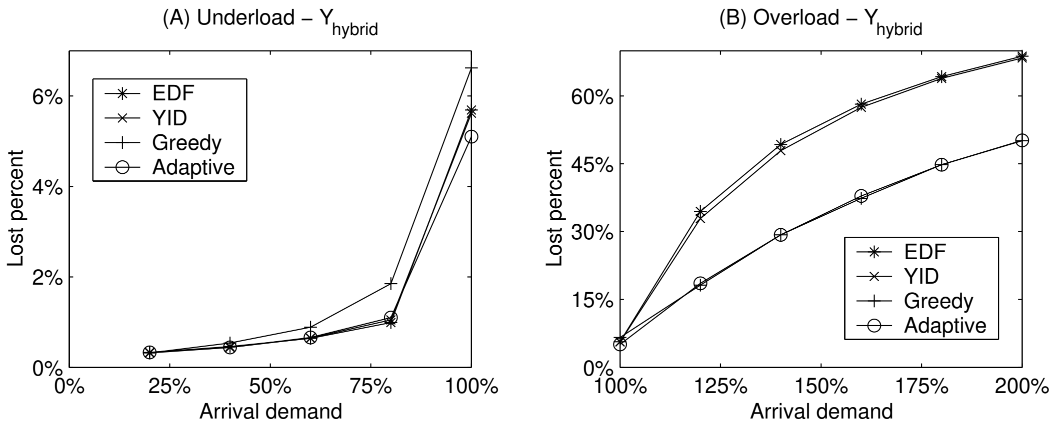

Figure 4.8: Performance of scheduling policies on Micro-benchmark.

Figure 4.9 shows the performance of scheduling policies on Differentiated

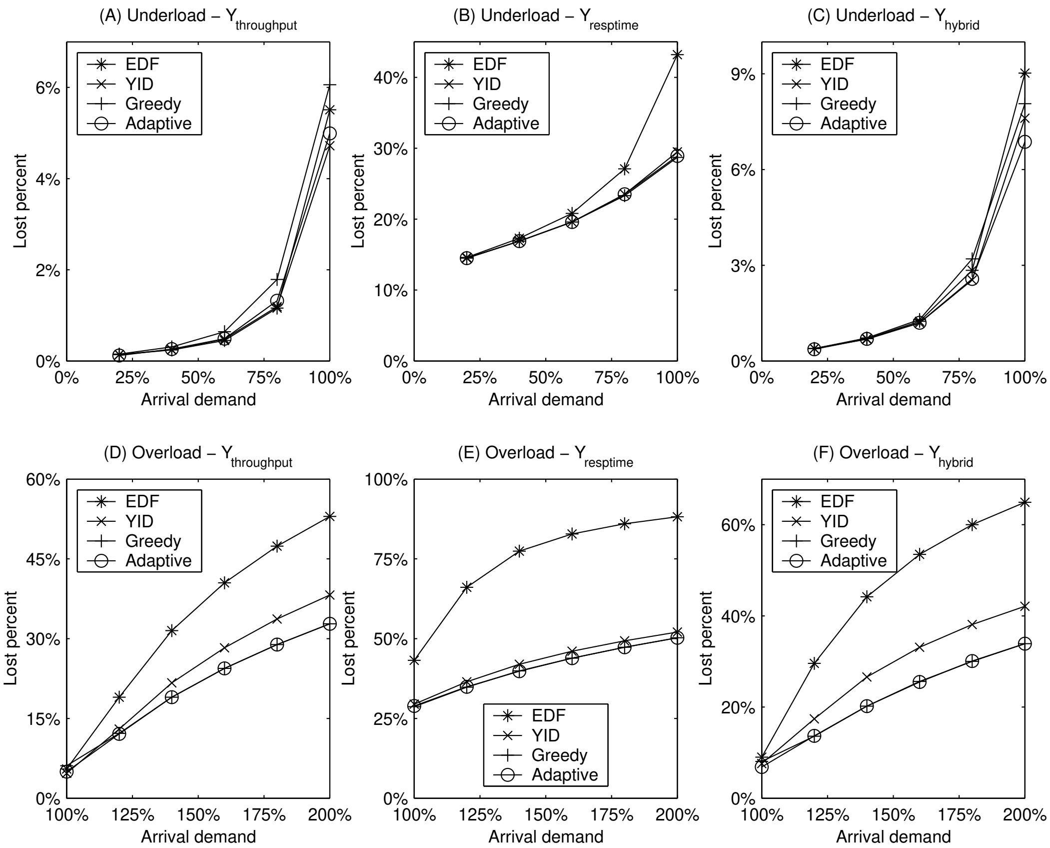

Figure 4.9: Performance of scheduling policies on Differentiated Search.

Search in a single-server setting. The experiments were conducted for all three forms of yield functions: $Y_{\text{throughput}}()$, $Y_{\text{resptime}}()$, and $Y_{\text{hybrid}}()$. Figure 4.8 shows the performance of scheduling policies on Micro-benchmark. Only the result for yield functions in $Y_{\text{hybrid}}()$ form is shown for this service. In each case, we show the performance results with a varying arrival demand of up to

200% of the available resources. The demand level cannot simply be the mean arrival rate times the mean service time due to various system overhead. We probe the maximum arrival rate such that more than 95% of all requests are completed within the deadline under EDF scheduling. Then we consider the request demand is 100% at this arrival rate. The desired demand level is then achieved by scaling the request arrival intervals. The performance results are separated into the under-load (arrival demand $\leq$ 100%) and overload (arrival demand $\geq$ 100%) situations. We employ no minimum resource guarantee for both services to better illustrate the comparison on the aggregate yield.

We observe that YID outperforms Greedy by up to 49% when the system is under-loaded and Greedy performs up to 39% better during system overload. The Adaptive policy is able to dynamically switch between YID and Greedy policies to achieve good performance on all studied load levels.

To further understand the performance difference among the scheduling policies and the impact on service differentiation, Figure 4.10 lists the per-class performance breakdown for Differentiated Search service with $Y_{\mathrm{hybrid}}()$ yield functions under 200% arrival demand. In terms of the throughput, all four policies achieve similar aggregate throughput, however, Greedy and Adaptive policies complete more requests of higher-priority classes, representing more

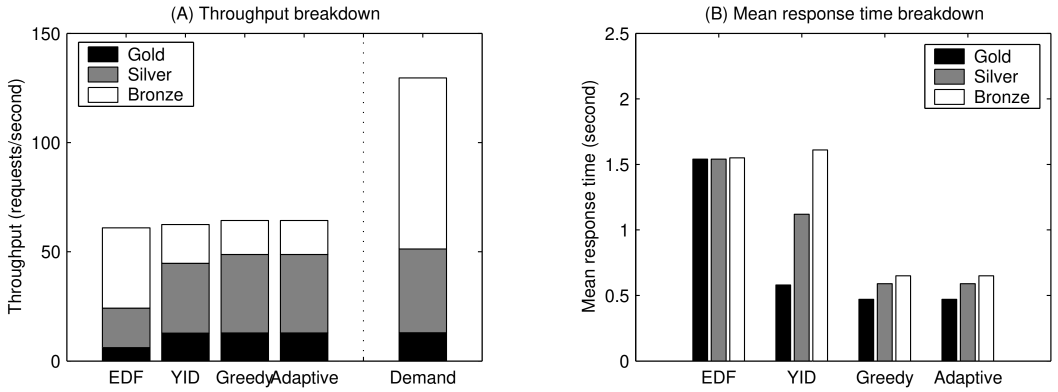

Figure 4.10: Per-class performance breakdown of Differentiation Search at 200% arrival demand.

efficient resource utilization. In terms of the mean response time, Greedy and Adaptive policies complete requests with shorter mean response time, representing better quality for completed requests.

4.5.3 Evaluation on Request Distribution across Replicated Servers

Figure 4.11 illustrates our evaluation results on two request distribution schemes: class-aware load balancing (used in Neptune) and server partitioning. For each service, we show the aggregate service yield of up to 16 replicated servers under slight under-load (75% demand), slight overload (125% demand), and severe overload (200% demand). The Adaptive scheduling policy is used

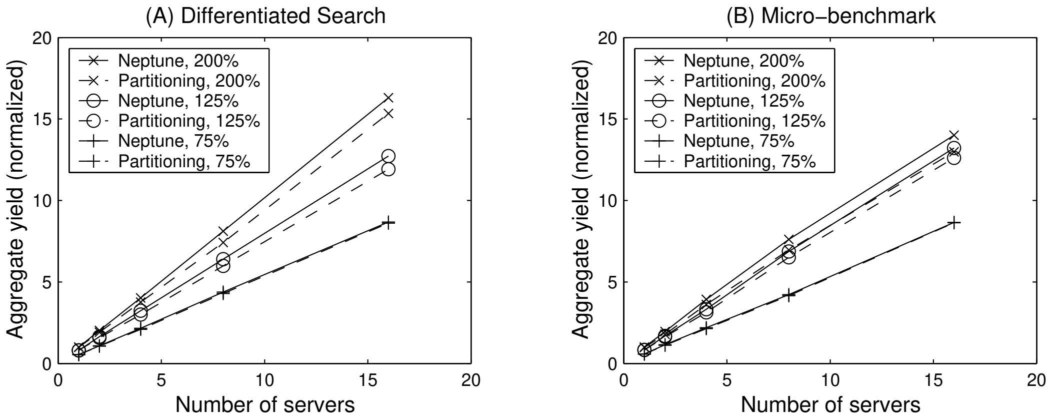

Figure 4.11: Performance and scalability of request distribution schemes.

in each server for those experiments. The aggregate yield shown in Figure 4.11 is normalized to the Neptune yield under 200% arrival demand. Our result shows that both schemes exhibit good scalability, which is attributed to our underlying load balancing strategy, the random-polling policy that discards slow-responding polls. In comparison, Neptune produces up to 6% more yield than server partitioning under high demand. This is because Neptune allows the whole cluster-wide load balancing for all service classes while server partitioning restricts the scope of load balancing to the specific server partition for the corresponding service class, which affects the load balancing performance.

4.5.4 Service Differentiation during Demand Spikes and Server Failures

In this section, we study the service differentiation during demand spikes and server failures. We use the Differentiated Search service with 20% resource guarantee for each class. In order to produce constantly controllable demand levels, we altered this service to generate fixed interval request arrivals and constant service time. Figure 4.12 illustrates the system behavior of such a service under Neptune and server partitioning approaches in a 16-server configuration. For each service class, we show the resource demand and the resource allocation, measured in two-second intervals, over a 300-second period.

Initially the total demand is 100% of the available resources, with 10%, 30%, and 60% of which belong to Gold, Silver, and Bronze class respectively. Then there is a demand spike for the Silver class between time 50 and time 150. We observe that Neptune promptly responds to the demand spike by allocating more resources to meet high-priority Silver class demand and dropping some low-priority Bronze class requests. This shift stops when Bronze class resource allocation drops to around 20% of total system resources, which is its guaranteed share. We also see the resource allocations for Silver and Bronze class quickly stabilize when they reach new allocation levels. In comparison, the server partitioning scheme responds to this demand spike in a slower pace because it cannot adjust to immediate demand changes until the next allocation interval. We also observe that the resource allocation for the highest-priority Gold class is isolated from this demand spike under both schemes.

At time 200, one server (allocated to the Gold class under server partitioning) fails and it recovers at time 250. Immediately after the server failure, we see a deep drop of Gold class resource allocation for about 10 seconds under server partitioning. This is again because it cannot adjust to immediate resource change until the next allocation interval. In comparison, Neptune exhibits much smoother behavior because losing any one server results in a proportional loss of resources for each class. Also note that the loss of a server reduces the available resources, which increases the relative demand to the available resources. This effectively results in another resource shortage. The system copes with it by maintaining enough allocation to Gold and Silver classes while dropping some of the low-priority Bronze class requests.

4.6 Related Work

Quality-of-service support and service differentiation. The importance of providing QoS support and service differentiation has been recognized in the networking community and the focuses of these studies are network bandwidth allocation and packet delay [18, 29, 58, 59, 65, 74]. The methods for ensuring bandwidth usage include delaying or dropping user requests [29, 59, 65] or reducing service qualities [1, 22]. In comparison, Neptune focuses on achieving efficient resource utilization and providing service differentiation for cluster-based services in which contents are dynamically generated and aggregated. Recent advances in OS research have developed approaches to provide QoS support at OS kernel level [14, 20, 30, 62, 73, 79]. Our work can be enhanced by those studies to support hard QoS guarantees and service differentiation at finer granularities.

The concept of service quality in this resource management framework refers to only the service response time. Service quality can have various application-specific additional dimensions. For instance, the partial failure in an partitioned search database results in a loss of harvest [39]. Neptune focuses on the service response time because it is one of the most general service qualities so that the proposed techniques can be effective for a large number of applications. In addition, we believe the current framework can be enhanced to support application-specific service qualities like harvest in a partitioned search engine.

Resource management for clustered services. A large body of work has been done in request distribution and resource management for cluster-based server systems [10, 11, 23, 64, 75, 85]. In particular, demand-driven service differentiation (DDSD) provides a dynamic server partitioning approach to differentiating services from different service classes [85]. Similar to a few other studies [10, 23], DDSD supports service differentiation in the aggregate allocation for each service class. In comparison, Neptune employs request-level service scheduling to achieve high service throughput as well as limit the response time of individual service requests. In addition, Neptune utilizes a fully decentralized architecture to achieve high scalability and availability.

It is worth mentioning that a previous study has proposed locality-aware request distribution (LARD) to exploit application-level data locality for Web server clusters [64]. Neptune does not directly address the data locality issue because the service data for cluster-based data-intensive applications is typically partitioned such that the critical working set [13] or the whole service data [42] can fit into the system memory. This study of resource management is geared toward replicated service nodes for each such data partition.

Service scheduling. Deadline scheduling, proportional-share resource scheduling, and value-based scheduling have been studied in both real-time systems and general-purpose operating systems [17, 50, 52, 55, 63, 73, 80]. Client request rates for Internet services tend to be bursty and fluctuate dramatically from time to time [13, 23, 27]. Delivering satisfactory user experience is important during load spikes. Based on an adaptive scheduling approach and a resource consumption estimation scheme, the service scheduling in Neptune strives to achieve efficient resource utilization under quality constraints and provide service differentiation.

4.7 Concluding Remarks

This chapter presents the design and implementation of an integrated resource management framework for cluster-based network services. This framework is flexible in allowing service providers to express desired service qualities based on the service response time. At the cluster level, a scalable decentralized request distribution architecture ensures prompt and smooth response to service demand spikes and server failures. Inside each node, an adaptive multi-queue scheduling scheme is employed to achieve efficient resource utilization under quality constraints and provide service differentiation.

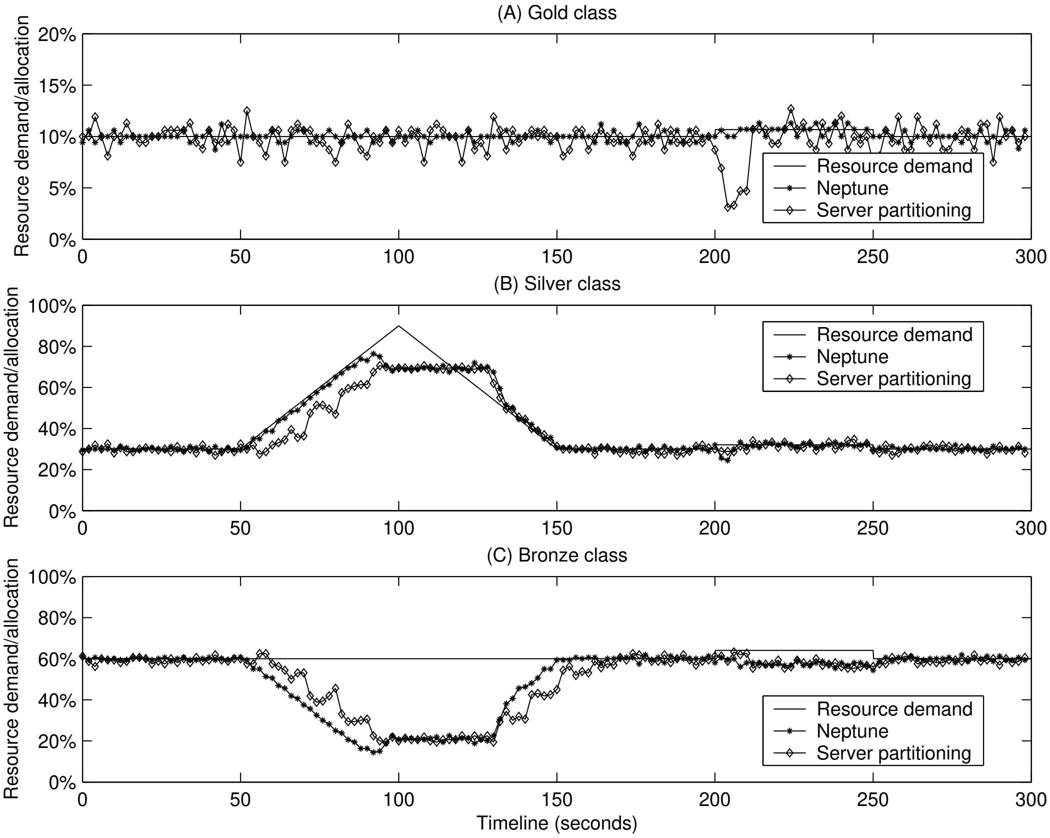

Figure 4.12: System behavior during demand spike and server failure with 16 servers. Differentiated Search with 20% resource guarantee for each class is used. One server (allocated to the Gold class under server partitioning) fails at time 200 and it recovers at time 250.

Chapter 5

Service Replication

5.1 Introduction

High availability, incremental scalability, and manageability are some of the key challenges faced by designers of Internet-scale network services and using a cluster of commodity machines is cost-effective for addressing these issues [9, 40, 67, 85]. Previous work has recognized the importance of providing software infrastructures for cluster-based network services. For example, the TACC and MultiSpace projects have addressed load balancing, fail-over support, and component reusability and extensibility for cluster-based services [40, 47]. These systems do not provide explicit support for managing frequently updated persistent service data and mainly leave the responsibility of storage replication to the service layer. Recently the DDS project has addressed replication of persistent data using a layered approach [46]. This work is focused on a class of distributed data structures and it cannot be easily applied to existing applications with complex data management logic.

Replication of persistent data is crucial to achieving high availability. Previous work has shown that synchronous replication based on *eager* update propagations does not deliver scalable solutions [8, 44]. Various asynchronous models have been proposed for wide-area or wireless distributed systems [5, 8, 44, 66]. However, these studies have not explicitly address the high scalability/availability demand and potentially weak consistency requirement of large-scale network services. Additional studies are needed to investigate the replica consistency and fail-over support for large-scale cluster-based Internet services.

In this chapter, we investigate techniques in providing scalable replication support for cluster-based network services. This work is built upon a large body of previous research in network service clustering, fault-tolerance, and data replication. The goal of this work is to provide flexible and efficient service replication support for network services with frequently updated persistent data. This model should make it simple to deploy existing applications and shield application programmers from the complexities of replica consistency and fail-over support. It also needs to have the flexibility to accommodate a variety of data management mechanisms that network services typically rely on. Under the above consideration, our system is designed to support multiple levels of replica consistency depending on application characteristics. The objective is to provide the desired level of replica consistency for large-scale Internet services with the emphasis on performance scalability and fail-over support.

Generally speaking, providing standard system components to achieve scalability and availability tends to decrease the flexibility of service construction. Neptune demonstrates that it is possible to achieve these goals by targeting partitionable network services and by providing multiple levels of replica consistency. Neptune addresses consistency in three levels. The highest level can ensure that network clients obtain and update information in a progressive order, which is desirable for many services. Data objects in network services are typically managed by databases or file systems and adding new object layers with replication support reduces the applicability of Neptune. To increase the compatibility with the existing applications, Neptune employs operation replication to maintain replica consistency indirectly.

The rest of this chapter is organized as follows. Section 5.1.1 presents the assumptions that Neptune's replication support is based upon. Section 5.2 describes Neptune's multi-level replica consistency scheme and the failure recovery model. Section 5.3 illustrates three service deployments on a Linux cluster. Section 5.4 evaluates Neptune's performance and failure management using those three services. Section 5.5 describes related work and Section 5.6 concludes this chapter.

5.1.1 Assumptions

We assume all hardware and software system modules follow the fail-stop failure model and network partitions do not occur inside the service cluster. We bear this principle in mind throughout the system implementation. Each module will simply terminate itself when an unexpected situation occurs. Nevertheless, we do not preclude catastrophic failures in our model. In other words, persistent data can survive through a failure that involves a large number of modules or even all nodes. In this case, the replica consistency will be maintained after the recovery. This is important because software failures are often not independent. For instance, a replica failure triggered by high workload results in even higher workload in remaining replicas and may cause cascading failures of all replicas. While the service will be certainly unavailable for a period of time, it is pivotal to maintain data consistency after recovery.

Neptune supports atomic execution of data operations through failures only if each underlying service module can ensure atomicity in a stand-alone configuration. This assumption can be met when the persistent data is maintained in transactional databases or transactional file systems. To facilitate atomic execution, we assume that each service module provides a CHECK callback so that the Neptune server module can check if a previously spawned service instance has been successfully completed. The CHECK callback is very similar to the REDO and UNDO callbacks that resource managers provide in the transaction processing environment [45]. It is only invoked during the node recovery phase and we will further discuss its usage and a potential implementation in Section 5.2.2.

We also assume that the service abort happens either at all replicas or not at all. Note that a server failure does not necessarily cause the active service request to abort because it can be logged and successfully reissued when the failing server recovers. This assumption can be met when service aborts are related to the state of service data only, e.g. violations of data integrity constraints. In other words, we do not consider the situations such that a service aborts because of insufficient disk space. Under this assumption, Neptune does not have to coordinate the service aborts because they either happen at all replicas or not at all. This assumption is not crucial to Neptune's fundamental correctness but it does greatly simplify our implementations. Without such an assumption, a proper UNDO mechanism will be required to maintain replica consistency.

5.2 Replica Consistency and Failure Recovery

In general, data replication is achieved through either *eager* or *lazy* write propagations [24, 44]. Eager propagation keeps all replicas exactly synchronized by acquiring locks and updating data at all replicas in a globally coordinated manner. In comparison, lazy propagation allows lock acquisitions and data updates to be completed independently at each replica. Previous work shows that synchronous eager propagation leads to high deadlock rates when the number of replicas increases [44]. The DDS project uses this synchronous approach and they rely on the timeout abort and client retry to resolve the deadlock issue [46]. In order to ensure replica consistency while providing high scalability, the current version of Neptune adopts a primary copy approach to avoid distributed deadlocks and a lazy propagation of updates to the replicas where the updates are completed independently at each replica. In addition, Neptune addresses the load-balancing problems of most primary copy schemes through data partitioning.

Lazy propagation introduces the problems of out-of-order writes and accessing stale data versions. Neptune provides a three-level replica consistency model to address these problems and exploit their performance tradeoff. Our consistency model extends the previous work in lazy propagation with a focus on high scalability and runtime fail-over support. Particularly Neptune's highest consistency level provides a staleness control which contains not only the quantitative staleness bound but also a guarantee of progressive version delivery for each client's service accesses. We can efficiently achieve this staleness control by taking advantage of the low latency, high throughput system-area network and Neptune's service publishing mechanism.

The rest of this section discusses the multi-level consistency model and Neptune's support for failure recovery. It should be noted that the current version of Neptune does not have full-fledged transactional support because Neptune restricts each service access to a single data partition.

5.2.1 Multi-level Consistency Model

As mentioned in Chapter 2, Neptune is targeted at partitionable network services in which service data can be divided into independent partitions. Therefore, Neptune's consistency model does not address the data consistency across partition boundaries. Neptune's first two levels of replica consistency are more or less generalized from the previous work [24, 67] and we provide an extension in the third level to address the data staleness problem from two different perspectives. Notice that a consistency level is specified for each service and thus Neptune allows co-existence of services with different consistency levels.

Level 1. Write-anywhere replication for commutative writes. In this level, each write is initiated at any replica and is propagated to other replicas asynchronously. When writes are commutative, eventually the client view will converge to a consistent state for each data partition. The append-only discussion groups in which users can only append messages satisfy this commutativity requirement. Another example is a certain kind of email service [67], in which all writes are total-updates, so out-of-order writes could be resolved by discarding all but the newest. The first level of replica consistency is intended for taking advantage of application characteristics and achieving high performance in terms of scalability and fail-over support.

Level 2. Primary-secondary replication for ordered writes. In this consistency level, writes for each data partition are totally ordered. A primary-copy node is assigned to each replicated data partition, and other replicas are considered as secondaries. All writes for a data partition are initiated at the primary, which asynchronously propagates them in a FIFO order to the secondaries. At each replica, writes for each partition are serialized to preserve the order. Serializing writes simplifies the write ordering for each partition, but it results in a loss of write concurrency within each partition. Since many Internet services have a large number of data partitions due to the information and user independence. There should be sufficient write concurrency across partitions. Besides, concurrency among read operations is not affected by this scheme. Level two consistency provides the same client-viewed consistency support as level one without requiring writes to be commutative. As a result, it could be applied to more services.

Level 3. Primary-secondary replication with staleness control. Level two consistency is intended to solve the out-of-order write problem resulting from lazy propagation. This additional level is designed to address the issue of accessing stale data versions. The primary-copy scheme is still used to order writes in this level. In addition, we assign a version number to each data partition and this number increments after each write. The staleness control provided by this consistency level contains two parts: 1) *Soft quantitative bound.* Each read is serviced at a replica that is at most x seconds stale compared to the primary version. The quantitative staleness between two data versions is defined by the elapsed time between the two corresponding writes accepted at the primary. Thus our scheme does not require a global synchronous clock. Currently Neptune only provides a soft quantitative staleness bound and it is described later in this section. 2) *Progressive version delivery.* From each client's point of view, the data versions used to service her read and write accesses should be monotonically non-decreasing. Both guarantees are important for services like large-scale on-line auction in which users would like to get as recent information as possible and they do not expect to see declining bidding prices in two consecutive accesses.

We explain below our implementations for the two staleness control guarantees in level three consistency. The quantitative bound ensures that all reads are serviced at a replica at most x seconds stale compared to the primary version. In order to achieve this, each replica publishes its current version number as part of the service announcement message and the primary publishes its version number at x seconds ago in addition. With this information, Neptune client module can ensure that all reads are only directed to replicas within the specified quantitative staleness bound. Notice that the published replica version number may be stale depending on the service publishing frequency, so it is possible that none of the replicas has a high enough version number to fulfill a request. In this case, the read is directed to the primary, which always has the latest version. Also note that the "x seconds" is only a soft bound because the real guarantee depends on the latency, frequency and intermittent losses of service announcements. However, these problems are insignificant in a low latency, reliable system area network.

The progressive version delivery guarantees that: 1) After a client writes to a data partition, she always sees the result of this write in her subsequent reads. 2) A user never reads a version that is older than another version she has seen before. In order to accomplish this, each service invocation returns a version number to the client side. For a read, this number stands for the data version used to fulfill this access. For a write, it stands for latest data version as a result of this write. Each client keeps this version number in a `NeptuneHandle` and carries it in each service invocation. The Neptune client module can ensure that each client read access is directed to a replica with a published version number higher than any previously returned version number.

5.2.2 Failure Recovery

In this section, we focus on the failure detection and recovery for the primary-copy scheme that is used in level two/three consistency schemes. The failure management for level one consistency is much simpler because the replicas are more independent from each other.

In order to recover lost propagations after failures, each Neptune service node maintains a REDO write log for each data partition it hosts. Each log entry contains the service method name, partition ID, the request message along with an assigned *log sequence number (LSN)*. The write log consists of a committed portion and an uncommitted portion. The committed portion records those writes that are already completed while the uncommitted portion records the writes that are received but not yet completed.

Neptune assigns a static priority for each replica of a data partition. The primary is the replica with the highest priority. When a node failure is detected, for each partition that the faulty node is the primary of, the remaining replica with the highest priority is elected to become the new primary. This election algorithm is the same as the classical Bully Algorithm [41] with the exception that each replica has a priority *for each data partition* it hosts. This fail-over scheme also requires that the elected primary does not miss any write that has committed in the failed primary. To ensure that, before the primary executes a write locally, it has to wait until all other replicas have acknowledged the reception of its propagation. If a replica does not acknowledge in a timeout period, this replica is considered to fail due to our fail-stop assumption and thus this replica can only rejoin the service cluster after going through the recovery process described below.

When a node recovers after its failure, the underlying single-site service module first recovers its data into a consistent state. Then this node will enter Neptune's three-phase recovery process as follows:

Phase 1: Internal synchronization. The recovering node first synchronizes its write log with the underlying service module. This is done by using the registered CHECK callbacks to determine whether each write in the uncommitted log has been completed by the service module. The completed writes are merged into the committed portion of the write log and the uncompleted writes are reissued for execution.

Phase 2: Missing write recovery. In this phase, the recovering node announces its priority for each data partition it hosts. If the partition has a higher priority than the current primary, this node will bully the current primary into a secondary as soon as its priority announcement is heard. Then it contacts the deposed primary to recover the writes that it missed during its down time. For a partition that does not have a higher priority than the current primary, this node simply contacts the primary to recover the missed writes.

Phase 3: Operation resumption. After the missed writes are recovered, this recovering node resumes normal operations by publishing the services it hosts and accepting requests from the clients.

Note that if a recovering node has the highest priority for some data partitions, there will be no primary available for those partitions during phase two of the recovery. This temporary blocking of writes is essential to ensure that the recovering node can bring itself up-to-date before taking over as the new primary. We will present the experimental study for this behavior in Section 5.4.3. We also want to emphasize that a catastrophic failure that causes all replicas for a certain partition to fail requires special attention. No replica can successfully complete phase two recovery after such a failure because there is no pre-existing primary in the system to recover missed writes. In this case, the replica with newest version needs to be manually brought up as the primary then all other replicas can proceed the standard three-phase recovery.

Before concluding our failure recovery model, we describe a possible CHECK callback support provided by the service module. We require the Neptune server module to pass the LSN with each request to the service instance. Then the service instance fulfills the request and records this LSN on persistent storage. When the CHECK callback is invoked with an LSN during a recovery, the service module compares it with the LSN of the latest completed service access and returns appropriately. As we mentioned in Section 5.1.1, Neptune provides atomic execution through failures only if the underlying service module can ensure atomicity on single-site service accesses. Such support can ensure the service access and the recording of LSN take place as an atomic action. A transactional database or a transactional file system can be used to achieve atomicity for single-site service accesses.

5.3 Service Deployments

We have deployed three demonstration services on a Neptune-enabled Linux cluster for performance evaluation. The first service is *on-line discussion group*, which handles three types of requests for each discussion topic: viewing the list of message headers (`ViewHeaders`), viewing the content of a message (`ViewMsg`), and adding a new message (`AddMsg`). Both `ViewHeaders` and `ViewMsg` are read-only requests. The messages are maintained and displayed in a hierarchical format according to the reply-to relationships among them. The discussion group uses MySQL database to store and retrieve messages and topics.

The second service is a prototype *auction* service, which is also implemented on MySQL database. The auction service supports five types of requests: viewing the list of categories (`ViewCategories`), viewing the available items in an auction category (`ViewCategory`), viewing the information about a specific item (`ViewItem`), adding a new item for auction (`AddItem`), and bidding for an item (`BidItem`). Level three consistency with proper staleness bound and progressive version delivery is desirable for this service in order to prevent auction users from seeing declining bidding prices.

The third service in our study is a *persistent cache* service. This service supports two service methods: storing a key/data pair into the persistent cache (`CacheUpdate`) and retrieving the data for a given key (`CacheLookup`). The persistent cache uses an MD5 encoding based hashing function to map the key space into a set of buckets. Each bucket initially occupies one disk block (1024 bytes) in the persistent storage and it may acquire more blocks in the case of overflow. We use `mmap()` utilities to keep an in-memory reference to the disk data and we purge the updates and the corresponding LSN into the disk at every tenth `CacheUpdate` invocation. The LSN is used to support the CHECK callback that we discussed in Section 5.2.2. The persistent cache is most likely an internal service, which provides a scalable and reliable data store for other services. We used level two consistency for this service, which allows high throughput with intermittent false cache misses. A similar strategy was adopted in an earlier study on Web cache clustering [51].

We note that MySQL database does not have full-fledged transactional support, but its latest version supports "atomic operations", which is enough for Neptune to provide cluster-wide atomicity. On the other hand, our current persistent cache is built on a regular file system without atomic recovery support. However, we believe such a setting is sufficient for illustrative purposes.

5.4 System Evaluations

Our experimental studies are focused on performance-scalability, availability, and consistency levels of Neptune cluster services. All the evaluations in this section were conducted on a rack-mounted Linux cluster with around 30 dual 400 Mhz Pentium II nodes, each of which contains either 512 MB or 1 GB memory. Each node runs Linux 2.2.15 and has two 100 Mb/s Ethernet interfaces. The cluster is connected by a Lucent P550 Ethernet switch with 22 Gb/s backplane bandwidth. All the experiments presented in this section use up to 16 server nodes and up to 6 client nodes. MySQL 3.23.22-Beta was used as the service database.

Even though all the services rely on protocol gateways to reach end clients, the performance between protocol gateways and end clients is out of the scope of this study. Our experiments are instead focused on studying the performance between clients and services inside a Neptune service cluster.

We used synthetic workloads in all the evaluations. Two types of workloads were generated for this purpose: 1) Balanced workloads where service requests are evenly distributed among data partitions were used to measure the best case scalability. 2) Skewed workloads, in comparison, were used to measure the system performance when some particular partitions draw a disproportional number of service requests. We measured the maximum system throughput when more than 98% of client requests were successfully completed within two seconds. Our testing clients attempted to saturate the system by probing the maximum throughput. In the first phase, they doubled their request sending rate until 2% of the requests failed to complete in 2 seconds. Then they adjusted the sending rates in smaller steps and resumed probing.

The rest of this section is organized as follows. Section 5.4.1 and Section 5.4.2 present the system performance under balanced and skewed workload. Section 5.4.3 illustrates the system behavior during failure recoveries. The discussion group service is used in all the above experiments. Section 5.4.4 presents the performance of the auction and persistent cache service.

5.4.1 Scalability under Balanced Workload

We use the discussion group to study the system scalability under balanced workload. In this evaluation, we studied the performance impact when varying the replication degree, the number of service nodes, the write percentage, and consistency levels. The write percentage is the percentage of writes in all requests and it is usually small for discussion group services. However,

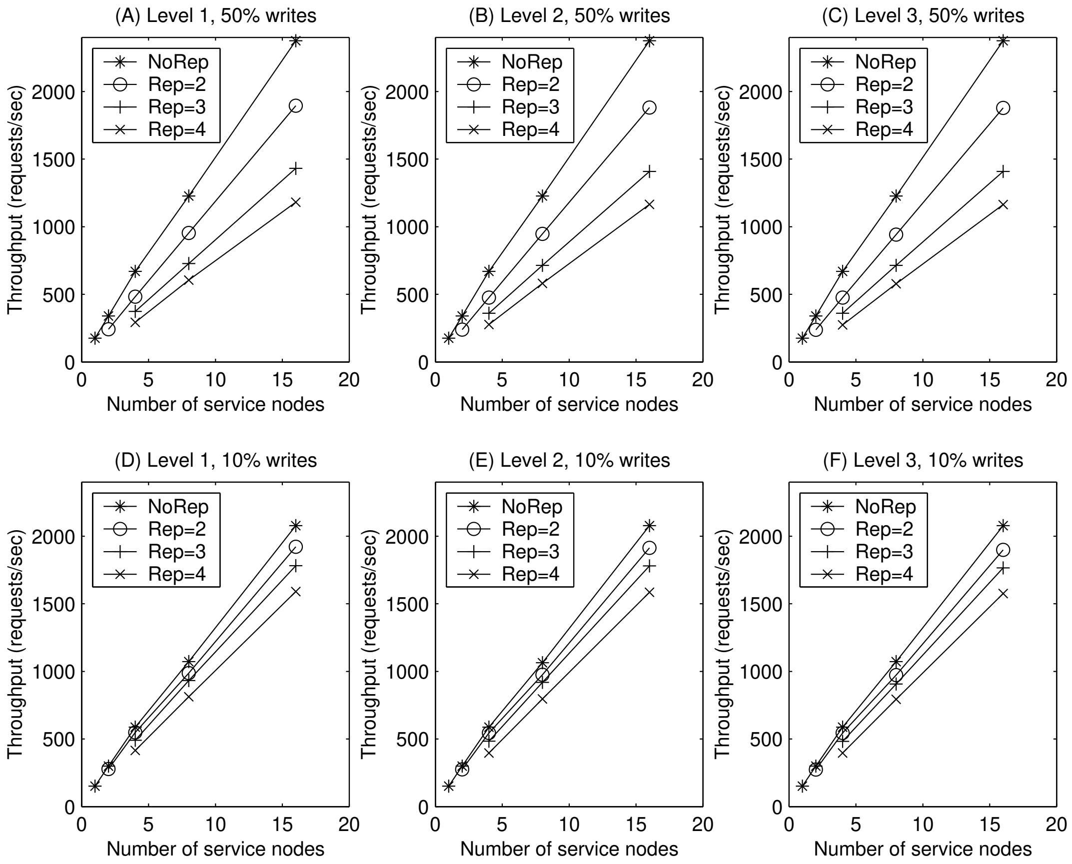

Figure 5.1: Scalability of discussion group service under balanced workload.

we are also interested in assessing the system performance under high write percentage, which allows us to predict the system behavior for services with more frequent writes. We present the results under two write percentages: 10% and 50%. In addition, we measured all three consistency levels in this study. Level one consistency requires writes to be commutative and thus, we used a variation of the original service implementation to facilitate it. For the purpose of performance comparison with other consistency levels, we kept the changes to be minimum. For level three consistency, we chose one second as the staleness bound. We also noticed that the performance of level three consistency is affected by the request rate of individual clients. This is because a higher request rate from each client means a higher chance that a read has to be forwarded to the primary node to fulfill progressive version control, which in turn restricts the system load balancing capabilities. We recognized that most Web users spend at least several seconds between consecutive requests. Thus we chose one request per second as the client request rate in this evaluation to measure the worst case impact.

The number of discussion groups in our synthetic workload was 400 times the number of service nodes. Those groups were in turn divided into 64 partitions. These partitions and their replicas were evenly distributed across service nodes. Each request was sent to a discussion group chosen according to an even distribution. The distribution of different requests (`AddMsg`, `ViewHeaders` and `ViewMsg`) was determined based on the write percentage.

Figure 5.1 shows the scalability of discussion group service with three consistency levels and two write percentages (10% and 50%). Each sub-figure illustrates the system performance under no replication (`NoRep`) and replication degrees of two, three and four. The `NoRep` performance is acquired through running a stripped down version of Neptune which does not contain any replication overhead except logging. The single node performance under no replication is 152 requests/second for 10% writes and 175 requests/second for 50% writes. We can use them as an estimation for the basic service overhead. Notice that a read is more costly than a write because `ViewHeaders` displays the message headers in a hierarchical format according to the reply-to relationships, which may invoke some expensive SQL queries.

We can draw the following conclusions based on the results in Figure 5.1: 1) When the number of service nodes increases, the throughput steadily scales across all replication degrees. 2) Service replication comes with an overhead because every write has to be executed more than once. Not surprisingly, this overhead is more prominent under higher write percentage. In general, a non-replicated service performs twice as fast as its counterpart with a replication degree of four at 50% writes. However, Section 5.4.2 shows that replicated services can outperform non-replicated services under skewed workloads due to better load balancing. 3) All three consistency levels perform very closely under balanced workload. This means level one consistency does not provide a significant performance advantage and a staleness control does not incur significant overhead either. We recognize that higher levels of consistency result in more restrictions on Neptune client module's load balancing capability. However, those restrictions inflict very little performance impact for balanced workload.

5.4.2 Impact of Workload Imbalance

This section studies the performance impact of workload imbalance. Each skewed workload in this study consists of requests that are chosen from a set of partitions according to a Zipf distribution. Each workload is also labeled with a *workload imbalance factor*, which indicates the proportion of the requests that are directed to the most popular partition. For a service with 64 partitions, a workload with an imbalance factor of $1/64$ is completely balanced. A workload with an imbalance factor of 1 is the other extremity in which all requests are directed to one single partition. Again, we use the discussion group service in this evaluation.

Figure 5.2 shows the impact of workload imbalance on services with different replication degree. The 10% write percentage, level two consistency, and eight service nodes were used in this experiment. We see that even though

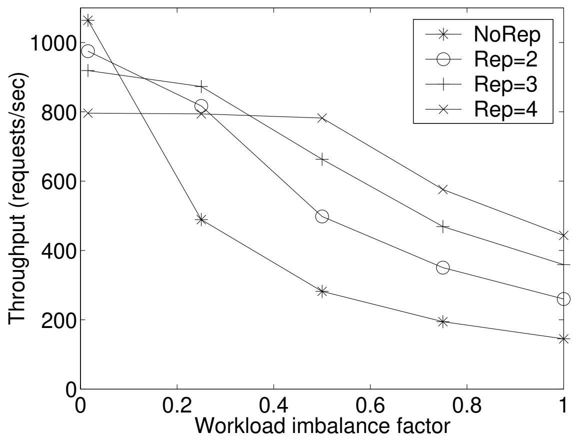

Figure 5.2: Impact of workload imbalance on the replication degrees with 8 service nodes.

service replication carries an overhead under balanced workload (imbalance factor = 1/64), replicated services can outperform non-replicated ones under skewed workload. Specifically, under the workload that directs all requests to one single partition, the service with a replication degree of four performs almost three times as fast as its non-replicated counterpart. This is because service replication provides better load-sharing by spreading hot-spots over several service nodes, which completely amortizes the overhead of extra writes in achieving the replica consistency.

We learned from Section 5.4.1 that all three consistency levels perform very closely under balanced workload. Figure 5.3 illustrates the impact of workload imbalance on different consistency levels. The 10% write percentage, a replica-

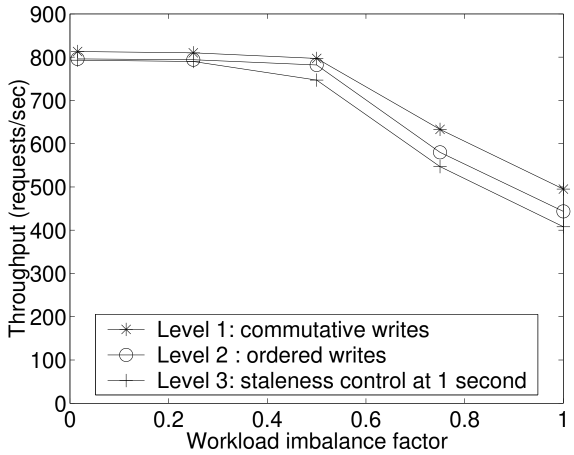

Figure 5.3: Impact of workload imbalance on consistency levels with 8 service nodes.

tion degree of four, and eight service nodes were used in this experiment. The performance difference among three consistency levels becomes slightly more prominent when the workload imbalance factor increases. Specifically under the workload that directs all requests to one single partition, level one consistency yields 12% better performance than level two consistency, which in turn performs 9% faster than level three consistency with staleness control at one second. Based on these results, we learned that: 1) The freedom of directing writes to any replica in level one consistency only yields moderate performance advantage. 2) Our staleness control scheme carries an insignificant overhead even though it appears slightly larger for skewed workload.

5.4.3 System Behavior during Failure Recoveries

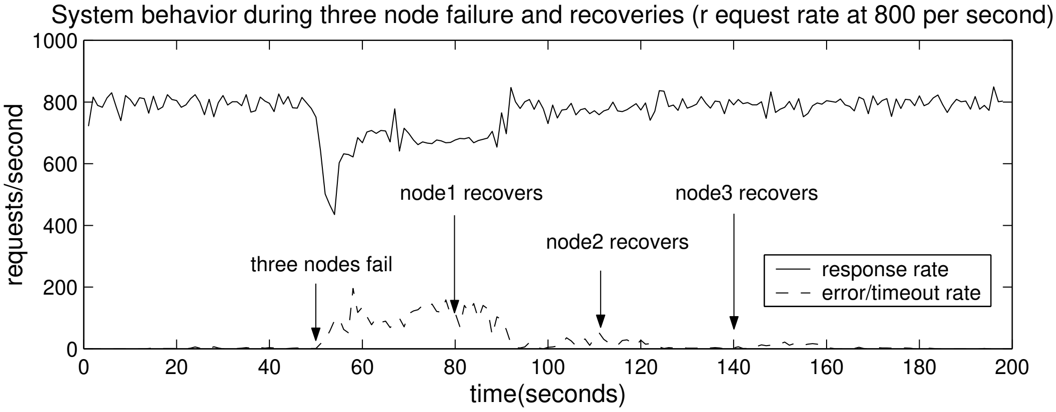

Figure 5.4: Behavior of the discussion group service during three node failure and recoveries. Eight service nodes, level two consistency, and a replication degree of four were used in this experiment.

Figure 5.4 depicts the behavior of a Neptune-enabled discussion group service during three node failures in a 200-second period. Eight service nodes, level two consistency, and a replication degree of four were used in the experiments. Three service nodes fail simultaneously at time 50. Node 1 recovers 30 seconds later. Node 2 recovers at time 110 and node 3 recovers at time 140. It is worth mentioning that a recovery may take much longer than 30 seconds in practice, especially when large data files need to be loaded over the network as part of such recovery [2]. However, we believe those variations do not affect the fundamental system behavior illustrated in this experiment. We observe that the system throughput goes down during the failure period. And we also observe a tail of errors and timeouts trailing each node recovery. This is caused by the lost of primary and the overhead of synchronizing lost updates during the recovery as discussed in Section 5.2.2. However, the service quickly stabilizes and resumes normal operations.

5.4.4 Auction and Persistent Cache

In this section, we present the performance of the Neptune-enabled auction and persistent cache service. We analyzed the data published by `eBay` about the requests they received from May 29 to June 9, 1999. Excluding the requests for embedded images, we estimate that about 10% of the requests were for bidding, and 3% were for adding new items. More information about this analysis can be found in our earlier study on dynamic Web caching [86]. We used the above statistical information in designing our test workload. We chose the number of auction categories to be 400 times the number of service nodes. Those categories were in turn divided into 64 partitions. Each request was made for an auction category selected from a population according to an even distribution. We chose level three consistency with staleness control at one second in this experiment. This consistency level fits the auction users' needs to acquire the latest information.

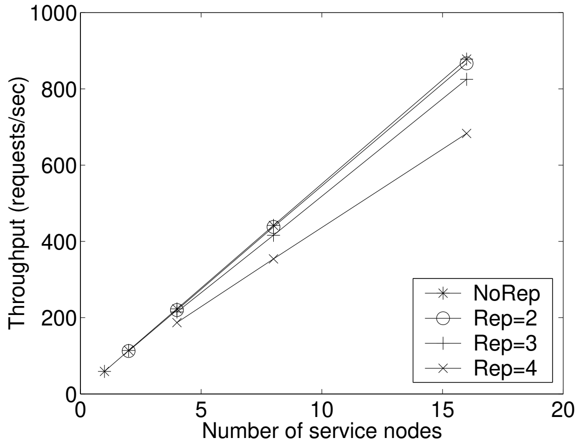

Figure 5.5: Performance of auction on Neptune.

Figure 5.5 shows the performance of a Neptune-enabled auction service. Its absolute performance is slower than that of the discussion group because the auction service involves extra overhead in authentication and user account maintenance. In general the results match the performance of the discussion group with 10% writes in Section 5.4.1. However, we do observe that the replication overhead is smaller for the auction service. The reason is that the tradeoff between the read load sharing and extra write overhead for service replication depends on the cost ratio between a read and a write. For the auction service most writes are bidding requests which incur very little overhead by themselves.

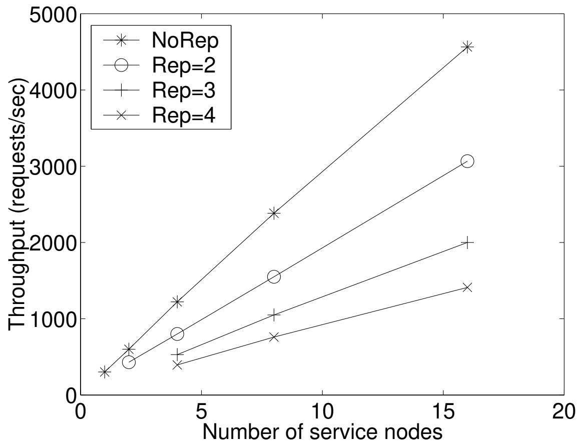

Figure 5.6: Performance of persistent cache on Neptune.

Figure 5.6 illustrates the performance of the persistent cache service. Level two consistency and 10% write percentage were used in the experiment. The results show large replication overhead caused by extra writes. This is because `CacheUpdate` may cause costly disk accesses while `CacheLookup` can usually be fulfilled with in-memory data.

5.5 Related Work

Replica consistency. Replication of persistent data is crucial to achieving high availability. The earlier analysis by Gray et al. shows that the synchronous replication based on eager update propagations leads to high deadlock rates [44]. A recent study by Anderson et al. confirms this using simulations [8]. The asynchronous replication based on lazy propagations has been used in Bayou [66]. Adya et al. have studied lazy replication with a type of lazy consistency in which server data is replicated in a client cache [4]. The serializability for lazy propagations with the primary-copy method is further studied by a few other research groups [8, 24] and they address causal dependence when accessing multiple objects. The most recent work by Yu and Vahdat provides a tunable framework to exploit the tradeoff among availability, consistency, and performance [81]. Neptune's replica consistency maintenance is close to the previous work on asynchronous replication in the use of logging/REDO facilities. Neptune differs from them in providing flexible replica consistency for cluster-based network services with the emphasis on performance scalability and fail-over support. In particular, Neptune employs a multi-level replica consistency model with data staleness control at its highest level.

Persistent data management for clustered services. The Porcupine project has developed a scalable cluster-based email service and its replication model is intended for services with only commutative writes [67]. Recently the DDS project has addressed replication of persistent data with a carefully built data management layer that encapsulates scalable replica consistency and fail-over support [46]. While this approach is demonstrated for services with simple processing logic like distributed hash tables, constructing such a data management layer could be difficult for applications with complex data management logic, including many database applications. Our work complements these studies by providing replication support for clustering stand-alone service modules with the capability of accommodating various underlying data management mechanisms. In addition, Neptune is novel in providing multiple levels of replica consistency support to exploit application-level service semantics and the performance tradeoff.

Replication support in database and transaction processing systems. Commercial database systems from Oracle, Sybase and IBM support lazy updates for data replication and they rely on user-specified rules to resolve conflicts. Neptune differs from those systems by taking advantage of the inherently partitionable property of most Internet services. As a result, Neptune's consistency model is built with respect to single data partition, which enables Neptune to deliver highly consistent views to clients without losing performance and availability. Nevertheless, Neptune's design on communication schemes and failure recovery model benefits greatly from previous work on transactional RPC and transaction processing systems [45, 77].

5.6 Concluding Remarks

Our work is targeted at aggregating and replicating partitionable network services in a cluster environment. The main contributions are the development of a scalable and highly available clustering infrastructure with replication support and the proposal of a weak replica consistency model with staleness control at its highest level. In addition, our clustering and replication infrastructure is capable of supporting application-level services built upon a heterogeneous set of databases, file systems, or other data management systems.

Service replication increases availability. However, it may compromise the throughput of applications with frequent writes due to the consistency management overhead. Our experiments show that Neptune's highest consistency scheme with staleness control can still deliver scalable performance with insignificant overhead. This advantage is gained by targeting at partitionable services and focusing on the consistency for a single data partition. In terms of service guarantees, our level three consistency ensures that client accesses are serviced progressively within a specified soft staleness bound, which is sufficient for many Internet services. In that sense, a strong consistency model, which allows clients to always get the latest version with the cost of degraded throughput, may not be necessary in many cases. Nevertheless, further investigation is needed for the incorporation of stronger consistency models.

Chapter 6

Conclusion and Future Work

Building large-scale network services with the ever-increasing demand on scalability and availability is a challenging task. This dissertation investigates techniques in building a middleware system, called *Neptune*, that provides clustering support for scalable network services. In particular, Neptune focuses on three specific aspects of service clustering support: the overall clustering architecture with load balancing support, a quality-aware resource management framework, and service replication support. Neptune has been implemented on Linux and Solaris clusters and a number of applications have been successfully deployed on Neptune platforms, including a large-scale document search engine.

Neptune employs a loosely-connected and functionally-symmetrical architecture in constructing the service cluster, which allows the service infrastructure to operate smoothly in the presence of transient failures and through service evolution. In addition, Neptune provides flexible interfaces that enable existing applications to be easily deployed in a Neptune service cluster, even for binary applications without recompilation [7]. Generally speaking, providing standard system components to achieve scalability and availability tends to decrease the flexibility of service construction. Neptune demonstrates that it is possible to achieve these goals by targeting partitionable network services. As part of the clustering architecture, this dissertation investigates cluster load balancing techniques with the focus on fine-grain services. Based on simulation and experimental studies, this dissertation finds that the random polling policy with small poll sizes are well-suited for fine-grain network services. And discarding slow-responding polls can further improve system performance.

This dissertation also presents the design and implementation of an integrated quality-aware resource management framework for cluster-based services. Although cluster-based network services have been widely deployed, we have seen limited research in the literature on comprehensive resource management with service differentiation support. This dissertation study is focused on clustered services with dynamic service fulfillment or content generation. In particular, it addresses the inadequacy of the previous studies and complements them in the following three aspects. First, it allows quality-aware resource management objectives which combine the individual service response times with the overall system resource utilization efficiency. Secondly, it employs a functionally symmetrical architecture that does not rely on any centralized components for high scalability and availability. Thirdly, it employs an adaptive scheduling policy that achieves efficient resource utilization at a wide range of load levels. This study on cluster resource management is focused on a single service tier. Our future work is to support comprehensive service quality guarantees for aggregated network services. Resource allocation across different service domains, however, requires service nodes capable of deploying components from multiple service domains. This is difficult for data intensive services that involve large data volume and long service warm-up time.

Finally, this dissertation studies service replication support for cluster-based network services. The main contributions are the development of a scalable and highly available clustering infrastructure with replication support and the proposal of a weak replica consistency model with staleness control at its highest level. In addition, our clustering and replication infrastructure is capable of supporting application-level services built upon a variety of underlying data management systems. Service replication increases availability, however, it may compromise the throughput of applications with frequent writes due to the consistency management overhead. Our experiments show that Neptune's highest consistency scheme with staleness control can still deliver scalable performance with insignificant overhead. This advantage is gained by targeting at partitionable services and focusing on the consistency for a single data partition. In terms of service guarantees, our level three consistency ensures that client accesses are serviced progressively within a specified soft staleness bound, which is sufficient for many Internet services. In that sense, a strong consistency model, which allows clients to always get the latest version with the cost of degraded throughput, may not be necessary in many cases. Nevertheless, further investigation is needed for the incorporation of stronger consistency models.

Neptune assumes all service components inside a cluster belong to a single administrative domain and it does not explicitly support security and access control. When third-party service components are deployed inside a service cluster, however, proper access control and resource accounting are required. These issues remain to be solved.

Bibliography

[1] T. F. Abdelzaher and N. Bhatti. Web Server QoS Management by Adaptive Content Delivery. In *International Workshop on Quality of Service*, London, UK, June 1999.

[2] A. Acharya. Google Inc., Personal Communication, 2002.

[3] ADL. Alexandria digital library project. http://www.alexandria.ucsb.edu.

[4] A. Adya and B. Liskov. Lazy Consistency Using Loosely Synchronized Clocks. In *Proc. of the ACM Symposium on Principles of Distributed Computing*, pages 73–82, Santa Barbara, CA, August 1997.

[5] D. Agrawal, A. El Abbadi, and R. C. Steinke. Epidemic Algorithms in Replicated Databases. In *Proc. of the 16th Symposium on Principles of Database Systems*, pages 161–172, Montreal, Canada, May 1997.

[6] J. Almeida, M. Dabu, A. Manikutty, and P. Cao. Providing Differentiated Levels of Service in Web Content Hosting. In *Proc. of SIGMETRICS Workshop on Internet Server Performance*, Madison, WI, June 1998.

[7] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic Local Alignment Search Tool. *Journal of Molecular Biology*, 215:403–410, 1990.

[8] T. Anderson, Y. Breitbart, H. F. Korth, and A. Wool. Replication, Consistency, and Practicality: Are These Mutually Exclusive? In *Proc. of 1998 ACM SIGMOD Intl. Conf. on Management of Data*, pages 484–495, Seattle, WA, June 1998.

[9] D. Andresen, T. Yang, V. Holmedahl, and O. Ibarra. SWEB: Towards a Scalable WWW Server on MultiComputers. In *Proc. of the 10th IEEE Intl. Parallel Processing Symposium*, pages 850–856, Honolulu, HI, April 1996.

[10] M. Aron, P. Druschel, and W. Zwaenepoel. Cluster Reserves: A Mechanism for Resource Management in Cluster-based Network Servers. In *Proc. of the 2000 ACM SIGMETRICS Intl. Conf. on Measurement and*

Modeling of Computer Systems, pages 90–101, Santa Clara, CA, June 2000.

[11] M. Aron, D. Sanders, P. Druschel, and W. Zwaenepoel. Scalable Content-aware Request Distribution in Cluster-based Network Services. In *Proc. of the 2000 USENIX Annual Technical Conf.*, San Diego, CA, June 2000.

[12] ArrowPoint. Web Switching White Papers. http://www.arrowpoint.com/solu-tions/white_papers/.

[13] AskJeeves. Ask jeeves search. http://www.ask.com.

[14] G. Banga, P. Druschel, and J. C. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In *Proc. of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, pages 45–58, New Orleans, LA, February 1999.

[15] A. Barak, S. Guday, and R. G. Wheeler. *The MOSIX Distributed Operating System: Load Balancing for UNIX*, volume 672 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[16] D. A. Benson, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, B. A. Rapp, and D. L. Wheeler. GenBank. *Nucleic Acids Research*, 30(1):17–20, 2002.

[17] N. Bhatti and R. Friedrich. Web Server Support for Tiered Services. *IEEE Network*, 13(5):64–71, September 1999.

[18] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An Architecture for Differentiated Services. *Internet Draft, IETF Diffserv Working Group*, August 1998.

[19] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Proc. of ACM Symposium on Principles & Practice of Parallel Programming*, pages 207–216, Santa Barbara, CA, July 1995.

[20] J. Bruno, E. Gabber, B. Ozden, and A. Silberschatz. The Eclipse Operating System: Providing Quality of Service via Reservation Domains. In *Proc. of USENIX Annual Technical Conf.*, pages 235–246, Orleans, LA, June 1998.

[21] E. V. Carrera and R. Bianchini. Efficiency vs. Portability in Cluster-based Network Servers. In *Proc. of the 8th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 113–122, Snowbird, UT, June 2001.

[22] S. Chandra, C. S. Ellis, and A. Vahdat. Differentiated Multimedia Web Services Using Quality Aware Transcoding. In *Proc. of IEEE INFO-COM'2000*, Tel-Aviv, Israel, March 2000.

[23] J. S. Chase, D. C. Anderson, P. N. Thakar, and A. M. Vahdat. Managing Energy and Server Resources in Hosting Centers. In *Proc. of the 18th ACM Symposium on Operating Systems Principles*, Banff, Canada, October 2001.

[24] P. Chundi, D. J. Rosenkrantz, and S. S. Ravi. Deferred Updates and Data Placement in Distributed Databases. In *Proc. of the 12th Intl. Conf. on Data Engineering*, pages 469–476, New Orleans, Louisiana, February 1996.

[25] COM. Component Object Model. http://www.microsoft.com/com.

[26] CORBA. Common Object Request Broker Architecture. http://www.corba.org.

[27] M. E. Crovella and A. Bestavros. Self-similarity in World Wide Web Traffic: Evidence and Possible Causes. *IEEE/ACM Transactions on Networking*, 5(6):835–846, 1997.

[28] S. Dandamudi. Performance Impact of Scheduling Discipline on Adaptive Load Sharing in Homogeneneous Distributed Systems. In *Proc. of Intl. Conf. on Distributed Computer Systems*, pages 484–492, Vancouver, BC, May 1995.

[29] C. Dovrolis, D. Stiliadis, and P. Ramanathan. Proportional Differentiated Services: Delay Differentiation and Packet Scheduling. In *Proc. of ACM SIGCOMM'99*, pages 109–120, Cambridge, MA, August 1999.

[30] P. Druschel and G. Banga. Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems. In *Proc. of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, Seattle, WA, October 1996.

[31] D. L. Eager, E. D. Lazowska, and J. Zahorjan. A Comparison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing. *Performance Evaluation*, 6:53–68, 1986.

[32] D. L. Eager, E. D. Lazowska, and J. Zahorjan. Adaptive Load Sharing in Homogeneous Distributed Systems. *IEEE Trans. on Software Engineering*, 12(5):662–675, May 1986.

[33] eBay. ebay online auctions. http://www.ebay.com.

[34] J. Postel Ed. Transmission Control Protocol Specification. SRI International, Menlo Park, CA, September 1981. RFC-793.

[35] A. Feldmann. Characteristics of TCP Connection Arrivals. Technical report, AT&T Labs Research, 1998.

[36] D. Ferrari. A Study of Load Indices for Load Balancing Schemes. Technical Report CSD-85-262, EECS Department, UC Berkeley, October 1985.

[37] S. Floyd and V. Jacobson. The Synchronization of Periodic Routing Messages. In *Proc. of ACM SIGCOMM'93*, pages 33–44, San Francisco, CA, September 1993.

[38] Foundry. Serveriron server load balancing switch. http://www.foundrynet.com/serverironspec.html.

[39] A. Fox and E. A. Brewer. Harvest, Yield, and Scalable Tolerant Systems. In *Proc. of HotOS-VII*, Rio Rico, AZ, March 1999.

[40] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-Based Scalable Network Services. In *Proc. of the 16th ACM Sym-*

posium on Operating System Principles, pages 78–91, Saint Malo, October 1997.

[41] H. Garcia-Molina. Elections in a Distributed Computing System. *IEEE Trans. on Computers*, 31:48–59, January 1982.

[42] Google. Google search. http://www.google.com.

[43] K. K. Goswami, M. Devarakonda, and R. K. Iyer. Prediction-Based Dynamic Load-Sharing Heuristics. *IEEE Trans. on Parallel and Distributed Systems*, 4(6):638–648, June 1993.

[44] J. Gray, P. Helland, P. O'Neil, , and D. Shasha. The Dangers of Replication and a Solution. In *Proc. of 1996 ACM SIGMOD Intl. Conf. on Management of Data*, pages 173–182, Montreal, Canada, June 1996.

[45] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publishers, San Francisco, California, 1993.

[46] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, Distributed Data Structures for Internet Service Construction. In *Proc. of the 4th USENIX Symposium on Operating Systems Design and Implementation*, San Diego, CA, October 2000.

[47] S. D. Gribble, M. Welsh, E. A. Brewer, and D. Culler. The MultiSpace: An Evolutionary Platform for Infrastructural Services. In *Proc. of the USENIX Annual Technical Conf.*, Monterey, CA, June 1999.

[48] MSN Groups. http://groups.msn.com.

[49] M. Harchol-Balter and A. B. Downey. Exploiting Process Lifetime Distributions for Dynamic Load Balancing. *ACM Transactions on Computer Systems*, 15(3):253–285, 1997.

[50] J. R. Haritsa, M. J. Carey, and M. Livny. Value-Based Scheduling in Real-Time Database Systems. *VLDB Journal*, 2:117–152, 1993.

[51] V. Holmedahl, B. Smith, and T. Yang. Cooperative Caching of Dynamic Content on a Distributed Web Server. In *Proc. of the 7th IEEE Symposium on High Performance Distributed Computing*, Chicago, IL, July 1998.

[52] J. Huang, J. Stankovic, D. Towsley, and K. Ramamritham. Experimental Evaluation of Real-Time Transaction Processing. In *Proc. of the Tenth IEEE Real-Time System Symposium*, pages 144–153, Santa Monica, CA, 1989.

[53] G. D. H. Hunt, G. S. Goldszmidt, R. P. King, and R. Mukherjee. Network Dispatcher: A Connection Router for Scalable Internet Services. In *Proc. of the 7th Intl. World Wide Web Conf.*, Brisbane, Australia, April 1998.

[54] Java. Java Platform. http://java.sun.com.

[55] M. B. Jones, D. Rosu, and M.-C. Rosu. CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities. In *Proc. of the 16th ACM Symposium on Operating Systems Principles*, pages 198–211, Saint-Malo, France, October 1997.

[56] L. Kleinrock. *Queueing Systems*, volume I: Theory. Wiley, New York, 1975.

[57] T. Kunz. The influence of Different Workload Descriptions on a Heuristic Load Balancing Scheme. *IEEE Trans. on Software Engineering*, 17(7):725–730, July 1991.

[58] J. Kurose. Open Issues and Challenges in Providing Quality of Service Guarantees in High-Speed Networks. *ACM Computer Communication Review*, 23(1):6–15, 1993.

[59] K. Li and S. Jamin. A Measurement-Based Admission-Controlled Web Server. In *Proc. of IEEE INFOCOM'2000*, pages 651–659, Tel-Aviv, Israel, March 2000.

[60] Z. Liu, M. S. Squillante, and J. L. Wolf. On Maximizing Service-Level-Agreement Profits. In *Proc. of 3rd ACM Conference on Electronic Commerce*, pages 14–17, Tampa, FL, October 2001.

[61] M. Mitzenmacher. On the Analysis of Randomized Load Balancing Schemes. In *Proc. of the 9th ACM Symposium on Parallel Algorithms and Architectures*, pages 292–301, Newport, RI, June 1997.

[62] J. Mogul and K. K. Ramakrishnan. Eliminating Receive Livelock in an Interrupt-driven Kernel. In *Proc. of USENIX Annual Technical Conf.*, San Diego, CA, January 1996.

[63] S. Nagy and A. Bestavros. Admission Control for Soft-Deadline Transactions in ACCORD. In *Proc. of IEEE Real-Time Technology and Applications Symposium*, pages 160–165, Montreal, Canada, June 1997.

[64] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-Aware Request Distribution in Cluster-based

Network Servers. In *Proc. of the ACM 8th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 205–216, San Jose, CA, October 1998.

[65] R. Pandey, J. F. Barnes, and R. Olsson. Supporting Quality of Service in HTTP Servers. In *Proc. of 17th ACM Symposium on Principles of Distributed Computing*, pages 247–256, Puerto Vallarta, Mexico, June 1998.

[66] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible Update Propagation for Weakly Consistent Replication. In *Proc. of the 16th ACM Symposium on Operating Systems Principles*, pages 288–301, Saint Malo, France, October 1997.

[67] Y. Saito, B. N. Bershad, and H. M. Levy. Manageability, Availability, and Performance in Porcupine: a Highly Scalable, Cluster-based Mail Service. In *Proc. of the 17th ACM Symposium on Operating Systems Principles*, pages 1–15, Charleston, SC, December 1999.

[68] H. V. Shah, D. B. Minturn, A. Foong, G. L. McAlphine, R. S. Madukkarumukumana, and G. J. Regnier. CSP: A Novel System Architecture for Scalable Internet and Communication Services. In *Proc. of the 3rd*

USENIX Symposium on Internet Technologies and Systems, pages 61–72, San Francisco, CA, March 2001.

[69] K. Shen, H. Tang, T. Yang, and L. Chu. Integrated Resource Management for Cluster-based Internet Services. In *Proc. of the 5th USENIX Symposium on Operating Systems Design and Implementation (To appear)*, Boston, MA, December 2002.

[70] K. Shen, T. Yang, and L. Chu. Cluster Load Balancing for Fine-grain Network Services. In *Proc. of International Parallel & Distributed Processing Symposium*, Fort Lauderdale, FL, April 2002.

[71] K. Shen, T. Yang, L. Chu, J. L. Holliday, D. A. Kuschner, and H. Zhu. Neptune: Scalable Replication Management and Programming Support for Cluster-based Network Services. In *Proc. of the 3rd USENIX Symposium on Internet Technologies and Systems*, pages 197–208, San Francisco, CA, March 2001.

[72] A. Singhai, S.-B. Lim, and S. R. Radia. The SunSCALR Framework for Internet Servers. In *Proc. of the 28th Intl. Symposium on Fault-Tolerant Computing*, Munich, Germany, June 1998.

[73] D. C. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole. A Feedback-driven Proportion Allocator for Real-Rate Scheduling. In *Proc. of 3rd USENIX Operating Systems Design and Implementation Symposium*, New Orleans, LA, February 1999.

[74] I. Stoica and H. Zhang. LIRA: An Approach for Service Differentiation in the Internet. In *Proc. of Nossdav*, June 1998.

[75] D. G. Sullivan and M. I. Seltzer. Isolation with Flexibility: A Resource Management Framework for Central Servers. In *Proc. of the 2000 USENIX Annual Technical Conf.*, San Diego, CA, June 2000.

[76] Teoma. Teoma search. http://www.teoma.com.

[77] Tuxedo. WebLogic and Tuxedo Transaction Application Server White Papers. http://www.bea.com/products/tuxedo/papers.html.

[78] W. Vogels, D. Dumitriu, K. Birman, R. Gamache, M. Massa, R. Short, J. Vert, J. Barrera, and J. Gray. The Design and Architecture of the Microsoft Cluster Service - A Practical Approach to High-Availability and Scalability. In *Proc. of the 28th Intl. Symposium on Fault-Tolerant Computing*, Munich, Germany, June 1998.

[79] T. Voigt, R. Tewari, D. Freimuth, and A. Mehra. Kernel Mechanisms for Service Differentiation in Overloaded Web Servers. In *Proc. of USENIX Annual Technical Conf.*, Boston, MA, June 2001.

[80] C. A. Waldspurger and W. E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proc. of USENIX Operating Systems Design and Implementation Symposium*, pages 1–11, Monterey, CA, November 1994.

[81] H. Yu and A. Vahdat. Design and Evaluation of a Continuous Consistency Model for Replicated Services. In *Proc. of the 4th USENIX Symposium on Operating Systems Design and Implementation*, San Diego, CA, October 2000.

[82] S. Zhou. An Experimental Assessment of Resource Queue Lengths as Load Indices. In *Proc. of the Winter USENIX Technical Conf.*, pages 73–82, Washington, DC, January 1987.

[83] S. Zhou. A Trace-Driven Simulation Study of Dynamic Load Balancing. *IEEE Trans. on Software Engineering*, 14(9):1327–1341, September 1988.

[84] H. Zhu, B. Smith, and T. Yang. Scheduling Optimization for Resource-Intensive Web Requests on Server Clusters. In *Proc. of the 11th ACM Symposium on Parallel Algorithms and Architectures*, pages 13–22, Saint-Malo, France, June 1999.

[85] H. Zhu, H. Tang, and T. Yang. Demand-driven Service Differentiation for Cluster-based Network Servers. In *Proc. of IEEE INFOCOM'2001*, Anchorage, AK, April 2001.

[86] H. Zhu and T. Yang. Class-based Cache Management for Dynamic Web Contents. In *Proc. of IEEE INFOCOM'2001*, Anchorage, AK, April 2001.