Clustering Support and Replication Management for Scalable Network Services

Kai Shen, Tao Yang, and Lingkun Chu

Abstract— **The ubiquity of the Internet and various intranets has brought about widespread availability of online services and applications accessible through the network. Cluster-based network services has been rapidly emerging due to their cost-effectiveness in achieving high availability and incremental scalability. This paper presents the design and implementation of the *Neptune* middleware system that provides clustering support and replication management for scalable network services. Neptune employs a loosely-connected and functionally-symmetric clustering architecture to achieve high scalability and robustness. It shields the clustering complexities from application developers through simple programming interfaces. In addition, Neptune provides replication management with flexible replication consistency support at the clustering middleware level. Such support can be easily applied to a large number of applications with different underlying data management mechanisms or service semantics. The system has been implemented on Linux and Solaris clusters, where a number of applications have been successfully deployed. Our evaluations demonstrate the system performance and smooth failure recovery achieved by proposed techniques.**

Index Terms— **Network services, programming support, replication management, failure recovery, load balancing.**

I. Introduction

Recent years have seen widespread availability of online services and applications accessible through the network. Examples include document search engines [6], [19], online digital libraries [1], data mining on scientific datasets [8], online discussion forums [25], and electronic commerce [15]. Cluster-based network services have become particularly popular due to their cost-effectiveness in achieving high availability and incremental scalability, especially when the system experiences high growth in service evolution and user demands. Within a large-scale complex service cluster, service components are usually partitioned, replicated, and aggregated to fulfill external requests. Despite its importance, service clustering to achieve high scalability, availability, and manageability remains a challenging task for service designers. And this is especially true for services with frequently updated persistent data. This paper recognizes the importance of service clustering support and its challenges. 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 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 paper examines the design and implementation of the *Neptune* middleware system, which provides clustering support and replication management for scalable network services. The main design goal of the Neptune clustering middleware is to support a simple, flexible, yet efficient model in aggregating and replicating network service components. Neptune employs a loosely-connected and functionally-symmetric clustering architecture to achieve high scalability and robustness. This architecture allows Neptune service infrastructure to operate smoothly in the presence of transient failures and service evolutions. Additionally, Neptune provides simple programming interfaces to shield application developers from clustering complexities.

Replication of persistent data is crucial to load sharing and achieving high availability. However, it introduces the complexity of consistency management when service requests trigger frequent updates on the persistent data. Though large-scale network services often do not require strong ACID consistencies [9], [21], it is important to maintain *eventual consistency* [17] and prevent the loss of any accepted updates. Besides, users of time-critical services (*e.g.*, auctions) demand certain guarantees to guard against accessing stale data. This paper investigates techniques in building a replication management framework as part of the Neptune clustering middleware. This work is closely related to a large body of previous research in network service clustering and data replication. In particular, recent projects have developed scalable replication support for specific cluster-based network services, including distributed hashtable [22] and the email service [28]. The key contribution of our work is the support of replication management at the clustering middleware level. Such support can be easily applied to a large number of applications with different underlying data management mechanisms or service semantics.

The rest of this paper is organized as follows. Section II presents the Neptune clustering architecture. Section III describes Neptune's multi-level service replication support. Section IV discusses the system implementation and service deployments. Section V presents the performance evaluation. Section VI discusses the related work and Section VII concludes this paper.

This work was supported in part by NSF grants ACIR-0082666, ACIR-0086061, ACIR-0234346, and CCR-0306473.

K. Shen is with the Department of Computer Science at the University of Rochester, Rochester, NY 14627. E-mail: kshen@cs.rochester.edu

T. Yang is with the Department of Computer Science at the University of California, Santa Barbara, CA 93106, and with Ask Jeeves. E-mail: tyang@cs.ucsb.edu

L. Chu is with the Department of Computer Science at the University of California, Santa Barbara, CA 93106. E-mail: lkchu@cs.ucsb.edu

II. Neptune Clustering Architecture

Generally speaking, providing standard system components at the middleware level to support service clustering and replication tends to decrease the flexibility of service construction. Neptune demonstrates that it is possible to achieve these goals by taking advantage of the following characteristics existing in many network 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 such examples. With these characteristics in mind, Neptune targets *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.

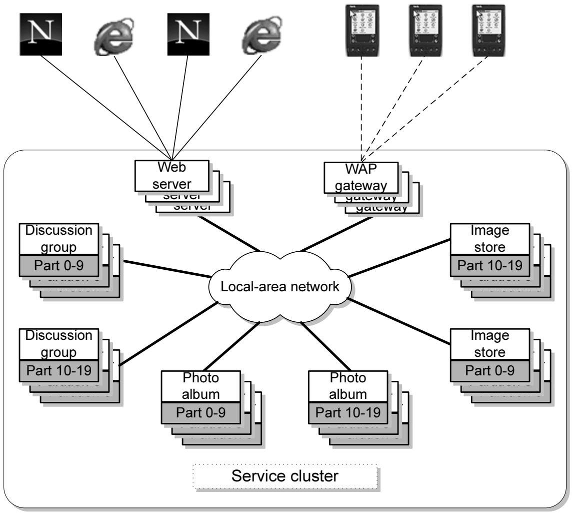


Fig. 1. Architecture for a discussion group and a photo album service.

Figure 1 illustrates the service cluster architecture for an online discussion group and a photo album service, similar to the MSN Groups service [25]. The discussion group service allows a group of users to post, read, and comment on messages. The photo album service provides a shared repository for a group of users to store, view, and edit photo images. The service cluster delivers these services to wide-area browsers and wireless clients through Web servers and WAP gateways. Note that the image store is an internal service in that it is only invoked by the photo album service components to fulfill external photo album service requests. In this example, all persistent service data is divided into twenty partitions according to user groups. All the components (including protocol gateways) are replicated in the service cluster.

Previous work has recognized the importance of providing software infrastructures for cluster-based network services [17], [32]. Most of these systems rely on central components to maintain the server runtime workload and service availability information. In comparison, Neptune employs a loosely-connected and functionally-symmetric architecture in constructing the service cluster. This architecture allows Neptune service infrastructure to operate smoothly in the presence of transient failures and service evolutions. The rest of this section examines the overall Neptune clustering architecture, its load balancing support, and programming interfaces.

A. Overall Clustering Architecture

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 symmetric* approach in constructing the server cluster. Every node in the cluster contains the same Neptune components and is functionally equal to each other. Each cluster node can elect to host service components with certain data partitions and it can also access service components hosted at other nodes in the cluster. Within a Neptune cluster, all the nodes are *loosely connected* through a publish/subscribe channel. This channel can be implemented using IP multicast or through a highly available well-known central directory.

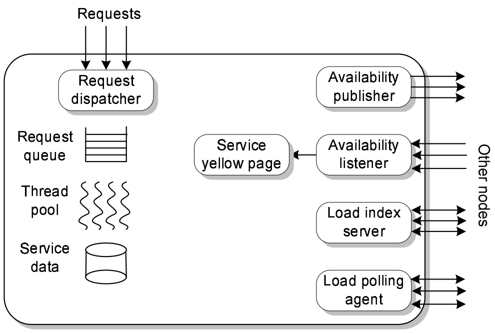


Fig. 2. Component architecture of a Neptune node.

Figure 2 illustrates the architecture of a Neptune node. Since Neptune service clusters are functionally symmetric, 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 instance 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 component, 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 components 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. For the load balancing purpose, each node can poll the load indexes of other nodes through a local *load polling agent*. The polls are responded by *load index server*s at the polled servers. Neptune employs a random polling-based load balancing policy, described in the next section. The request dispatcher, request queue, thread/process pool, availability publisher, and load index server can be considered as service-side components in a service invocation, and they are in aggregate called *Neptune service module*. Similarly, the availability listener, service yellow page, and the load polling agent are together called *Neptune client module*.

Basically, each service access is initiated at the requesting node with a service name, a data partition ID, a service method name, and a read/write access mode[1]. 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 access request, the Neptune server module in the chosen node dispatches a service instance to fulfill the service access. In the case of service replication, further request propagations may occur due to consistency management. Details on this will be discussed in Section III.

In a Neptune-enabled service cluster, the application-level service programmer only needs to implement the stand-alone application components. The aggregation of multiple data partitions or application components as well as data replication are supported transparently by Neptune clustering modules. For the service cluster shown in Figure 1, each photo album service instance locates and accesses an image store service instance through the local Neptune client module. In addition, each gateway node relies on its own Neptune client module to export the discussion group and photo album services to external users. Overall, Neptune's loosely-connected and functionally-symmetric architecture provides the following advantages:

- *Scalability* and *robustness*. The functionally-symmetric architecture removes scaling bottlenecks from the system. It also exhibits strong robustness in the face of random or targeted failures because a failure of one node is no more disastrous than that of any other.
- *Manageability*. Neptune allows service infrastructure to operate smoothly through service evolutions. For instance, new nodes can be added into the cluster by simply announcing locally hosted service components, data partitions, and the access interfaces. Therefore, the process of increasing system capacity or upgrading service features can be completed transparently to the ongoing operations in the system.

B. Cluster Load Balancing

Neptune employs a random polling-based load balancing policy for the cluster-wide request distribution. For every service access, the service client poll several randomly selected servers for load information and then directs the service access to the server responding with lightest load. The load index we use is the total number of active and queued service accesses at each server. Prior studies have suggested the resource queue lengths can be an excellent predictor of service response time [16], [34]. The random polling policy fits seamlessly into Neptune's functionally symmetrical architecture because it does not require any centralized components. The load balancing is conducted by polling agents and load index servers, both of which exist at each node. 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 substantial additional improvement [24]. By limiting the poll size, we can ensure the polling overhead for each service access does not increase with the growth of the system size, which is important for the system scalability.

In practice, Neptune's cluster load balancing support is based on a random polling policy with a poll size of three. On top of the basic random polling policy, we also made an enhancement by discarding slow-responding polls. Through a ping-pong test on two idle machines in a Linux cluster connected by a switched 100Mbps Ethernet, we measured that a UDP-based polling roundtrip cost is around 290μs. However, it may take much longer than that for a busy server to respond to a polling request. Such polling delays can affect the service response time and this impact is more severe for fine-grain services with small service processing time. With this in mind, we enhanced the basic polling policy by discarding polls not responded within 10ms, which is the smallest timeout granularity supported by the `select` system call in Linux. Intuitively, this results in a tradeoff between spending less time waiting for polls and acquiring more polled information. In addition, slow-responding polls deliver inaccurate load information due to long delays. Discarding those slow-responding polls can avoid using such stale load information, serving as an additional advantage. This again tends to be more significant for fine-grain services.

C. Neptune Programming Interfaces

Neptune supports two communication schemes between clients and service instances inside the cluster: a request/response scheme and a stream-based scheme. In the request/response scheme, the client and the service 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. Neptune only supports stream-based communication for read-only service accesses due to the difficulty of replicating and logging streams. Below we briefly describe the interfaces between Neptune and service modules at both the client and the service sides for the request/response communication scheme. The

[1] 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.

complete programming interface can be found in the Neptune programming guide [12].

- 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 for a client session. It maintains the information related to each client session and we will discuss it further in Section III-B. The meanings of other parameters are straightforward.
- At the service side, all the service method implementations need to be registered at the service deployment phase. This allows the Neptune service module to invoke the corresponding service instance when a request is received.

III. Replication Management

In this section, we investigate techniques in providing scalable replication support for cluster-based network services. The goal of this work is to provide simple, flexible, and yet 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 replication 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. In order to support applications with different service semantics and consistency requirements, Neptune employs a multi-level replication consistency scheme. The rest of this section describes the underlying assumptions of our work, Neptune's replication consistency support, and its failure recovery model.

A. Assumptions for Replication Management

We assume all hardware and software system components follow the fail-stop failure model and network partitions do not occur inside the service cluster. Following this principle, each component 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 components or even all nodes. The replication 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.

Neptune supports atomic execution of service operations through failures only if each underlying service component 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 component provides a CHECK callback so that the Neptune service 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 [9], [21]. It is only invoked during the node recovery phase and we will further discuss its usage and a potential implementation in Section III-C.

We also assume that the service abort happens either at all replicas or not at all, which frees Neptune from coordinating service aborts. 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 only related to the state of service data, *e.g.*, violations of data integrity constraints. In other words, we do not consider the situation in which a service aborts due to insufficient disk space. This assumption is not crucial to Neptune's fundamental correctness but it greatly simplifies our implementation. Without such an assumption, a proper UNDO mechanism will be required to maintain replication consistency.

B. Multi-level Replication Consistency Model

In general, data replication is achieved through either *eager* or *lazy* write propagations [3], [13], [20]. Eager propagation keeps all replicas exactly synchronized by acquiring locks and updating data at all replicas within a globally coordinated transaction. 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 [20]. The DDS project uses this synchronous approach and they rely on the timeout abort and client retry to resolve the deadlock issue [22]. Neptune's consistency model extends the previous work in lazy propagation with a focus on high scalability and runtime fail-over support. Lazy propagation introduces the problems of out-of-order writes and reading stale data versions. In order to support applications with different service semantics and consistency requirements, Neptune employs a flexible three-level replication consistency scheme to address these problems. In particular, Neptune's highest consistency level provides a staleness control which contains a quantitative staleness bound and a guarantee of progressive version delivery for each client's perspective.

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 persistent data will converge to a consistent state after all outstanding writes are completed. 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 [28], 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 replication consistency is intended to take advantage of application characteristics and to achieve 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 network 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 reading 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 a certain amount of time (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 its implementation 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 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.

We describe our implementation 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 availability announcement 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. Note 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 availability announcements. However, these problems are insignificant in a low latency, reliable local 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 these guarantees, 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 the latest data version as a result of this write. Each client keeps this version number in a session handle (*a.k.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.

As mentioned in Section II, Neptune targets partitionable network services in which service data can be divided into independent partitions. Therefore, Neptune's consistency model does not address data consistency across partition boundaries. Note that a consistency level is specified for each service and thus Neptune allows co-existence of services with different consistency levels.

C. 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 [18] 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 have failed due to our fail-stop assumption and thus this replica can only rejoin the service cluster after going through a recovery process.

When a node recovers after its failure, the underlying single-site service component 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 component. This is done by using registered CHECK callbacks to determine whether each write in the uncommitted log has been completed by the service component. 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. For each partition that the recovering node 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 each partition that the recovering node 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 the recovering node has the highest priority for some data partitions it hosts, 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 V-C. 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 and then all other replicas can proceed according to the standard three-phase recovery.

Before concluding the discussion on our failure recovery model, we describe a possible implementation of the CHECK callback facility. We require the Neptune service 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 component compares it with the LSN of the latest completed service access and returns appropriately. As we mentioned in Section III-A, Neptune provides atomic execution through failures only if the underlying service component 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.

IV. 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 the /proc file system in Linux or the kstat facility in Solaris. We 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 in the service yellow page of each node, 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, hosted services can be compiled into a dynamically linked library. They are linked into Neptune process space at runtime and run as threads. Alternatively, each service instance could run as a separate OS process, which provides better fault isolation and resource control at the cost of degraded performance. 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 services.

Overall, this implementation incurs moderate overhead. For instance, we implemented an echo service on top of a Neptune-enable Linux cluster connected by a switched 100Mbps Ethernet. The echo service simply responds to the client with a message identical to the request message. The response time of such an echo service is measured at 1128μs excluding the polling overhead incurred in service load balancing. Taking away the TCP roundtrip time with connection setup and teardown which is measured at 1031μs, Neptune is responsible for an overhead of 97μs in each service invocation.

So far we have deployed a document search engine, an online discussion group, an auction, and a persistent cache service on Neptune clusters, described as follows. Document search serves read-only user requests, thus it is not concerned with the issue of replication consistency.

- *Document search* 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. Neptune has been deployed at Web search engines Teoma [29] and Ask Jeeves [6]. As of summer 2003, a Neptune-enabled Linux/Solaris cluster at Ask Jeeves contains over 1,200 SMP servers and maintains a search index of more than 1 billion Web documents.
- *On-line 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 uses MySQL database to store and retrieve messages and topics.

- A prototype *auction* service is also implemented on MySQL database. Level three consistency with proper staleness bound and progressive version delivery is desirable for this service because auction users are sensitive to data staleness.
- *Persistent cache* 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-based hashing function to map the key space into a set of buckets. We use the mmap utility 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 III-C. The persistent cache is typically used as an internal utility, providing a scalable and reliable data store for other services. The level two consistency can be used for this service, which allows high throughput at the cost of intermittent false cache misses.

V. Performance Evaluation

Our evaluation examines the performance-scalability, impact of workload variation, and fault-tolerance for Neptune's replication management. In addition, we also provide a performance evaluation on Neptune's cluster load balancing support. All the evaluations are conducted on a rack-mounted Linux cluster with 30 dual 400MHz Pentium II nodes, each of which contains 512MB or 1GB memory. Each node runs Linux 2.2.15 and has two 100Mbps Ethernet interfaces. The cluster is connected by a Lucent P550 Ethernet switch with 22Gbps backplane bandwidth. MySQL 3.23.22-Beta is used for the discussion group and auction services. 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. Our persistent cache is built on a regular file system without atomic recovery support. We believe such a setting is sufficient for illustrative purposes.

This paper is focused on the clustering support and replication management within local-area server clusters. Networking issues between protocol gateways and wide-area clients are out of the scope of this study. Therefore our experiments target the performance evaluation between client nodes and service nodes inside a Neptune service cluster. All the experiments presented in this section use up to 16 service nodes and 6 client nodes.

We use synthetic workloads in our evaluations on replication management. Two types of workloads are generated for this purpose: 1) *Balanced workloads* where service requests are evenly distributed among data partitions are used to measure the best case scalability. 2) *Skewed workloads* are used to measure the system performance when some particular partitions draw a disproportional number of service requests. In our performance report, the system throughput is measured as the maximum request rate at which more than 98% of client requests can be successfully completed within two seconds.

A. Replication Management under Balanced Workload

We use the discussion group to study the system scalability under balanced workload. We vary the replication degree, the number of service nodes, the write percentage, and consistency levels in this evaluation. The replication degree is the number of replicas for each partition. The write percentage indicates the proportion of writes in all requests. We use two write percentages in our evaluation: 10% and 50%. We measure all three consistency levels in this study. Level one consistency requires writes to be commutative and thus, we use a variation of the original service implementation to facilitate it. For the purpose of performance comparison with other consistency levels, we keep the changes to a minimum. For level three consistency, we choose one second as the staleness bound. We also notice 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 choose a very high per-client request rate (one request per second) in this evaluation to measure the worst case scenario.

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

Figure 3 shows the scalability of discussion group service at various configurations. 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 that does not incur 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. Note 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 expensive SQL queries.

We can draw the following conclusions based on the results in Figure 3: 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 significant 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%

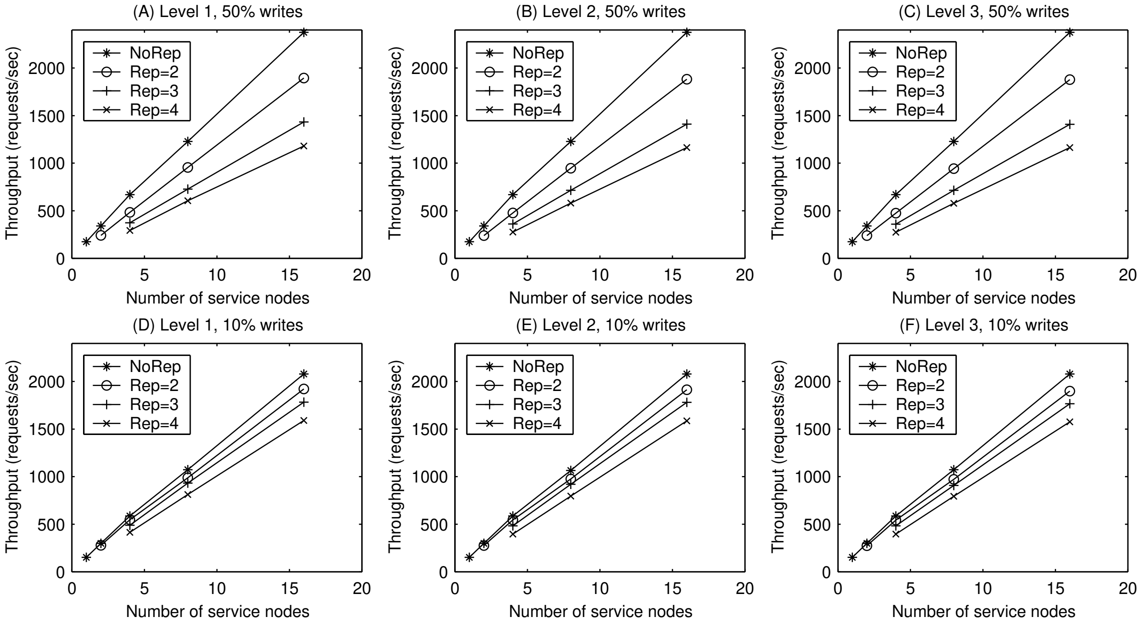


Fig. 3. Scalability of the discussion group service under balanced workload.

writes. However, Section V-B shows that replicated services can outperform non-replicated services under skewed workloads due to better load balancing. 3) All three consistency levels perform very similarly under balanced workloads. This means level one consistency does not provide a significant performance advantage and our staleness control does not incur significant overhead either.

B. Impact of Partition Imbalance for Replication Management

This section studies the performance impact of partition 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 *partition 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 extreme in which all requests are directed to one single partition.

Figure 4(A) shows the impact of partition imbalance on services with different replication degree for the discussion group service. The 10% write percentage, level two consistency, and eight service nodes are used in this experiment. We see that even though 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 where all requests are directed 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 replication consistency.

We learned from Section V-A that all three consistency levels perform very closely under balanced workload. Figure 4(B) illustrates the impact of workload imbalance on different consistency levels. The 10% write percentage, a replication degree of four, and eight service nodes are used in this experiment. The performance difference among three consistency levels becomes slightly more prominent when the partition imbalance factor increases. Specifically under the workload where all requests are directed to one single partition, level one consistency yields 12% better performance

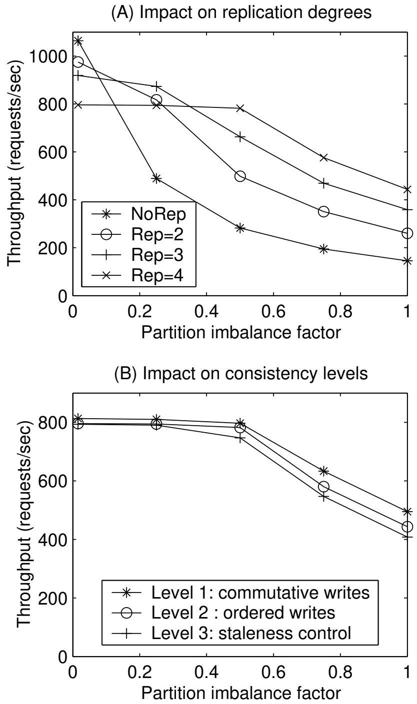


Fig. 4. Impact of partition imbalance on an 8-node system.

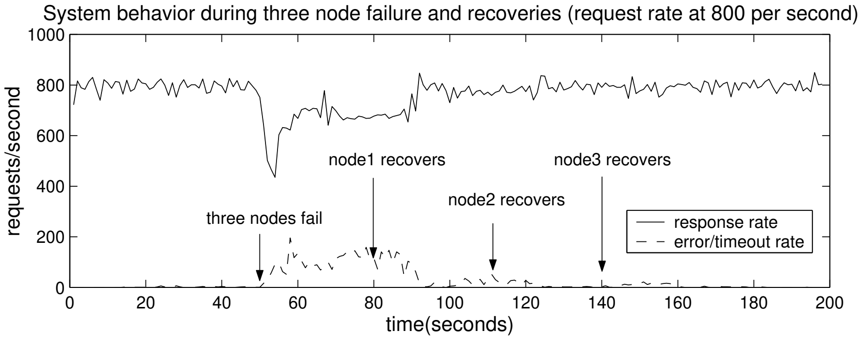


Fig. 5. Behavior of the discussion group service during three node failure and recoveries. Eight service nodes, level two consistency, and a replication degree of four are used in this experiment.

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 learn 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.

C. Replication Management during Failure Recoveries

Figure 5 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 are 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. 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. A sharp performance drop at the beginning of the failure period is caused by the temporary loss of primaries, which is remedied after the elections of new primaries. And we also observe a tail of errors and timeouts trailing each node recovery. This is caused by the overhead of synchronizing lost updates as part of the recovery. Overall, the service quickly stabilizes and resumes normal operations after the recoveries.

D. Replication Performance of Auction and Persistent Cache

In this section, we present the performance of replication management for the Neptune-enabled auction and persistent cache service. Our test workload for auction is based on the published access trace from eBay between May 29 and June 9, 1999. Excluding the requests for embedded images, we estimate that about 10% of the requests are for bidding, and 3% are for adding new items. More information about this analysis can be found in our earlier study on dynamic web caching [36]. We choose the number of auction categories to be 400 times the number of service nodes. Those categories are in turn divided into 64 partitions. Each request is made

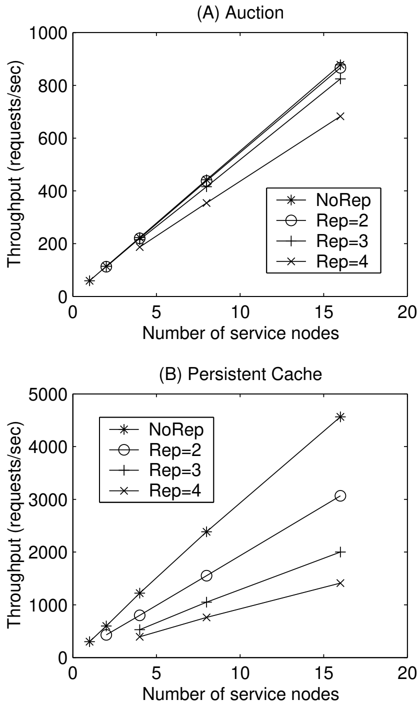


Fig. 6. Replication performance for auction and persistent cache.

for an auction category selected from a population according to an even distribution. We choose 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 6(A) shows the performance of a Neptune-enabled auction service. In general the results match the performance of the discussion group with 10% writes in Section V-A. 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. Most writes for the auction service are bidding requests which incur small overhead.

Figure 6(B) illustrates the performance of the persistent cache service. Level two consistency and 10% write percentage are used in the experiment. The results show large replication overhead caused by extra writes. This is because

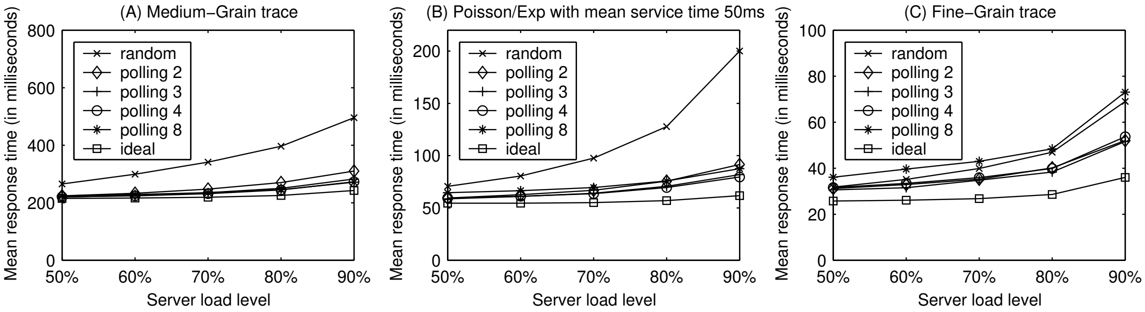


Fig. 7. Performance of cluster load balancing on a 16-server system.

CacheUpdate may cause costly disk accesses while CacheLookup can usually be fulfilled with in-memory data.

E. Evaluation on Cluster Load Balancing

This section shows a performance evaluation on Neptune's cluster-wide load balancing support. We use read-only services in this study to minimize the impact of replication management on the load balancing performance. This evaluation is based on two service traces from Web search engine Teoma [29]. 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. It has a mean service time of 22.2ms and we call it the *Fine-Grain* trace. The other service supports a similar translation for Web page descriptions. It has a mean service time of 208.9ms and we call it the *Medium-Grain* trace. In addition to the two traces, we also include a synthetic workload with Poisson process arrivals and exponentially distributed service times. We call this workload *Poisson/Exp*. The service processing for Poisson/Exp is emulated using a CPU-spinning microbenchmark that consumes the same amount of CPU time as the intended service time.

Figure 7 shows the mean response time for using different load balancing policies. The random polling policies with the poll size of 2, 3, 4, and 8 are compared with the random and ideal approach in a 16-server system. The ideal performance in this study is measured through a centralized load index manager which keeps track of all server load indexes. In terms of server load level, we consider a server has reached full load (100%) when around 98% of client requests are successfully completed within two seconds. Then we use this as the basis to calculate the client request rate for various server load levels. We observe that the results for Medium-Grain trace and Poisson/Exp workload largely confirm Mitzenmacher's analytical results [24]. For the Fine-Grain trace, however, 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 resulted from larger poll size; 2) less accurate server load information 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.

Workload	Mean response time		Performance gain
	Original	Optimized	
Medium-Grain trace	282.1ms	283.1ms	-0.4%
Poisson/Exp	81.8ms	79.2ms	3.2%
Fine-Grain trace	51.6ms	47.3ms	8.3%

TABLE I
PERFORMANCE GAINS OF DISCARDING SLOW-RESPONDING POLLS.

Table I shows the performance improvement of discarding slow-responding polls. The experiments are conducted with a poll size of 3 and a server load level of 90%. The result on the Medium-Grain trace shows a slight performance degradation due to the loss of load information. However, the results on both Fine-Grain trace and Poisson/Exp workload exhibit sizable improvement. Overall, the enhancement of discarding slow-responding polls can improve the load balancing performance by up to 8.3%. Note that the performance results shown in Figure 7 are not with discarding slow-responding polls.

VI. Related Work

Our work is built upon a large body of previous research in network service clustering and replication management in distributed systems. We describe the work related to this paper in the following categories.

Software infrastructure for cluster-based network services. Previous work has addressed the scalability and availability issues in providing clustering support for network service aggregation and load balancing [17], [32]. For instance, the TACC project employs a two-tier architecture in which the service components called "workers" run on different back-ends while incoming requests to workers are controlled by front-ends [17]. Most of these systems employ two-tier clustering architectures and rely on centralized components to maintain the server runtime workload and service availability information. In comparison, Neptune employs a loosely-connected and functionally-symmetric architecture to achieve high scalability and robustness. This architecture allows Neptune service infrastructure to operate smoothly in the presence of transient failures and service evolutions.

Replication consistency. Replication of persistent data is crucial to load sharing and 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 [20]. A recent study by Anderson *et al.* confirms this using simulations [4]. The asynchronous replication based on lazy propagations has been used in Bayou [27]. Adya *et al.* have studied a type of lazy consistency in which server data is replicated in a client cache [2]. The serializability for lazy propagations with the primary-copy method is further studied by a few other research groups [4], [13] and their work addressed causal dependence for accessing multiple objects. The recent work by Yu and Vahdat provides a tunable framework to exploit the tradeoff among availability, consistency, and performance [33]. Neptune's replication consistency maintenance differs from the previous work in providing flexible replication consistency at the clustering middleware level. In particular, Neptune's multi-level replication consistency model supports data staleness control at its highest level.

Neptune's data staleness control is similar to the session guarantees in the Bayou project [30]. Bayou's session guarantee supports the following properties for operations belonging to the same client session: "read your writes", "monotonic reads", "writes follow reads", and "monotonic writes". The first two guarantees are covered in Neptune's "progressive version delivery". The other two properties are inherently supported by level two and three of Neptune's replication consistency model. Additionally, our "soft quantitative bound" enforces a soft bound on the data staleness in time quantities. Both "soft quantitative bound" and "progressive version delivery" are important for staleness-sensitive services like auction.

Replication support for network services and database systems. The Ivory system provides automatic state replication for dynamic Web services through Java bytecode rewriting [10] and the Porcupine project developed a large-scale cluster-based email service with replicated mailboxes [28]. Replication models for these systems are intended for services with only commutative updates, for which ordered updates are unnecessary. Recently the DDS project addressed replication of persistent data with a carefully built data management layer that encapsulates scalable replication consistency and fail-over support [22]. While this approach has been 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.

Traditional databases and transaction processing systems enforce ACID consistencies in replication management through global synchronous locking [9], [21], [31]. Recently, 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 supporting replication management in the clustering middleware level. The key advantage is that such a support can be easily applied to a large number of applications with different underlying data management mechanisms. Our work demonstrates this wide applicability can be achieved along with flexible replication consistency, performance scalability, and failure recovery support for cluster-based network services.

Clustering load balancing. A large body of work has been done to optimize HTTP request distribution among a cluster of Web servers [5], [11], [23], [26]. 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. Our study in this paper shows that an optimized random polling policy that does not require centralized statistics can deliver good load balancing performance.

Previous research has proposed and evaluated various load balancing policies for distributed systems [7], [14], [24], [35]. Those studies are mostly designed for coarse-grain distributed computation and they often ignore fine-grain jobs by simply processing them locally. Built on top of these previous results, our study finds that the random polling policy with a small poll size performs well for cluster-based network services. In addition, we find that the technique of discarding slow-responding polls can further improve the performance for fine-grain services.

VII. Concluding Remarks

Building large-scale network services with the ever-increasing demand on scalability and availability is a challenging task. This paper investigates techniques in building the Neptune middleware system. Neptune provides clustering support and replication management for scalable network services. The system has been implemented on Linux and Solaris clusters where a number of applications have been successfully deployed.

Neptune provides multi-level replication consistency for cluster-based network services with performance scalability and fail-over support. This work demonstrates that a flexible replication management framework can be built at the clustering middleware level. 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 network services. Nevertheless, the current three consistency levels are not meant to be complete. Additional consistency levels can be incorporated into Neptune's replication consistency model when needed.

References

[1] The Alexandria Digital Library Project. http://www.alexandria.ucsb.edu.

[2] 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.

[3] 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.

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

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

[6] Ask Jeeves Search. http://www.ask.com.

[7] 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.

[8] 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.

[9] P. Bernstein and E. Newcomer. *Principles of Transaction Processing*. Morgan Kaufmann Publishers, 1997.

[10] G. Berry, J. Chase, G. Cohen, L. Cox, and A. Vahdat. Toward Automatic State Management for Dynamic Web Services. In *Proc. of the Network Storage Symposium*, Seattle, WA, October 1999.

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

[12] L. Chu, K. Shen, and T. Yang. A Guide to Neptune: Clustering Middleware for Online Services, April 2003. Part of the Neptune software distribution. Available at http://www.cs.ucsb.edu/projects/neptune.

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

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

[15] eBay Online Auctions. http://www.ebay.com.

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

[17] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-Based Scalable Network Services. In *Proc. of the 16th ACM Symposium on Operating System Principles*, pages 78–91, Saint Malo, France, October 1997.

[18] H. Garcia-Molina. Elections in a Distributed Computing System. *IEEE Transactions on Computers*, 31(1):48–59, January 1982.

[19] Google Search. http://www.google.com.

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

[21] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, 1993.

[22] 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.

[23] 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 International World Wide Web Conference*, Brisbane, Australia, April 1998.

[24] 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.

[25] MSN Groups Service. http://groups.msn.com.

[26] 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 8th ACM Conference on Architectural Support for Programming Languages and Operating Systems*, pages 205–216, San Jose, CA, October 1998.

[27] 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.

[28] 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.

[29] Teoma Search. http://www.teoma.com.

[30] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch. Session Guarantees for Weakly Consistent Replicated Data. In *Proc. of International Conference on Parallel and Distributed Information Systems*, pages 140–149, Austin, TX, September 1994.

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

[32] J. Robert von Behren, E. A. Brewer, N. Borisov, M. Chen, M. Welsh, J. MacDonald, J. Lau, S. Gribble, and D. Culler. Ninja: A Framework for Network Services. In *Proc. of the USENIX Annual Technical Conference*, Monterey, CA, June 2002.

[33] 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.

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

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

[36] H. Zhu and T. Yang. Class-based Cache Management for Dynamic Web Contents. In *Proc. of the IEEE INFOCOM*, Anchorage, AK, April 2001.

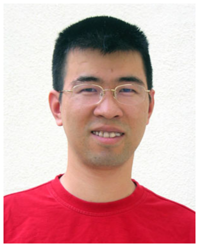
Kai Shen received the B.S. degree in Computer Science and Engineering from Shanghai Jiaotong University, China, in 1996, and the Ph.D. degree in Computer Science from the University of California, Santa Barbara, in 2002.

He is currently an Assistant Professor in the Department of Computer Science at the University of Rochester, Rochester, New York. His research interests are in the areas of parallel and distributed systems, computer networks, and Web searching.

Tao Yang received the B.S. degree in Computer Science from Zhejiang University, China, in 1984, and the M.S. and Ph.D. degrees in Computer Science from Rutgers University, in 1990 and 1993, respectively. He is an Associate Professor at the Department of Computer Science, University of California, Santa Barbara. His main research interests have been parallel and distributed systems, high performance scientific computing, cluster-based network services, and Internet search. He has published more than seventy refereed conference papers and journal articles.

Dr. Yang has also been the Chief Scientist and VP of Research and Development at Teoma Technologies from 2000 to 2001, the Chief Scientist at Ask Jeeves since 2001, where he has directed research of large-scale Internet search and led the development of Teoma search engine running on giant SMP clusters.

Dr. Yang has been an Associate Editor for IEEE Transactions on Parallel and Distributed Systems from 1999 to 2002 and served on the program committee of many high performance computing conferences. Dr. Yang received the Research Initiation Award from NSF in 1994, the Computer Science Faculty Teacher Award in 1995 from the UCSB College of Engineering, the CAREER Award from NSF in 1997, and Noble Jeeviant Award from AskJeeves in 2002.



Lingkun Chu received the B.S and M.S. degrees in Computer Science from Zhejiang University, China, in 1996 and 1999, respectively. He is a Ph.D. candidate at the Department of Computer Science, University of California, Santa Barbara.

He is currently working with Prof. Tao Yang in the area of parallel and distributed systems, with a particular focus on cluster-based network services.