Neptune: Scalable Replication Management and Programming Support for Cluster-based Network Services

Kai Shen, Tao Yang, Lingkun Chu, JoAnne L. Holliday, Douglas A. Kuschner, and Huican Zhu
Department of Computer Science, University of California at Santa Barbara, CA 93106
{kshen, tyang, lkchu, joanne46, kuschner, hczhu}@cs.ucsb.edu

Abstract

Previous research has addressed the scalability and availability issues associated with the construction of cluster-based network services. This paper studies the clustering of replicated services when the persistent service data is frequently updated. To this end we propose Neptune, an infrastructural middleware that provides a flexible interface to aggregate and replicate existing service modules. Neptune accommodates a variety of underlying storage mechanisms, maintains dynamic and location-transparent service mapping to isolate faulty modules and enforce replica consistency. Furthermore, it allows efficient use of a multi-level replica consistency model with staleness control at its highest level. This paper describes Neptune's overall architecture, data replication support, and the results of our performance evaluation.

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 [4, 7, 17, 23]. 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, failover support, and component reusability and extensibility for cluster-based services [7, 12]. 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 and it is focused on a class of distributed data structures [11]. In comparison, we design and build an infrastructural middleware, called *Neptune*, with the goal of clustering and replicating existing stand-alone service modules that use various storage management mechanisms.

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 [3, 9]. Various asynchronous models have been proposed for wide-area or wireless distributed systems [2, 3, 9, 15, 22]. These results are in certain parts applicable for cluster-based Internet services; however they are designed under the constraint of high communication overhead in wide-area or wireless networks. Additional studies are needed to address high scalability and runtime failover support required by cluster-based Internet services [6, 18].

The work described in this paper is built upon a large body of previous research in network service clustering, fault-tolerance, and data replication. The goal of this project is to propose a simple, flexible yet efficient model in aggregating and replicating network service modules with frequently updated persistent data. The model has to be simple enough to shield application programmers from the complexities of data replication, service discovery, load balancing, failure detection and recovery. 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 with varying performance tradeoffs. In particular, we have developed a consistency scheme with staleness control.

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.

The rest of this paper is organized as follows. Section 2 presents Neptune's overall system architecture and the assumptions that our design is based upon. Section 3 describes Neptune's multi-level replica consistency scheme and the failure recovery model. Section 4 illustrates a prototype system implementation and three service deployments on a Linux cluster. Section 5 evaluates Neptune's performance and failure management using those three services. Section 6 describes related work and Section 7 concludes the paper.

2 System Architecture and Assumptions

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 data manipulated by such a service can be divided into a large number of independent data 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 flat architecture in constructing the service infrastructure and a node is distinguished as a client or a server only in the context of each specific service invocation. In other words, a server node in one service invocation may act as a client node in another service invocation. Within a Neptune cluster, all the nodes are loosely connected through a well-known publish/subscribe channel. Published information is kept as soft state in the channel such that it has to be refreshed frequently to stay alive [16]. This channel can be implemented using IP multicast or through 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 node can also choose to host a Neptune client module which subscribes to the well-known channel and maintains a service/partition mapping table. Services can be acquired by any node in the cluster through the local Neptune client module by using the published service access interface. The aggregate service could be exported to external clients through protocol gateways.

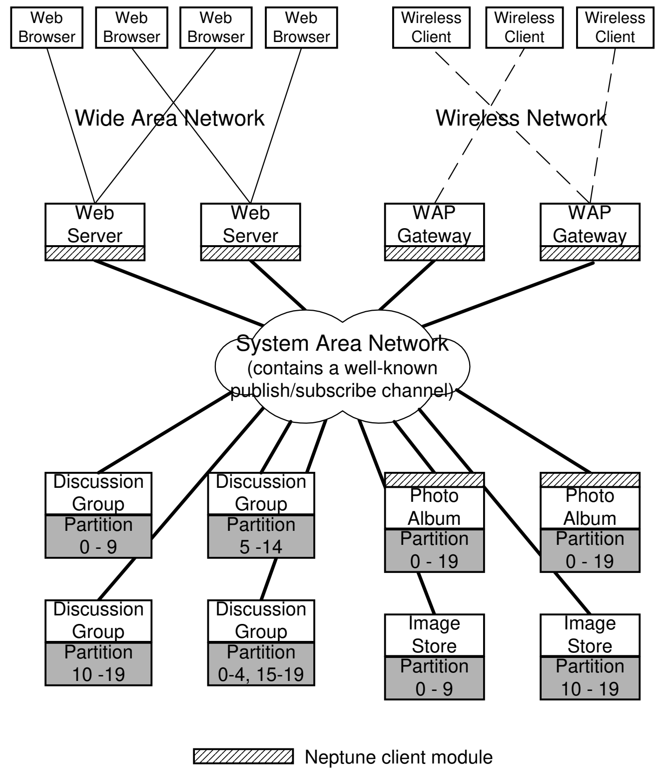

Figure 1: Architecture of a sample Neptune service cluster.

Figure 1 illustrates the architecture of a sample Neptune service cluster. In this example, 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. All the persistent data are divided into twenty partitions according to user accounts. The discussion group service is delivered independently while the photo album service relies on an internal image store service. Therefore, each photo album node needs to host a Neptune client module to locate and access the image store in the service cluster. A Neptune client module is also present in each gateway node in order to export internal services to external clients. The loosely-connected and flat architecture allows Neptune service infrastructure to operate smoothly in the presence of transient failures and through service evolution.

2.1 Neptune Modules and Interfaces

Neptune supports two communication schemes between clients and servers: a request/response scheme and a stream-based scheme. In the request/response scheme, the client and the server 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 server 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 server. 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 support for the request/response scheme.

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.

Neptune provides a client-side module and a server-side module to facilitate location-transparent service invocations. Figure 2 illustrates the interaction among service modules and Neptune modules during a 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, and runtime workload. Currently Neptune's load balancing decision is made based on the number of active service invocations at each service node. Previous work has also shown that locality-aware request distribution could yield better caching performance [14]. We plan to incorporate locality-based load-balancing schemes in the future. 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. The service instance could be compiled into a dynamically linked library and linked into Neptune process space during runtime. Alternatively, it could run as a separate process, which would provide better fault isolation and resource control at the cost of degraded performance. Finally, since a data partition may be replicated across multiple nodes, the Neptune server module propagates writes to other replicas to maintain replica consistency.

We discuss below the interfaces between Neptune and service modules at both the client and the server sides:

- 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 and we will discuss it further in Section 3.1. 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. In addition, each service has to provide a CHECK callback which allows the Neptune server module to 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 [10]. It is only invoked during the node recovery phase and we will further discuss its usage and a potential implementation in Section 3.2.

2.2 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. 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.

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

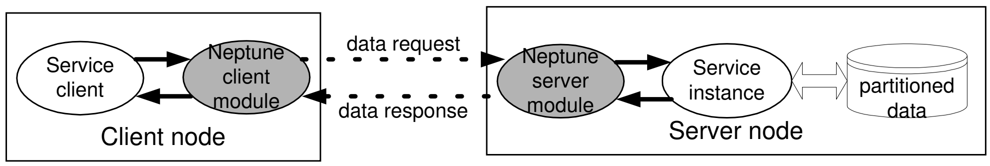

Figure 2: Interaction among service modules and Neptune modules during a request/response service invocation.

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. It should be noted that the current version of Neptune does not have full-fledged transactional support largely because Neptune restricts each service access to a single data partition.

3 Replica Consistency and Failure Recovery

In general, data replication is achieved through either *eager* or *lazy* write propagations [5, 9]. 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 [9]. 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 failover 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.

3.1 Multi-level Consistency Model

Neptune's first two levels of replica consistency are more or less generalized from the previous work [5, 17] 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. Some append-only discussion groups satisfy this commutativity requirement. Another example is a certain kind of email service [17], in which all writes are total-updates, so out-of-order writes could be resolved by discarding all but the newest.

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. This results in a loss of write concurrency within each partition. However, the large number of independent data partitions usually yields significant concurrency across partition boundaries. In addition, the concurrency among reads is not affected by this scheme.

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 when strong consistency is hard to achieve.

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. Note that the "x seconds" is only a soft bound because the real guarantee also 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.

3.2 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 based on the classical Bully Algorithm [8] except that each replica has a priority *for each data partition* it hosts. This failover 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 5.3. 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 2.2, 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.

4 System and Service Implementations

We have implemented and deployed a prototype Neptune infrastructure on a Linux cluster. The node workload is acquired through the Linux `/proc` file system. The publish/subscribe channel is implemented using IP multicast. Each multicast message contains the service announcement and node runtime workload. 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 all the soft states expire in five seconds. That means a faulty node will be detected when five of its multicast messages are not heard in a row. These numbers are rather empirical and we never observed a false failure detection in practice. This is in some degree due to the fact that each node has two network interface cards, which allows us to separate the multicast traffic from other service traffic.

Each Neptune server module could be configured to run service instances as either threads or processes. The server module also keeps a growable process/thread pool and a waiting queue. When the upper limit for the growable pool is reached, subsequent requests will be queued. This scheme allows the Neptune server module to gracefully handle spikes in the request volume while maintaining a desirable level of concurrency.

We have deployed three network services in the Neptune infrastructure. 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.

Our final study was a *persistent cache* service, which 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 3.2. 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 [13].

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 System Evaluations

Our experimental studies are focused on performance-scalability, availability, and consistency levels of Neptune cluster services. All experiments described here were conducted on a rack-mounted Linux cluster with around 40 nodes. The nodes we used in the experiments include 16 dual 400 Mhz Pentium II processors, and 4 quad 500 Mhz Pentium II Xeon processors, all of which contains 1 GBytes 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.

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 paper. Our experiments are instead focused on studying the performance between clients and services inside a Neptune service cluster. During the evaluation, the services were hosted on dual-Pentium IIs and we ran testing clients on quad-Xeons. MySQL 3.23.22-Beta was used as the service database.

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 an 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.

Our experience of using Linux as a platform to build large-scale network services has been largely positive. However, we do observe intermittent kernel crashes under some network-intensive workload. Another problem we had during our experiments is that the system only allows around 4000 ports to stay in the TCP TIME_WAIT state at a given time. This causes our testing clients to perform abnormally in some large testing configurations, which forces us to use more machines to run testing clients than otherwise needed. We have recently ported Neptune to Solaris and we plan to conduct further studies on the impact of different OS networking kernels.

The rest of this section is organized as follows. Section 5.1 and Section 5.2 present the system performance under balanced and skewed workload. Section 5.3 illustrates the system behavior during failure recoveries. The discussion group service is used in all the above experiments. Section 5.4 presents the performance of the auction and persistent cache service.

5.1 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, 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 per-

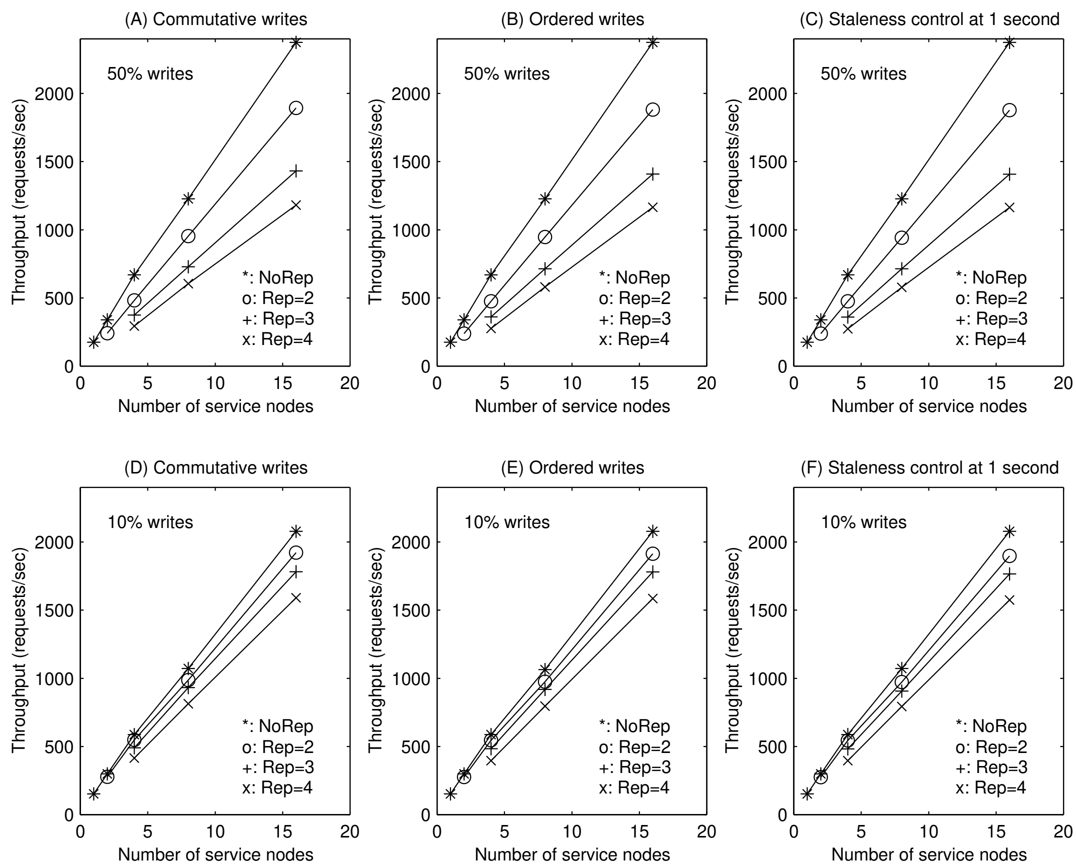

Figure 3: Scalability of discussion group service under balanced workload.

centages: 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 3 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 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 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 5.2 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.

5.2 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.

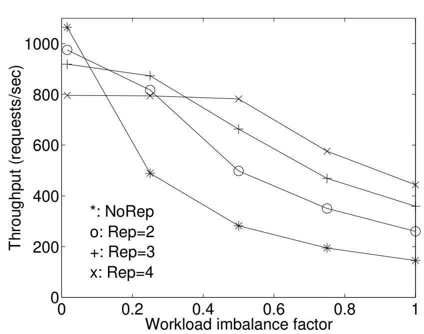

Figure 4: Impact of workload imbalance on the replication degrees with 8 service nodes.

Figure 4 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 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.

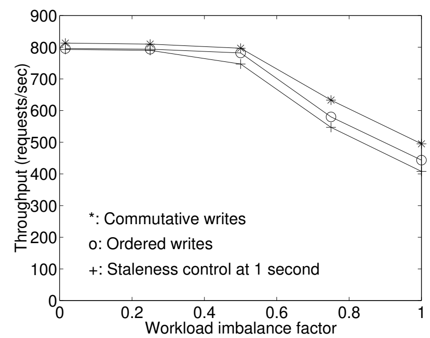

Figure 5: Impact of workload imbalance on consistency levels with 8 service nodes.

We learned from Section 5.1 that all three consistency levels perform very closely under balanced workload. Figure 5 illustrates the impact of workload imbalance on different consistency levels. The 10% write percentage, a replication 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.

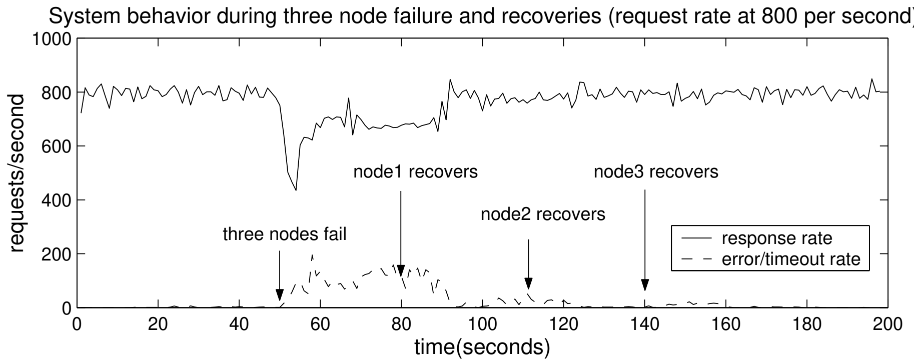

Figure 6: Behavior of the discussion group service during three node failure and recoveries. Eight service nodes, level two consistency, and a replication degree of four were used in this experiment.

5.3 System Behavior during Failure Recoveries

Figure 6 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 in 30 seconds is fairly common for component failures. System crashes usually take longer to recover, but operating systems like Linux could be configured to automatically reboot a few seconds after kernel panic. 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 3.2. However, the service quickly stabilizes and resumes normal operations.

5.4 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 [24]. 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.

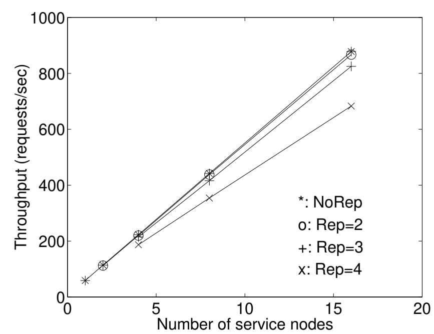

Figure 7: Performance of auction on Neptune.

Figure 7 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 5.1. 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 8 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

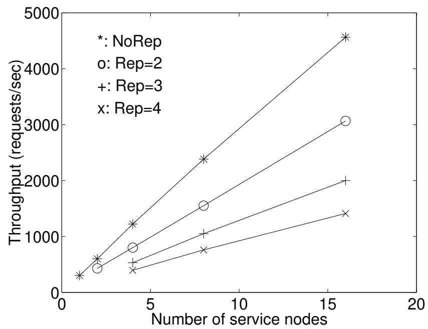

Figure 8: Performance of persistent cache on Neptune.

while `CacheLookup` can usually be fulfilled with inmemory data.

6 Related Work

Our work is in large part motivated by the TACC and MultiSpace projects [7, 12] with a focus on providing infrastructural support for data replication and service clustering. It should be noted that replication support for cluster-based network services is a wide topic. Previous research has studied the issues of persistent data management for distributed data structures [11] and optimistic replication for Internet services [18]. Our work complements these studies by providing infrastructural software support for clustering stand-alone service modules with the capability of accommodating various underlying data management solutions and integrating different consistency models.

The earlier analysis by Gray et al. shows that the synchronous replication based on eager update propagations leads to high deadlock rates [9]. A recent study by Anderson et al. confirms this using simulations [3]. The asynchronous replication based on lazy propagations has been used in Bayou [15]. Adya et al. have studied lazy replication with a type of lazy consistency in which server data is replicated in a client cache [1]. The serializability for lazy propagations with the primary-copy method is further studied by a few other research groups [3, 5] 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 [22]. These studies are mainly targeted on loosely-coupled distributed services, in which communication latency is relatively high and unbounded. In comparison, our contribution focuses on time-based staleness control by taking advantage of low latency and high throughput system area networks.

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 [20, 10].

Providing reliable services in the library level is addressed in the SunSCALR project [19]. Their work relies on IP failover to provide failure detection and automatic reconfiguration. The Microsoft cluster service (MSCS) [21] offers an execution environment where off-the-shelf server applications can operate reliably on an NT cluster. These studies do not address persistent data replication.

7 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 because of 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 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, we plan to further investigate the incorporation of stronger consistency models in the future.

Acknowledgment. This work was supported in part by NSF CCR-9702640, ACIR-0082666 and 0086061. We would like to thank Amr El Abbadi, Hong Tang, and the anonymous referees for their valuable comments.

References

[1] 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, August 1997.

[2] 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.

[3] 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.

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

[5] 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.

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

[7] 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, October 1997.

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

[9] 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.

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

[11] 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.

[12] 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, 1999.

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

[14] 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 (ASPLOS-VIII)*, San Jose, CA, October 1998.

[15] 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.

[16] S. Raman and S. McCanne. A Model, Analysis, and Protocol Framework for Soft State-based Communication. In *Proc. of ACM SIGCOMM'99*, pages 15–25, Cambridge, Massachusetts, September 1999.

[17] 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, December 1999.

[18] Y. Saito and H. Levy. Optimistic Replication for Internet Data Services. In *Proc. of the 14th Intl. Conf. on Distributed Computing*, October 2000.

[19] 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.

[20] BEA Systems. Tuxedo Transaction Application Server White Papers. http://www.bea.com/products/tuxedo/papers.html.

[21] 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.

[22] 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.

[23] 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.

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