UNIVERSITY OF CALIFORNIA, SAN DIEGO

Fault-tolerant Grid Services

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in Computer Science

by

Xianan Zhang

Committee in charge:

Professor Keith Marzullo, Chair
Professor Samuel Buss
Professor Sidney Karin
Professor Kenneth Kreutz-Delgado
Professor Geoff Voelker

2006

The dissertation of Xianan Zhang is approved, and it is acceptable in quality and form for publication on microfilm:

Chair

University of California, San Diego

2006

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

This dissertation would not have been possible without the support and encouragements of many people. I owe debts of gratitude to those people who introduce me to the fascinating computer system research world and help me in different aspects.

First and foremost, to my advisor and mentor, Professor Keith Marzullo, who brought me into this area and led me through the graduate study by teaching me how to solve problems and how to express more clearly. My graduate student experience has been both enjoyable and successful because Professor Marzullo has always been supportive and understanding. My thanks also go to all my committee members, Prof. Samuel Buss, Prof. Sidney Karin, Prof. Kenneth Kreutz-Delgado and Prof. Geoff Voelker for their helpful suggestions and advice.

Thanks next go to Dr. Rick Schlitching and Dr. Matti Hiltunen. I knew Rick and Matti when I did my summer intern at AT&T Labs - Research in 2003. Since then, we have been working closely together. They are my wonderful mentors and close friends. I learned a lot from working with them, and I enjoyed this experience a lot.

I also thank Dr. Flavio Junqueira who cooperated with me on the asynchronous replication protocols, and Dr. Dmitrii Zagorodnov who cooperated with me on the work on studying primary-backup grid services.

Finally, I owe an enormous dept of gratitude to my family and friends, especially my parents, my dear husband Pengyue Wen and my sweet daughter Jacqueline, for all the love and support they have generously given to me. I would not have made it without you!

Chapter III is, in part, reprints of material as it appears in "Fault-tolerant Grid Services Using Primary-Backup: Feasibility and Performance," by Xianan Zhang, Dmitrii Zagorodnov, Matti Hiltunen, Keith Marzullo, and Richard D. Schlichting, in the Proceedings of the 2004 IEEE International Conference on Cluster Computing, September 2004. The dissertation author was the primary coauthor and co-investigator of this paper.

Chapter IV is, in part, reprints of material as it appears in "Customizable Service State Durability for Service Oriented Architecture," by Xianan Zhang, Matti A. Hiltunen, Keith Marzullo, and Richard D. Schlichting, in the Proceedings of the 6th European Dependable Computing Conference (EDCC-6), October, 2006. The dissertation author was the primary coauthor and co-investigator of this paper.

Chapter VI is, in part, reprints of material as it appears in "Replicating Nondeterministic Services on Grid Environments," by Xianan Zhang, Flavio Junqueira, Matti A. Hiltunen, Keith Marzullo, and Richard D. Schlichting, in the Proceedings of the 15th IEEE International Symposium on High-Performance Distributed Computing (HPDC-15), June, 2006. The dissertation author was the primary coauthor and co-investigator of this paper.

VITA

1976	Born, Jiamusi, Heilongjiang Province, P.R.China
1999	B.S. Department of Computer Science, Peking University, Beijing, P.R.China
2000 – 2006	Research Assistant, Department of Computer Science and Engineering, University of California, San Diego
2002	M.S., Department of Computer Science and Engineering, University of California, San Diego
2003	Summer Intern, AT&T Labs - Research
2004	Teaching Assistant, Department of Computer Science and Engineering, University of California, San Diego
2004	Lecturer, Department of Computer Science and Engineering, University of California, San Diego
2006	Intern, IBM Almaden Research Lab
2006	Ph.D., Department of Computer Science and Engineering, University of California, San Diego

PUBLICATIONS

Xianan Zhang, Matti A. Hiltunen, Keith Marzullo, and Richard D. Schlichting, "Customizable Service State Durability for Service Oriented Architecture", in the Proceedings the 6th European Dependable Computing Conference (EDCC-6), October, 2006.

Xianan Zhang, Flavio Junqueira, Matti A. Hiltunen, Keith Marzullo, and Richard D. Schlichting, "Replicating Nondeterministic Services on Grid Environments", in the Proceedings of the 15th IEEE International Symposium on High-Performance Distributed Computing (HPDC-15), June, 2006.

Xianan Zhang, Dmitrii Zagorodnov, Matti Hiltunen, Keith Marzullo, and Richard D. Schlichting, "Fault-tolerant Grid Services Using Primary-Backup: Feasibility and Performance", in the Proceedings of the 2004 IEEE International Conference on Cluster Computing, September 2004.

Kjetil Jacobsen, Xianan Zhang, and Keith Marzullo. Group Membership and Wide-Area Master-Worker Computations, in Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS 2003), May 19-22, 2003, Providence, Rhode Island, USA.

Hyojong Song, Xin Liu, Denis Jakobsen, Ranjita Bhagwan, Xianan Zhang, Kenjiro Taura, and Andrew Chien. The MicroGrid: a Scientific Tool for Modeling Computational Grids, Super Computing 2000 (SC 2000), Nov. 4-10, 2000, Dallas, TX.

Hyojong Song, Xin Liu, Denis Jakobsen, Ranjita Bhagwan, Xianan Zhang, Kenjiro Taura, and Andrew Chien. The MicroGrid: a Scientific Tool for Modeling Computational Grids, Scientific Programming, Vol. 8, No. 3, pp. 127-141, 2000.

FIELDS OF STUDY

Major Field: System and Networking, Computer Science

ABSTRACT OF THE DISSERTATION

Fault-tolerant Grid Services

by

Xianan Zhang
Doctor of Philosophy in Computer Science
University of California, San Diego, 2006
Professor Keith Marzullo, Chair

My thesis investigates how to replicate services using the primary-backup approach in grid environments. First, the design of a primary-backup protocol using the OGSI grid standard and implementation based on Globus Toolkit 3 are presented, in order to evaluate the feasibility and performance of existing grid middlewares. Then an architecture of integrating the primary-backup replication with other techniques is proposed and evaluated. Finally, this thesis addressing the issue replicating services on asynchronous grid environments. The experiments on local clusters and PlanetLab show the protocols we proposed provide good performance for replicating grid services.

Chapter I

Introduction

Grid concepts and technologies [26] were originally developed to enable resource sharing among scientific collaborators. Scientists use grid middleware plateforms to manage their computing resources and storage resources, so that when they want to deploy the experiments and need more resources than available locally, they can borrow resources from collaborators while ensuring safety, privacy, and efficiency of the resource owners' jobs and data at the same time. For example, suppose that research lab A and B both have certain computing capabilities. When A needs to handle a computing task that may take days or weeks to execute, and B's machines happen to be idle at that time, time can be saved if A is capable of using some of B's computing power for its task. Furthermore, B may be happy to help A since B may also need similar helps from A from time to time when there is a need for B to handle a task beyond its capability. On the other hand, B should not have to relinquish priority, access, privacy and security of its machines for allowing A to use them. Scientific grid has achieved great successes, through testbeds and cooperative projects, such as GrADS [8], GriPhyN [30] and TeraGrid [55].

More recently, it has become clear that similar requirements arise in the commercial sector such as enterprise application integration and business-to-business partner collaboration over the Internet. The case given above for two research labs can also happen for two business partners or two departments of the

same company. Just as the Web began as a technology in scientific community and was adopted by e-business later, we see a similar trend for the Grid.

Since then, the emphasis of grid research has shifted from building basic functions to providing easily deployed and integrated infrastructures and interfaces. Combining grid technologies with service oriented architectures (SOA) meets this requirement. Since web services' standards are the most successful example of the SOA interfaces, they have been extended and evolved to the interfaces of grid services – the Open Grid Services Infrastructure (OGSI) [56] standards and the Web Service Resource Framework (WSRF) [19].

Given the fact that grid resources are heterogeneous and often belong to multi-organizations, failures of the resources are not uncommon. Therefore, it is critical to build reliable grid services based on unreliable resources. Although building reliable services is not a new problem and has been investigated as fault tolerant techniques for years, it remains as an unsolved issue whether the existing fault tolerant protocols can work well for grid services.

Web services are often simply replicated for fault tolerance since they are stateless. The data of the web services are usually stored in the database and hence the services read and write the database tables when they need to refer to the data for serving clients' requests. If one service replica fails, the clients' requests will be directed to another service – either a newly-created one or an existing one. Since no data is lost because of the failed service replica and all the replicas have the same functions, the web service performs well for the clients as long as there are some replica available.

However, this simple approach doesn't work for grid services because they are stateful – unlike for web services: if one replica of a grid service fails, all its states are lost and won't be available at other replicas unless the service replicas synchronize their states with each other.

There are two common fault tolerant approaches for synchronizing replicas' states – state machine and primary-backup. The state machine approach

requires all replicas execute the clients' requests in the same sequence. Despite the fact that this approach has the benefit of tolerating not only machine crashes and network failures, but also the Byzantine failures, it requires the services to be deterministic. On the other hand, in the primary-backup approach one replica serves as the primary while the others are backups. Only the primary executes the clients' requests and sends its new state to the backups. Then, the backups update their states accordingly. In this way, although the primary-backup doesn't tolerate the Byzantine failures, it works even if the services are not deterministic.

Which approach is appropriate depends on whether *nondeterminism* – different replicas of the same service may have divergent states even though they execute the same sequences of commands – is significant in grid services. Here I use two examples to demonstrate how nondeterminism can arise in a grid service context.

The first example is a Grid Resource Broker Service. A "resource broker" is a useful service in a grid environment, and grid resource brokers have been developed in several projects. Large-scale grid platforms contain large numbers of resources and each under the control of a local resource manager. Information about the available resources can typically be found via grid information services [20]. Grid users that wish to acquire grid resources to execute applications could then retrieve this resource information, select resources, and interact with local resource managers directly. This approach, while possible, has two main disadvantages. First, the grid users should not be responsible for developing the above functionality. Indeed, this functionality is likely needed by many users and it makes sense to provide a single implementation that can be shared. Also, resource selection is a notoriously difficult question and many users lack the expertise necessary for implementing it effectively. Second, having each user select resources individually may not be desirable. Indeed, providing a single brokering service that is used by many users provides the opportunity to implement techniques such as load balancing across the resources. For the above reasons, several research

projects have investigated grid resource broker services [28, 48] and tackled the design and development of grid resource brokers, which can be deployed as grid services and isolate users from the complexity of the resource environment.

A resource broker service accepts requests for resources and uses specific algorithms to select appropriate resources among the ones that the broker has discovered using grid resource information service. A popular way to perform such selection is to employ a randomized algorithm [23, 44, 62], so as to achieve load-balancing (e.g., to avoid sending the same reply to two consecutive requests). This is an inherent source of nondeterminism, which is part of the algorithm itself. To make this service fault-tolerant, a simple replication of the service is not feasible as replicas may provide diverging answers.

The second example is a grid scheduling service that accepts jobs from the clients, invokes some resource broker service to select appropriate resources and then deploys the jobs to selected resources. This example actually arose in the NILE Global Planner [4]. Assume the grid scheduling service examines the jobs in First-Come-First-Serve (FCFS) order while FCFS order can be overridden by job priorities. This service can become nondeterministic as follows. Assume Job A arrives at time t_1 and Job B arrives at t_2 ($t_2 > t_1$) with a higher priority. Depending on the speed of the scheduler, if it examines the job queues at any time between t_1 and t_2, it will select Job A. However, if it examines the job queue after t_2, it will schedule Job B before Job A. So the service's behavior depends not only on the sequence of requests received, but also on the processing speed of the machine that hosts the service. Although the service developer does not intend to build a nondeterministic service, this grid scheduling service ends up having this characteristic.

It is often desirable to make such a grid scheduling service highly-available using replication to ensure the clients' jobs are served in a timely manner. The simple approach is not to synchronize these service replicas. Then, when a replica examines the jobs, it does not know which jobs have been examined by other

replicas. Although this approach is relatively easy to implement, it has some limitations. For example, unless it relies on other techniques to collect grid resource information (e.g., machines' load) as the Condor Matchmaker Service does [49], it is hard for the scheduling service to deliver policies such as load-balancing or high throughput without knowing the previous assignments. This follows since the machine load and the job delivery time depend on the jobs that are executing or waiting to execute—a subset of previous scheduled jobs. Another solution is to use replication in time based on message logging techniques [24]. This is the approach that was used in NILE. A third solution, which has much lower fault recovery latency, is to synchronize active replicas of the grid scheduling service and ensure that the service replicas agree on the previous job scheduling decisions. To do this, we need a protocol that can synchronize the replicas of a nondeterministic service.

Since the grid services are often nondeterministic as described in the given examples, my thesis focuses on the fault tolerant approach applicable to nondeterministic services – the primary-backup.

The primary-backup itself has been well studied. But how to make it work well on the grid environments remains as an open question. More specifically, the following issues are worth investigating.

1. **Tradeoff between performance and utilization of grid facilities.** Grid services are designed using very high-level protocols and services. We could not expect that a service built on top of SOAP will have the same latency as a service built directly on top of TCP. Using the functions defined in grid service standards to provide high service availability is appealing: one could provide it as a feature portable across any grid service. But, if the performance is much worse than the same feature implemented as a low level, the performance may outweigh the engineering appeal of a high-level implementation. Therefore, it is critical to analyze the tradeoff between performance and the utilization of facilities provided by the grid standards.

2. **Integration with other techniques making states durable.** The primary-backup makes not only the grid services highly-available, but also the services' states durable against failures. Given the fact that the service states differ in the degree of durability that they require and each mechanism comes with a certain cost, it is appealing to provide an architecture allowing the application developers use different fault-tolerance techniques with different tradeoffs to achieve the durability of the service states. For example, the developer might want to use a database for states that need to survive very severe failures, such as billing information, while use the primary-backup replication for the states that need less durability. In SOAs, services are designed to be independent and largely self-contained abstractions that interact only through well-defined service interfaces. Therefore, it is non-trivial to integrate different fault-tolerance techniques across services in a coherent way, while still providing a good balance with the conflicting requirements of application transparency and execution efficiency.

3. **Practical protocols applicable on asynchronous systems.** The grid environments are often asynchronous – the process execution speeds and message delivery delays are not bounded. In asynchronous systems, it is not possible to implement a perfect failure detector, which is necessary for the primary-backup implementations. Using Consensus algorithms to implement a protocol that work as the primary-backup in asynchronous systems has been proposed and studied only theoretically [22]. No practical study has been reported in the literature. Therefore, it still remains as an open question how we can design and implement the primary-backup on asynchronous systems efficiently in practice.

The following chapters of my thesis are organized as following. Chapter II reviews the background work and introduces the functions of the primary-backup replication, the grid standards and a Consensus protocol *Paxos* on which we built

our protocols for asynchronous systems.

Chapter III studies the tradeoff between performance and use of grid facilities associated with building highly-available grid services using the primary-backup approach. First, the design of a primary-backup protocol using OGSI is presented. Then the performance implications and tradeoffs are investigated based on the implementation on Globus [27] Toolkit 3. In particular, using a simple example grid service, the performance of this implementation is compared with variants in which the primary and backups communicate using standard service method calls and TCP, respectively.

Chapter IV addresses the integration issue by presenting a software architecture for building highly available and reliable services. The central idea of our architecture is the concept of a durable resource, which is an abstraction providing an essential capability: the ability for state to persist across failures. In our architecture, the designer of a resource can control which portions of a service's state are persistent, the tradeoff between the degree of durability and the performance, and the atomicity of accesses to the resource with respect to failures. Such durable resources can then be used as building blocks to construct highly available and reliable services using existing techniques.

Chapter V and chapter VI study how to replicate services on asynchronous systems. Chapter V investigates the issues that impact the practical performance of the protocols in the Paxos family – Classic Paxos and Fast Paxos – using simulations and experiments on a local cluster and PlanetLab. Then we extend Paxos for supporting nondeterministic services. In chapter VI, we first describe the basic idea of how to use the Consensus algorithm for implementing the primary-backup, followed by the demonstration on how to optimize the protocol to reduce the overhead of read requests and requests using transactions. A prototype application is used to evaluate the performance.

Finally, my thesis is concluded in chapter VII.

Chapter II

Background

First, the functions and limitations of the primary-backup replication approach are described in section II.A. Then the concept of Service Oriented Architecture is introduced in section II.B. The grid standards – the Open Grid Service Infrastructure and the Web Service Resource Framework – are discussed in section II.C. Finally, the variant versions of Paxos protocol are introduced in section II.D as the foundation work of asynchronous systems.

II.A Primary–backup

Primary–backup is a well-known technique for making services highly available [1, 9, 11]. A client sends a request to the primary, which receives and executes the request. The primary then sends a state update message to the backups and replies to the client. Typically, the primary does not reply to the client until it knows that all backups have received the state update. This is done to ensure that the backups are always consistent with the client: it is impossible for the client to know that the primary executed the request without the backups also knowing this. Figure II.1 shows a space-time diagram of the execution of a simple primary–backup protocol.

Primary–backup requires that a client be notified that the primary has failed and allow the client to rebind to the newly-appointed primary. Ideally, this

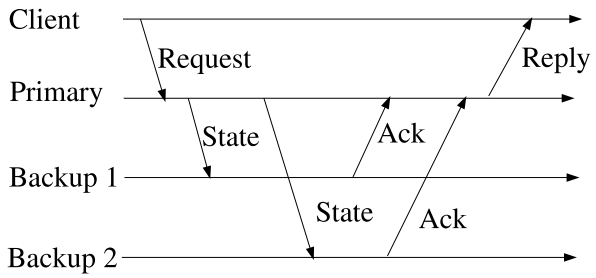


Figure II.1: Primary-backup Protocol

ability should be available below the level of the service request: doing so allows a client designed to interact with a single non-replicated server to be transparently ported to interact with a primary–backup service.

Primary–backup has a few drawbacks. First, it is only suited for tolerating benign failures such as crashes and message loss, rather than arbitrary or malicious failures. Unless malicious failures are a concern in a specific grid environment, we consider masking only benign failures a worthwhile tradeoff for the ability to run non-deterministic applications.

Another drawback is that primary–backup requires that the environment is synchronous enough to support the use of heartbeats to detect failures. In practice, this means that the primary and the backups need to run on a cluster that is managed by some failure detector and recovery manager. Commercial products, such as the VERITAS Cluster Server [57], can be used for this purpose. In systems that may suffer from network partitions or in which there is no bound on process execution speed or message delivery speed, it is not possible to implement a reliable failure detector [25, 16]. Primary-backup replication can not ensure the consistency of replicas on such systems as described in Chapter V and Chapter VI.

II.B Service Oriented Architectures

Service Oriented Architectures (SOAs) structure software functionality in a distributed system as collections of interacting services as illustrated in fig-

ure II.2. The services include both *infrastructure services*, such as directory services (UDDI), monitoring, and resource allocation services, as well as *application services* that implement some application specific service. The infrastructure services, denoted as shaded ovals in the figure, provide the foundation for building, running, and accessing the application services. The web services architecture and grid services architecture are popular examples of SOAs in which services interact by exchanging XML documents.

A service invocation, from a client's desktop application for example, often results in a sequence or chain of interleaved service invocations between different services potentially in different administrative domains. This leads to a situation where, paraphrasing Leslie Lamport, *a service-oriented architecture is one in which the failure of a service you didn't even know existed can render your application unusable.*

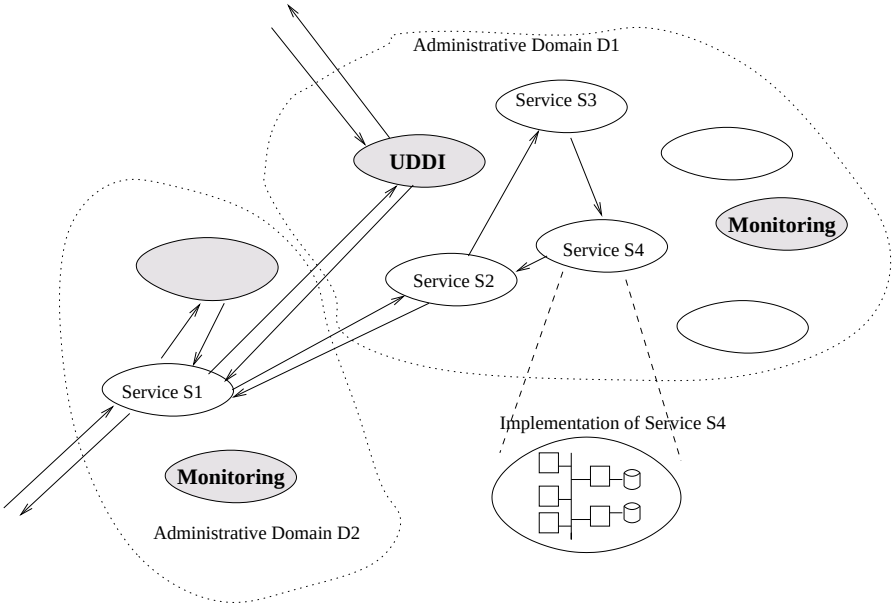


Figure II.2: Example Service Oriented Architecture

II.C The Grid Standards

Building on both grid and Web services technologies, the Open Grid Services Infrastructure (OGSI) defines mechanisms for creating, managing and exchanging information among entities called *grid services*. Unlike Web services, grid services are stateful and may be short-lived. The OGSI model allows each client to choose among several available instances of a service or create its own instance. The instances may have a limited lifetime since resources can be created to serve certain clients and are removed after they are no longer needed.

Interaction between a grid service and a client happens in a request-reply fashion using strictly-defined interfaces and a certain encoding of data (interfaces are described by WSDL and the messages are encoded using SOAP, both of which are XML-based). In addition to regular requests and replies, grid instances may subscribe using an OGSI-specified interface to *notifications*, which asynchronously alert subscribers (called *sinks* in OGSI terminology) of state changes.

Notifications can be of two types: a *push* notification sends information along with the notification, whereas a *pull* notification is used to indicate that something has changed: it is up to the notification subscriber to request (or pull) the information using a regular request. Pull notification gives the subscriber the freedom to decide whether and when to get information associated with the notification, while push notification avoids the overhead of that additional call in situations where the information is needed immediately.

In 2004, the Web Services Resource Framework (WSRF) was proposed to refactor OGSI aimed at exploiting new Web services standards, specifically WS-Addressing. WSRF retains essentially all of the functional capabilities present in OGSI, while changing some of the syntax (e.g., to exploit WS-Addressing) and also adopting a different terminology in its presentation.

WSRF specifications model state as stateful resources and codify the relationship between services and stateful resources in terms of the implied resource

access pattern. They define WS-Resource as a web service associated with a stateful resource.

A stateful resource is defined to:

- have a specific set of state data expressible as an XML document;
- have a well-defined lifecycle;
- be known to, and acted upon, by one or more web services.

Examples of system components that may be modeled as stateful resources are files in a file system, rows in a relational database, and encapsulated objects such as Entity Enterprise Java beans. A stateful resource can also be a collection or group of other stateful resources.

II.D Paxos

At a higher level, Paxos is a protocol for a set of processes to reach consensus [50] on a series of proposals. With a leader election algorithm, a process p elected as leader first carries out the prepare phase of the protocol. In this phase, p sends a *prepare* message to all processes to declare the *ballot number* it uses for its proposals, learns about all the existing proposals, and requests promises that no smaller ballot numbers will be accepted afterwards. The prepare phase is completed once p receives acknowledgements from a majority. If p learned any existing proposals in the prepare phase, p can only propose a new proposal that is consistent with the existing ones.[1] Otherwise, p can propose any value. To have a proposal committed, the leader p initiates the accept phase by sending an *accept* message to all processes with the proposal and the ballot number declared in the prepare phase. The proposal is *chosen* when a majority of the processes accept it. After receiving accept acknowledgments from a majority, p *commits* the proposal and informs all the processes that the proposal has been chosen. Figure II.3 shows

[1] If the ballot numbers of the existing proposals p learned during the prepare phase are not all the same, p only makes its new proposal consistent with the ones with the highest ballot number.

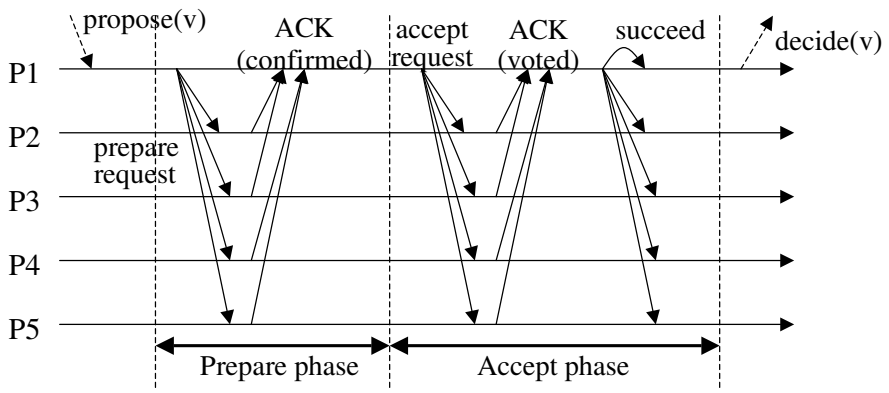


Figure II.3: Paxos

a space-time diagram of an execution of the Paxos protocol. In the later contexts, we use Classic Paxos to refer to this original version of Paxos protocol.

Classic Paxos requires two message delays between when the leader proposes a value and when other processes learn that value has been chosen. However, in most systems that use consensus, values are not picked by the system itself; instead, they come from clients. When the messages from the client are counted, Classic Paxos requires three message delays.

Fast Paxos is an extention of Classic Paxos to achieve the optimal performance in term of message delays. In Fast Paxos, ballot numbers are partitioned into *fast* ballot numbers and *classic* ballot numbers. In a round associated with a fast ballot number, if the leader can pick any proposed value in the accept phase, it sends an *any* message to the acceptors. When an acceptor receives an *any* message in the accept phase, it may treat any cleint's message proposing a value as if it were an ordinary an *accept* message with that value.

A round associated with a classic ballot number works the same as in Classic Paxos. The leader picks the value that the acceptors can vote for, so different acceptors can not vote to accept different values in the same round. This is not the case when a fast ballot number is used. If the leader sends an *any* message in the accept phase, each acceptor independently decides what client message to take as the *accept* message. Different acceptors can therefore vote to accept differ-

ent values when a fast ballot number is used. In case of such a collision happens, the leader needs to take extra steps to coordinate. Therefore, Fast Paxos can not ensure that learning occurs in two message delays in case of collisions.

Fast Paxos requires more acceptors to ensure safety. To tolerate t fault acceptors, Classic Paxos requires $2t + 1$ acceptors and at least $t + 1$ acceptors (a majority) have to accept in order to have a value chosen. To tolerate the same number of fault acceptors, Fast Paxos needs $3t + 1$ acceptors and $2t + 1$ acceptors accepting a value to have it chosen.

Chapter III

Primary-backup on Grid Middlewares

The goal of this chapter is to evaluate the tradeoffs associated with using primary–backup as a fundamental technique for building highly available grid services in the context of OGSI and Globus [27] Toolkit 3 (GT3). Much of this focuses on the tradeoff of performance versus use of facilities provided by the OGSI standard. First, a primary–backup protocol is designed using OGSI to determine whether it supplies the necessary features, such as state update and client rebinding, and to see what changes might be needed to support such an approach. It is found that it is not hard to accommodate primary–backup, and that the solution is simple and requires only small changes to the service. The use of the OGSI notification interface to handle replica updates is the key distinguishing feature of this approach.

Then, this approach is implemented using GT3 to better understand the performance implications and tradeoffs of doing primary–backup at such a high level. In particular, using a simple example grid service, the performance of this notification-based approach is compared to variants in which replica update is done using standard grid service method calls and TCP, respectively. Our example grid service implements a simplified version of the Condor Matchmaker service [49].

Nondeterminism arises in this service both from the way resources are selected and from priorities.

The performance penalty was, in fact, quite high. While some of this may result from the lack of performance tuning in GT3, our findings also have larger implications related to how and where replication should be used to provide fault tolerance in grid service architectures.

Client failures are not considered for this work. One of the attractions of the primary–backup approach is that it defines a very simple client–server protocol that does not depend on clients being reliable. In other words, the correctness of the server, in terms of how it responds to requests, does not depend on help from the clients, which means that client failures can be dealt with using orthogonal approaches such as timeouts and leases [53]. Software bugs that can lead to completely correlated failures are also not considered. In this case, the primary and all backups could simultaneously crash. Again, there are separate techniques that are used in practice for tolerating such failures.

Section III.A introduces the architecture of building the primary-backup services using OGSI notification interface. The implementation issues are discussed in section III.B. Then section III.C represents the performance evaluation. Finally, section III.D summarizes our observation from this work.

III.A Architecture

There are three general problems that any implementation of a primary–backup mechanism needs to solve:

1. *Transfer of application state.* Before replying to the client, the primary needs to send the change in its state to the backups. A reply can be sent to the client only when it is known that the backups will eventually apply the state change.

2. *Detecting failures.* Crashes and lost messages need to be detected. This is normally done by setting a timeout for every message. If no messages are sent for a long time, then a *heartbeat* message can be sent to check on a machine.

3. *Switching to a new primary.* Originally, one of the service instances is designated as a primary and others as backups. After a failure of the primary, the backups agree on a new primary and ensure that all future requests are directed to it.

Grid service notifications are a natural mechanism for solving all these three problems because state updates and failures are inherently asynchronous events. Also, notifications provide a simple mechanism for disseminating information to a number of interested parties—several backups may be interested in the same state update, and several clients may be interested in the same failure notification. Consequently, in our system, backups register with the primary as sinks for state update notifications and heartbeat notifications, and each client registers with each backup as a sink for the failover notification that tells it to switch to a different primary and to resend the last request if it was expecting a reply. We use a push notification for the state transfer because the backup needs every state update. For heartbeat and failover, pull notifications are used because there is no data associated with those events.

The normal execution proceeds as follows. A client makes a grid service request to the primary, which executes the request. When execution ends, the change in the state of the service is extracted and sent to the backups via a notification. When the primary collects acknowledgments from all backups it replies to the client. The state extraction and injection are application-specific: the grid service needs to support methods that allow this to be done. In addition, the service can be designed to have the primary send checkpoints to the backups if its computation is long-running.

Failure of the primary is detected by backups when they do not receive a heartbeat message after a certain period of time. This method allows detection of host and task crashes, as well as network partitions. At that point, the backups need to cooperate in election of the new primary. The newly elected primary then sends a failover notification to the client so it can obtain a new server instance handle. If the client was expecting a reply from the service when a failover notification arrives, then the client resubmits the request to the new service instance. If the old primary had already sent a state update to the new primary, then the new primary can reply with the result computed by the old primary. Otherwise, it can compute the result itself (perhaps starting from a checkpoint if the primary had sent checkpoints to the backups).

Failures of the backups do not interfere with the operation of the surviving system components, so the only new issues are the detection of backup failures and the integration of new backups into the system. Neither is conceptually hard to implement, although integration of a new backup may require a large amount of state to be transferred. The details of how to best do this are outside the scope of this chapter.

III.B Implementation

In this section we give a more detailed overview of our system using pseudocode to illustrate key actions performed by each of the three participants: a client, a primary, and a backup. Each one is enclosed in an object with private variables and methods. Note that we use C language convention for pointers: $\&x$ is a reference to variable x.

The client code, shown in Figure III.1, is interposed between the client application and the original SOAP stub in such a way that client code is not changed. The original stub supports the *init* method, which is called when the client binds to a grid service, and a number of operations, shown here collectively

as *op*. We intercept *init* with the *init_client* method to register for receipt of failover notifications from each backup. The *op* method spawns a separate thread, implemented by *invoke_op*, to invoke the operation via the original stub.

```
var target // points to the current primary
var ops // contains records of on-going operations
INIT_CLIENT ( ) // called when client binds to a Grid service
    (replica_1, replica_2, ...replica_n) ← find_replicas();
    target ← replica_1;
    ops ← ∅;
    for each host ∈ {replica_2...replica_n} do
        register_notification(&FAILURE_HANDLER, host, FAILURE);
    stub.init();
OP (params)
    var op // object for holding operation parameters
    var done // a semaphore for waiting until success
    var result // object for holding results of executions
    op ← { params, &done, &result };
    ops ← ops ∪ &op; // add op to the list of on-going operations
    create_thread(&INVOKE_OP, &op);
    wait_on_semaphore(&done); // wait for someone to succeed in op
    ops ← ops \ &op; // remove op from the list
    return result;
INVOKE_OP (op)
    var result // object for holding results of executions
    result ← stub.op(op.params); // make the SOAP call
    if result ≠ failure then
        op.result ← result; // pass result back to OP()
        signal(op.done); // wake it up
FAILURE_HANDLER (new_primary)
    target ← new_primary;
    for each op ∈ ops do
        create_thread(&INVOKE_OP, op);
```

Figure III.1: Pseudocode for the fault-tolerant stub on the client.

If the primary crashes during this invocation, two things happen in arbitrary order: the call to *stub.op*() returns an error message, and a failure notification arrives from the backup, causing $failure\_handler$ method to execute. To reduce

the failover duration, we don't wait for the *stub.op*() to return (it may take a while for the TCP socket to time out), so we re-submit the request to the new primary in $failure\_handler$ by spawning another *invoke_op* thread. The parameters for this invocation are kept in the list *ops*. Eventually, some invocation should succeed, allowing *op* to wake up and return the result.

The OGSI model specifies a method for clients to deal with failures of grid service instances. Specifically, each grid service has a persistent handle called a GSH (Grid Service Handle) and this handle can be resolved into a handle, called a GSR (Grid Service Reference), for an instance of this grid service. The GSR may become invalid over time and the client can reacquire a valid GSR by re-resolving its GSH. The handle resolution is performed by a grid service called the Handle Resolution Service. Although our design could incorporate this approach, in this chapter we use a design that by-passes the Handle Resolution Service for two reasons:

- The Handle Resolution Service, if not fault-tolerant itself, would provide a single point of failure that could make all grid services that rely on it unavailable.
- In the handle resolution approach, the client only detects the failure of the primary when it attempts to use its GSR. In our approach, the client is notified immediately.

The code on the replicas is interposed between the grid infrastructure and the service implementation. For each client *op* there is an implementation of that operation on the server. To make stateful primary–backup replication possible, the service must implement two additional methods for state transfer: $extract\_state()$ and $inject\_state()$. Ideally, state transfer can be done by a small set of values describing all the relevant application state, but in the extreme it could be a full application checkpoint.

On the primary, as shown in Figure III.2, we intercept each one of the

operations with the *execute* method. It first checks whether this request has already been processed—this can happen when a server crashes after sending the state to the backups, but before replying to the client. In that case the old result is returned without executing the request. Otherwise, the request is executed, followed by the extraction of state, which is sent to backups via notifications. Note that *notify_change_with_ack* blocks until it gets an acknowledgment from every backup. In the initialization routine, the primary advertises itself as a source of two types of notifications (*heartbeat* and *state_update*) and schedules a heartbeat routine to run regularly.

```
var rate_sending // time interval for sending heartbeats
INIT_PRIMARY ( )
    claim_notification_source(HEARTBEAT); // register as source
    claim_notification_source(STATE_UPDATE);
    schedule(&HEARTBEAT_GENERATOR, rate_sending); // run regularly

HEARTBEAT_GENERATOR ( )
    notify_change(HEARTBEAT); // send a notification

EXECUTE (request)
    var result // object for holding results of executions
    result ← check_previous_requests(request);
    // if request is completed result is not NULL, but for new requests it is
    if result = NULL then
        var state // encoding of application state
        result ← service.op(request.params); // execute the request
        state ← service.extract_state(); // obtain state of the application
        notify_change_with_ack(STATE_UPDATE, state); // waits for acks
    return result;
```

Figure III.2: Pseudocode for the primary service.

Figure III.3 shows the pseudocode for backups. During normal operation they receive two kinds of notifications: their *state_handler* receives state updates and injects the state into the service application and their *hb_handler* receives heartbeats. Both store the current timestamp in the global variable *last_notification*. *failure_detector* checks this variable to make sure it is not stale. If it is, then the primary is assumed to have failed.

```
var rate_checking // time interval for checking for notifications
var last_notification // timestamp of the last notification
var primary_is_up // boolean flag
var senior // this is the senior backup

INIT_BACKUP ( )
    (replica_1, replica_2, ...replica_n) ← find_replicas();
    primary_is_up ← TRUE;
    if my_url() = replica_2 then
        senior ← TRUE;
    register_notification(&HB_HANDLER, replica_1, HEARTBEAT); // register sinks w
    primary
    register_notification(&STATE_HANDLER, replica_1, STATE_UPDATE);
    claim_notification_source(FAILURE); // register as a source for clients
    schedule(&FAILURE_DETECTOR, rate_checking);
    SETUP_SENIOR(replica_2);

SETUP_SENIOR (senior_url)
    if senior = TRUE then
        claim_notification_source(HEARTBEAT);
        claim_notification_source(STATE_UPDATE);
    else
        register_notification(&HB_HANDLER, senior_url, HEARTBEAT);
        register_notification(&STATE_HANDLER, senior_url, STATE_UPDATE);

FAILURE_DETECTOR ( )
    if ( current_time() − last_notification ) > rate_checking then
        if senior = TRUE then
            switch_to_primary();
            notify_change(FAILURE); // notify client
        else
            if primary_is_up = TRUE then
                primary_is_up ← FALSE; // wait for the next timeout
            else
                INIT_BACKUP(); // backups re-initialize, electing a new primary
    else
        if primary_is_up = FALSE then
            primary_is_up ← TRUE;
            (replica_1, replica2, ...replica_n) ← find_replicas(); // elect new senior
            SETUP_SENIOR(replica_2);

STATE_HANDLER (state)
    service.inject_state(state);
    last_notification ← current_time();

HB_HANDLER ( )
    last_notification ← current_time();
```

Figure III.3: Pseudocode for the backup service

Switching to a new primary can take a long time because it needs to register as a source of notifications and all backups must re-bind to the new primary. If we delayed client-bound failover notification until re-binding is complete, the failover time of our system would be extremely large (binding can take tens of seconds!). We avoid this performance penalty by binding all backups to one special backup, which we call the *senior* backup, at the time of service initialization.

If a failure is detected, the senior backup becomes the primary and notifies the client immediately, since it already has all backups registered with it to receive state updates and heartbeats. The remaining backups then chose a new senior and bind to it off-line, without delaying processing of client requests. This binding is implemented in the *setup_senior* method, which is called during initialization and during recovery. In the rare situation that the senior backup fails together with the primary, all surviving backups will assume the failure of the senior after the second missing heartbeat and they will go through full re-initialization by calling *init_backup*.

For simplicity, the pseudocode shows that the state is applied immediately by calling *inject_state* in *state_handler*. In a real implementation it it would be better to queue up the state update, send back an acknowledgment and apply the state later, so as to impose as small of a penalty on the response time as possible. As implemented, the protocol queues state updates and applies them later in this way. Doing so can slow down failover because the backup may have to apply queued state messages before processing new requests.

III.C Performance

While it appears that OGSI is a suitable platform for building primary–backup fault tolerance, the overhead of replication may ultimately determine whether the technique is useful in practice. In this section, we describe the performance of our prototype implementation using GT3.

Our example highly available grid service is a simplified version of the well-known Condor Matchmaker service. We measured the transfer overhead, the request response time, and the failure notification overhead of a prototype service structured according to the primary–backup approach described above. We performed experiments on a pair of dual-CPU Pentium II 300MHz workstations with 400Mb of memory, running Linux 2.4. We only considered a system with a primary and one backup, since this is by far the most common way primary–backup is used.

III.C.1 Matchmaker Grid Service

We designed the grid matchmaker service based on existing (but more complex) non-grid services, such as Condor Matchmaker [49], Java Market [2], and the resource management tools in Globus [21]. We chose to use this service because it is an example of an important class of grid services, and because is inherently nondeterministic.

Our Matchmaker service keeps track of machines available in the Grid, accepts requests for allocating machines, and maps each request to a suitable machine. There are two kinds of requests: one is a *resourceAdvertise* request, and the other is a *jobSubmit* request. A resourceAdvertise request provides information about a machine that is available for allocation. The input of this request is: the resource ID, the available CPU speed, the available memory size, the available disk size, the machine's IP address, and an identification string used to implement a simple capability for using the machine. A jobSubmit request sends a specification for a desired machine. If there are suitable machines available, then the Matchmaker service will choose one and send the address of this machine and the identification string back to the client. The input of this request is: the job ID, the required CPU speed, the required memory size, the required disk size, the priority of the job. The response of this request is the address and identity of the chosen machine, if there is one available; otherwise, the request returns a null string.

Table III.1: State transfer round-trip time (milliseconds)

		10 B	100 B	1 kB	10 kB	100 kB
Notification	Median	192.5	189.0	193.0	195.5	190.0
	Mean	201.3	191.8	196.7	199.8	192.7
	St. Dev.	29.8	13.0	15.5	23.6	14.8
Call	Median	19.5	19.0	26.5	30.0	209.0
	Mean	26.4	24.2	33.1	32.6	299.0
	St. Dev.	12.0	9.2	17.9	6.3	333.0
Socket	Median	1.0	2.0	2.5	12.0	133.0
	Mean	1.5	1.7	2.5	14.8	144.5
	St. Dev.	0.8	0.5	0.5	11.9	32.4

This Matchmaker service is non-deterministic for two reasons. First, if there are several machines that satisfy a jobSubmit request, then the machine that is allocated can be nondeterministically chosen. Second, the Matchmaker service is implemented by two threads: one enqueues requests and one executes enqueued requests. Requests are enqueued in priority order, and is FIFO within each priority. Two servers S_1 and S_2 could behave differently because of these rules on priority as described in chapter I.

III.C.2 State transfer

To fully understand the sources of overhead in state transfer, we compared the implementation using OGSI notifications for state updates (labeled *Notification*) to two alternative implementations, one that uses direct grid service method calls (labeled *Call*) and one that uses TCP connections (labeled *Socket*). In the following tables we present the median, the mean, and the standard deviation for a set of 20 round-trip measurements. To better understand the overhead of state updates, our service can be configured to send an arbitrary amount of data in each state update.

First, Table III.1 shows the round-trip time of a single state update, for a number of different state sizes (from 10 bytes to 100 kilobytes), as measured on

Table III.2: Client request round-trip time (milliseconds)

		10 B	100 B	1 kB	10 kB	100 kB
Notification	Median	242.0	241.0	240.0	238.5	232.0
	Mean	252.3	247.4	251.2	301.1	241.2
	St. Dev.	41.9	36.5	36.6	257.9	34.2
Call	Median	65.0	72.0	76.5	74.0	261.0
	Mean	71.4	78.0	89.7	82.5	350.8
	St. Dev.	21.9	28.9	40.9	21.9	333.5
Socket	Median	45.5	47.0	48.5	55.5	182.0
	Mean	52.8	56.5	55.5	62.6	195.9
	St. Dev.	21.6	26.6	21.8	22.9	50.4

the primary. Not surprisingly, *Socket* always has the smallest round-trip time, but this advantage goes from around 200 times faster for 10B updates to only 1.4 times faster when the state size is 100 kB. *Call* has intermediate round-trip times: at 10B, it is about 20 times slower than *Socket*, while at 10 kB it is only 2.5 times slower.

Note that the round-trip times for *Notification* are mostly insensitive to the size of the state update. Essentially, the cost of sending 10 bytes and 100 kilobytes with a notification is roughly the same. We think the cause of this lies in the format of GT3 notification messages; this is something that might be worth examining for later versions of the toolkit.

We also observed very high variance in samples: the standard deviation is sometimes higher than 50% and in one case is larger than the mean. This last case is due to a single outlier in the *Call* experiment for 100 kB of state update, which took 1.7 seconds. We think that much of this large variance is an artifact of Java garbage collection or other background processing in the Java virtual machine or the grid container.

Table III.2 shows round-trip times of client requests during normal, failure-free operation, as measured by the client. We would expect these numbers to be, approximately, the sum of the request round-trip time without primary–backup

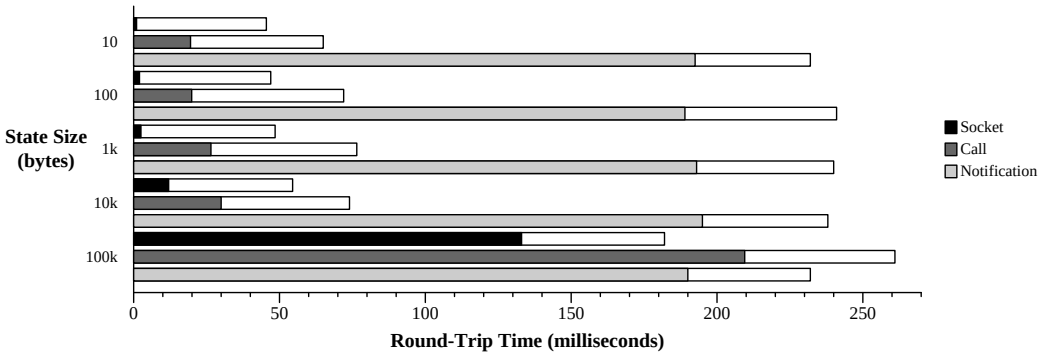


Figure III.4: Median Round-trip times for state transfer using *Socket*, *Call* and *Notification* with different state sizes. The white portion is the round-trip time of a client request without replication. Therefore, the overall size of the each bar is the total client round-trip time.

Table III.3: Ratio of client request round-trip with primary-backup to median/mean client round-trip without replication

		10 B	100 B	1 kB	10 kB	100 kB
Notification	Median	5.5	5.5	5.5	5.4	5.3
	Mean	4.6	4.6	4.6	5.5	4.4
Call	Median	1.5	1.6	1.7	1.7	5.9
	Mean	1.3	1.4	1.7	1.5	6.5
Socket	Median	1.0	1.1	1.1	1.3	4.1
	Mean	1.0	1.0	1.0	1.2	3.6

replication plus the state update overhead shown in Table III.1 above. This is, indeed, the case since the request round-trip time of a normal grid service, without replication, is 54.3 ms on average with the median being 44 ms. The data from the previous two tables is summarized graphically in Figure III.4, where client request round-trip time is broken down into interaction between the client and the primary (white) and the interaction between the primary and the backup (solid, upward diagonal, and downward diagonal).

In Table III.3, we normalized the data of Table III.2 by dividing the median and mean numbers by the median and mean of the normal grid service

Table III.4: Failover duration (milliseconds)

	1	2	4	8	16
Median	189	215	524	896	1989
Mean	194	302	547	1018	2269
St. Dev.	21	352	83	454	666

round-trip. So, each number shows the magnitude of overhead imposed by replication. The table shows that with *Socket* the median overhead of replication is small: for small state sizes (up to 10 kB) is 30% or less. With *Call*, the median overhead is 70% or less for small state sizes. For large state sizes all approaches perform similarly, with overheads of 400% and more.

From these results, we conclude that notifications are considerably less efficient than socket messages and service calls for small state sizes. For larger state sizes all of the three approaches impose a high overhead. Note that in all cases the requests have very low overheads. In this situation a request that used to take 44 ms ends up taking between 4 and 6.5 times as long with replication. For grid services that have longer-running requests, the overhead of replication will be diminished. For example, for a request that takes 3 seconds to execute and has state size of 100 kB, the overhead of replication is less than 10%. Hence, the drawback of using GT3 to implement primary–backup becomes negligible for long-running requests.

III.C.3 Failover

Another important metric for the performance of a fault-tolerant system is failover duration. This is a sum of two quantities: the time it takes for the backup to detect the failure, and the time it takes for the backup to notify the clients of a failover. The first quantity depends on the frequency of heartbeat messages and is largely independent of the implementation. Therefore, we only measure the second quantity, as shown in Table III.4.

With one client, it takes 194 ms on average to notify the client of a failure. As the number of clients increases, the notification overhead increases linearly. In [60] we report that recovering a network connection endpoint in less than 200 ms requires significant investment in equipment for logging of packets that may be lost due to the failure, and so we believe that 194 ms is quite acceptable. If the number of clients that shares the same instance is large, however, then the overhead may become too large. Again, these results were obtained based the current implementation of GT3. A later version may be able to have notifications run faster than linear in the number of sinks.

Note that if the client doesn't have outstanding requests to the primary service when the failure happens, then the overhead of the failover at client is almost zero, since the client only needs to change the address of the service invoked.

III.D Summary

Fault tolerance of stateful grid services is becoming increasingly important with the development and use of OGSI. Both infrastructure services such as monitoring, resource allocation, and scheduling, as well as grid applications implemented as grid services, are required to be reliable and highly available. In this chapter, it is shown that the facilities defined in OGSI and the newly proposed WS-Notification extension to Web services [29] can be used to design a primary–backup service. While not described in this chapter, this service can be easily extended to multiple backups and to dynamically adding backups. In addition, by using slightly modified client stubs, failover can be done transparently to clients.

We found the overhead of using GT3 implementation of the OGSI notification to be quite high. The overhead is particularly large in the cases where the state data is small or the number of clients is large. Much of the overhead seems to come from the cost of notifications, which can most likely be improved in future implementations of GT3. Failing that, one might wish to provide state update

below the OGSI level or by using simpler OGSI facilities such as basic grid service method calls. It might be possible to improve performance of primary–backup by using an alternate protocol binding—something that is specified in OGSI but not available in GT3—but we have not explored this option in any detail.

The approach for primary–backup is only applicable for replicas located in a cluster, since otherwise failure detection becomes too unreliable for primary–backup. The protocols for asynchronous systems will be discussed in Chapter V and Chapter VI.

III.E Acknowledgements

This chapter is, in part, reprints of material as it appears in "Fault-tolerant Grid Services Using Primary-Backup: Feasibility and Performance," by Xianan Zhang, Dmitrii Zagorodnov, Matti Hiltunen, Keith Marzullo, and Richard D. Schlichting, in the Proceedings of the 2004 IEEE International Conference on Cluster Computing, September 2004. The dissertation author was the primary coauthor and co-investigator of this paper.

Chapter IV

Durability – a Service Attribute

A key requirement for implementing highly available services is the ability to maintain service state across machine failures and server process crashes. This problem has been extensively studied and many techniques exist for protecting service state, including storing the state in a database or maintaining it in replicated processes. Each such technique can be characterized based on its level of protection and its cost in terms of hardware resources required and performance overhead. While many such techniques exist, service developers typically have a very small set of options (e.g., a database or nothing) and the chosen option must be coded in the service implementation (e.g., by denoting objects as session or entity beans in J2EE [10]). Specifically, there is no integrated and transparent way to use different techniques to protect service state. However, different types of service state differ in the degree of protection required; some types such as billing information needs to survive very severe failures, while others might be reconstructed relatively easily should a failure occur. Furthermore, the business or cost requirements associated with the service may change over time, resulting in large code rewrites.

This chapter introduces an architecture that allows service developers to use different techniques with different tradeoffs to protect service state flexibly and transparently. The service state is stored in one or more *state objects*. We propose to treat *state durability*—that is, the likelihood that the state can survive failures—

as an explicit design parameter that is associated with each state objects used by a service. Based on this durability attribute, different techniques with different tradeoffs can then be used to ensure the durability of different state objects. For example, the value of one state object can be stored in a database, while another is replicated in-memory on two or more computers. Note that one can view the different techniques as implementations of the stable storage abstraction [39], but with an explicit recognition of and control over the tradeoff between the fidelity of the abstraction and the cost of implementing it.

We have built a prototype based on Globus Toolkit 4 (GT4) using Java. In this prototype, service designers can choose for each state object between distinct points on the durability-performance continuum, including implementing durability using primary-backup replication or implementing it using a database. Different choices can be made for different service state objects and the choice has no impact on the way in which the state object is used by services. To illustrate the way in which the architecture can be used, we describe a prototype implementation of a highly available Matchmaker service.

IV.A States and Durability

This section discusses the role of state in SOAs and defines the concept of state durability.

IV.A.1 State in SOAs

There are many aspects to providing reliable and highly available services. Specifically, the service architecture must be able to detect failures of services, restart failed services, and reconnect clients (which may be end-applications or other services) with the recovered or relocated service instances. A key factor in providing such services is maintaining the service state through failures. If the service state can survive the failure of the server processes that provide the

service, the service can be restarted by simply starting a new server process with the preserved service state.

The type of service state depends on the specifics of the service as well as the middleware platform used to implement the services and service interactions. Services may be completely stateless, that is, all the state required to process a service request is included in a request (e.g., an encryption service that receives the data and key in the request). Otherwise, the service may have:

- **Internal state.** State that the service reads or updates when processing a request, e.g., an inventory database.
- **Session state.** State associated with the interaction between the service and (typically) one client, e.g., a shopping cart.

Distributed object platforms such as CORBA and DCOM support both internal and session state maintained in the object instance accessed by the client. Traditional web services do not explicitly support any state and are therefore often considered stateless for this reason. However, many web service applications require state that is maintained across client requests and most web service platforms support storing important state in a database (either explicitly like in ASP.NET or implicitly like in J2EE entity beans).

The first generation of grid services as defined by the OGSI standard supported both session and internal state. Conceptually, these grid services were very similar to distributed object systems with the concept of creating grid service instances that maintain state. The new WSRF [19] replaces OGSI and allows stateful grid services to be built directly on the web services foundation. WSRF defines a framework for managing state in distributed systems using services. Such state is modeled as a WS-Resource that provides a web service interface for manipulating a stateful resource (e.g., a database table, file, J2EE entity bean).

IV.A.2 State Durability

We use the term *durability* to describe the attribute of service state that describes its resilience against hardware and software failures, that is, durability is the probability that the state will persist for a specified length of time despite failures. While other dependability concepts such as reliability and availability [6] are attributes of measuring the overall service, durability has its focus on service state, which is often the key factor to provide the dependability of the overall service. Analogously to reliability or availability, the state durability depends on the techniques used and the numbers, types, and frequency of failures that occur in the underlying resources during the lifetime of the service state. Similar to these two other metrics, it is impossible to provide 100% durability but it can be be increased to be arbitrarily close to 100% by using the appropriate techniques.

There are a number of techniques that can be used to increase the durability of a state object. For example, the state object may be replicated on multiple processors, stored in a file, or stored in a database. Note that not all file systems and databases provide the same degree of durability. For example, a file system that uses redundant disks (e.g., RAID) provides higher durability than a normal file system that relies on a single hardware disk, and a replicated database typically provides a higher durability than a non-replicated database. The cost of providing durability depends on the technique and implementation platform chosen. This cost includes both of the runtime performance overhead and the cost of the hardware and software. Furthermore, the different durability techniques may have different recovery time (MTTR - mean time to repair) and thus, affect the overall availability of the service.

The state in different services, and different types of state within a service, have different durability requirements. For example, an e-commerce service maintains inventory information, information about the regular customers (e.g., address, credit card number, preferences), and state about on-going customer interactions ("shopping carts"). The inventory information is most valuable because

its loss would prevent the service from operating, while the loss of the information about on-going customer interactions would be a nuisance for the users, but would not stop the service. Note, however, that even such issue may be enough to cause users to switch to a competing service, resulting in loss of revenue.

Note that the durability concept is more general than the typical classification of state as soft or hard state. *Soft state* is often defined as state that can be reconstructed automatically given enough time and/or work (computation), while *hard state* is typically defined as state that cannot be reconstructed automatically if lost due to a failure. Note, however, that whether the data can be automatically reconstructed does not necessarily reflect the true value of a state. For example, a shopping cart is hard state but not extremely high value, while a soft state such as a simulation result may have a high value if its reconstruction takes days of CPU time on hundreds of computers. Therefore, the concept of state durability allows the durability techniques to be tailored based purely on the value of the state. This realization that all the state is not the same can result in performance improvements, cost reduction, as well as an increase in service availability.

IV.B Customizable Transparent Durability

The goal of this work is to provide services with customizable availability, durability, performance, and resource cost tradeoffs. Specifically, our approach allows the service state to be made durable by applying different durability techniques transparently and automatically to different state objects, as independently as possible from the semantics and representation of the state. By doing so, we provide *durability transparency* to the service: the desired mechanisms can be instantiated when the service is built or configured. Furthermore, given such durable state, we show how highly available services can be constructed using standard services.

IV.B.1 Assumptions

We assume that a service is implemented as a Java class and each state object used by the service is implemented as a separate Java class. Thus, all state access operations by the service are method calls to the Java objects implementing the state. Each method exported by the service may read and update one or more of the state objects, one or more times.

As an example, consider an on-line bagel shop service that allows clients to order bagels by issuing service requests. Examples of state objects in this example could be the store inventory (e.g., how many bagels of each kind are available) and customer information (e.g., delivery address and credit card information). The service can be implemented as a stateless Java class that takes the request, checks the inventory and reserves bagels for the customer, issues a credit card transaction to the customer's bank (using a service request), and if everything succeeds, schedules delivery to the customer's delivery address. Standard transactional techniques can be used to ensure bagels are delivered if and only if the credit card transaction succeeds. Other service operations are available for registering a new customer, querying price of bagels, etc.

In this chapter, we consider crash failures of the machines on which the service is running, as well as crash failures of the container process that hosts the service. Specifically, we assume that all state maintained in the memory will be lost if a failure occurs. Furthermore, we assume a failure can be detected reliably (i.e., the availability of perfect failure detector).

IV.B.2 Durability Mechanisms

In this chapter, we consider three mechanisms with different performance-durability tradeoffs:

- **Database.** State object is backed up in a database.

- **Replication.** State object is backed up in a backup replica(s) of the state object on another machine(s).
- **Reconstruction.** State object is stored in volatile memory only without any backup, and then reconstructed in case of failure.

We chose these three different mechanisms because their implementations are quite different: accommodating them in a way that is transparent to the service is a challenge. Also, they provide different performance, durability, availability, and resource cost tradeoffs. Note, however, that a specific mechanism can provide a range of possible tradeoff points depending on factors such as number of replicas used (for replication) or the type of database (and the underlying file system, OS, and hardware) used. For example, an in-memory database implementation uses the database interface, but does not provide any additional increase in durability.

Many other mechanisms with different performance-durability tradeoffs, such as client-side caching, files, and state machine replication, resemble in some ways resources constructed using one of these approaches.

IV.B.3 Challenges

There are many challenges in providing such customizable transparent durability.

- **Internal transparency.** How to add state durability mechanisms to an existing service without requiring manual modification of the service code (including the state object code).
- **External transparency.** How to ensure that the different durability mechanisms used within a service do not change the external behavior of the service.
- **State update and restoration.** The different durability mechanisms use very different representations of the state, and thus, the operations for updating the state are very different. The database solution requires the state to be

converted into database tables and state update operations into updates of the tables. A trivial solution is to use Java serialization to serialize the state object and store this serialized form in the database. However, this solution can be very expensive if the state is large. Since the replication approach replicates the whole state object, it may be possible to execute the same operations on the backup (in the same order) as are executed on the original object (primary). However, if the update operations are nondeterministic, this simple solution cannot be used.

- **Atomicity w.r.t. failures.** Since a service request may update multiple state objects (or a single object more than once), we may want to ensure all-or-nothing semantics in the case of the service failure in the middle of processing a request. That is, if the service fails in the middle of the request, the impact on the state objects is as if the request was never executed. Given such semantics, the client can simply reissue to request with no undesirable side effects. While database transactions provide such atomicity for the the database-based solution, similar guarantees must be provided for the replication approach.

IV.C Solution

This section describes our approach in detail and explains how the challenges above can be addressed.

IV.C.1 Overview

Our solution is based on the following key concepts:

- **Durability proxies** that implement a durability mechanism in a generic, state independent, manner.
- **Durability mapping** that specifies which durability mechanism is to be used for each state object, as well as any needed object specific instructions.

- **Durability compiler** that takes the service and state object code, the durability mapping, and the necessary durability proxies and generates a service and the associated state objects where the desired durability mechanisms are used for each state object.

An example of a service transformed by the durability compiler is shown in Figure IV.1. In this example, a service uses two state objects, one of which uses the database (via the database proxy) and the other object replication (via replication proxies). Note that the backup state replica is maintained inside a backup service, but this backup service only maintains the state object and does not serve clients' service requests, unless the primary service fails.

If the (primary) service fails, a backup service takes over the processing of client requests. Specifically, the recovery operation in the database proxy retrieves the object state from the database. Since the replication proxy at the backup already maintains the current object state, it simply changes its role to be the primary replication proxy and a new backup service is created, if necessary. If no backup service exists when the service fails, a new service instance can be created and the chosen proxies restores the service state before the service starts serving client requests.

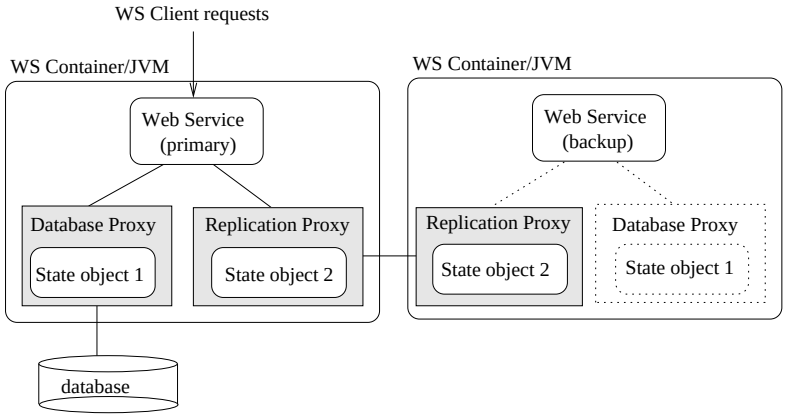


Figure IV.1: Example Architecture

The service client may use some form of service transactions—defined by the WS-Transactions specifications including WS-Coordination [13], WS-AtomicTransaction [14], and WS-BusinessActivity [12]—to access the service and the service may participate in WS-Transactions spanning multiple other services. When transactions are used, the usual transactional guarantees are provided by our customized service with regard to atomicity and serializability. When WS-Transactions are not used, our service ensures semantic durability and consistency of any state changes. Specifically, this means that if a client executes a service request that updates the state of the service and the client receives a response indicating success, this change remains in effect even if the original service instance subsequently fails. Thus, if the client issues a subsequent service request that is served by a backup service, the state of this service will be consistent with the client expectation, that is, as if no failure had occurred. If a service crashes in the middle of a service request, our solution ensures that any partial changes made as a part of this request are rolled back.

We assume that the service client detects the failure of the service and reissues the failed request. The failure of the service is detected when the request times out and the underlying HTTP layer returns an error. The fact that the service has moved to another machine can be hidden using standard load balancing techniques that virtualize the physical address of the machine providing the service. Alternatively, the client may need to relocate the service using facilities based on the proposed WS-RenewableReference specification.

This chapter does not address the specific details of how failure(s) of the service instance(s) are detected or which component in the system is responsible for starting a new instance of the service or notifying the backup to change its role. There are no established services standards for such services yet, although the work on services and grid computing standards at OASIS (Organization for the Advancement of Structured Information Standards), DMTF (Distributed Management Task Force), and GGF (Global Grid Forum) may provide appropriate

standards in the near future. Without such standards, each SOA environment provides its own proprietary facilities. Typical monitoring systems for distributed systems (e.g., HP Openview, BigBrother [47]) allow scripts to be specified to be executed when a specific event (e.g., failure of the primary service) occurs and such scripts can be used to start or activate a service. Alternatively, systems specifically geared for automatic recovery through restart can be used [32]. Finally, the backup service can be programmed to be activated if it receives a client request directly, assuming that the backup service only receives requests when the original service has failed.

IV.C.2 Durability Proxies

The durability proxies "wrap" state objects to make their state durable. The wrapping is implemented using the Java reflection API—Java Dynamic Proxy classes. Java Dynamic Proxies allow a method call to the object proxied to be intercepted and customized behaviors to be added before and after the actual object method call is completed.

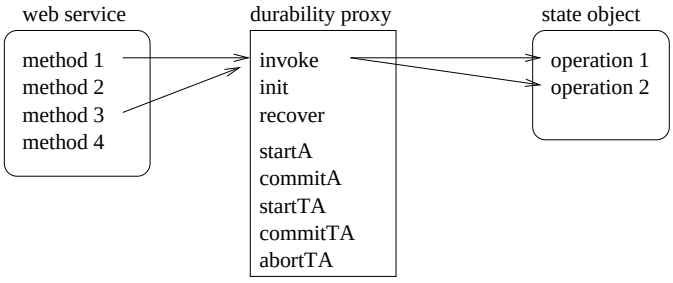


Figure IV.2: Durability proxy interface

Figure IV.2 illustrates the interface provided by a durability proxy. Specifically, each durability proxy provides an *invoke* operation that is invoked every time an operation on the proxied state object is invoked. If this operation updates the state object, the durability proxy saves the new state using its durability mechanism (e.g., database). Furthermore, each durability proxy provides *init* and *recovery* methods that are used to initialize the object state at startup and to re-

store the object state in case the original service fails, respectively. For example, at initialization, a database proxy creates the database tables used for storing the object state and at recovery, the proxy at the backup or newly created service instance loads the object from the database.

Finally, a durability proxy also implements five *action control* methods that the (modified) service calls directly to ensure the desired atomicity properties for the service request. These operations are used to ensure the desired atomicity for the service method execution with and without WS Transactions:

- *startA:* start of a new service request (atomic action),
- *commitA:* complete (commit) the state changes made during the execution of this service request (complete atomic action),
- *startTA:* start service transaction,
- *commitTA:* commit state changes made during the current service transaction,
- *abortTA:* abort (undo) any state changes made during the current service transaction.

The durability compiler modifies the service code so that it issues calls to these proxy methods when appropriate, see section IV.C.4. Specifically *startA* is called at the beginning of the service request and *commitA* at the end of the service request for all the state objects for which the update operations are called during the processing of the service request. The *startTA*, *commitTA*, and *abortTA* operations are called by the service when it participates in the WS transaction coordination protocols. Note that when a service method is called inside a WS transaction, the *startA* and *commitA* operations have no impact since the state changes cannot be completed until the WS Transaction commits (or aborts). The durability proxy simply ignores these operations when it knows that the state object is involved in a WS Transaction.

Since it is impossible to implement efficient generic durability proxies, some of the state object-specific instructions used by the durability proxies are specified in the durability mapping, see section IV.C.3. The durability compiler combines the generic durability proxy code with the information specified in the durability mapping to generate state object specific durability proxies.

Replication proxy

As illustrated in figure IV.1, the replication proxy replicates the state object by maintaining backup copies of the object in backup copies of the service. The replication proxy at the service acts in the role of the primary proxy and the replication proxies at the backup services act in the role of backup proxies. The replication proxy implements the basic proxy API as follows:

- *init*: create a TCP connection between the primary and backup proxies.
- *recover*: (backup) change role to primary and create TCP connections to new backup proxies.
- *startA*: (primary) create an empty update buffer at the primary.
- *invoke*: (primary) if the operation issued is an update operation, forward the request to the local state object and store the corresponding state update in the update buffer.
- *commitA*: (primary) send update buffer to backup(s), wait for acknowledgment. When a backup proxy receives this state update, it sends back an acknowledgment to the primary proxy. Note that the backup can issue the state updates after sending this acknowledgment.

Implementing transactional semantics involves deciding whether the failure of a service instance (primary or any of the backups) should be masked from the WS transaction or whether the WS transaction should be aborted if one of the service instance fails during the transaction. While doing the first would provide

failure masking and slightly better service reliability, it would require changes to the WS Transaction specifications. As WS-Transaction is currently defined, a service automatically joins a transaction when it receives a service request containing a new transaction object. If this service does not respond during the two-phase commit protocol, then the decision will eventually be to abort since the service will not be around to vote *yes* to commit the transaction. Hence, one would need to have a way to de-register the failed service with the transaction (which would be a change to WS-Coordination) and to allow the backup (or newly started service instance) to join the transaction. Another alternative would be to migrate the identity of the registered service from the failed service to the backup. Doing the latter would require some support for migration of identities.

The second approach can easily be implemented with the current WS-Transactions standards and is thus used by our current replication proxy. In this case, if a service instance fails during the transaction, the transaction coordinators will eventually abort the transaction. After the service has recovered, the client can reissue a transaction request similar to the case without transactions. When a transaction starts, the service invokes the *startTA* method on the replication proxy and when the transaction commits or aborts, the service invokes the *commitTA* or *abortTA* operations, respectively. The transaction context is passed as an argument to the *startTA* operation. The implementations of these operations are as follows:

- *startTA*: (primary) checkpoint current state (see durability mapping below); create update buffer; send transaction context to the backup(s), wait for acknowledgment(s); (backup) register as a participant in the transaction.
- *commitTA*: (primary) send update buffer to backup(s); wait for acknowledgement(s); discard checkpoint; (backup) apply state updates sent by primary; send an acknowledgement.
- *abortTA*: roll back state, discard update buffer.

Database proxy

As illustrated in figure IV.1, the database proxy backs up the object state in a database. When an update operation is invoked on the state object, the database proxy updates the object state in the database by issuing appropriate database operations. The *init* method creates the database tables necessary for storing the object state and the *recover* method at the database proxy at a backup (or newly created) service issues the appropriate database query operations to load the current object state.

The action control operations in this case simply rely on the transactional support provided by the underlying database system. Both the atomic actions (defined by *startA* and *commitA* methods) and WS Transactions (defined by *startTA*, *commitTA*, and *abortTA*) are implemented by calling the methods for starting, committing, and aborting a transaction provided by the database. For example, if the Java Transaction API (JTA) is used, the operations could be `TransactionManager.begin`, `TransactionManager.commit`, and `TransactionManager.rollback`. Note that the *startA* and *commitA* operations check to see if the state object is already involved in a WS Transaction and if it is, they simply return without invoking any database operations.

When the state object is involved in a WS Transaction, it must be able to undo any changes made in case the transaction is aborted. Similar to the replication proxy case, the object state can be checkpointed at the beginning of the transaction (*startTA* operation); however, in this case the state can also be restored by simply using the *recover* operation that pulls the state from the database (after the transaction has been aborted). These operations can be specified in the durability mapping.

Reconstruction proxy

The reconstruction proxy is appropriate in situations where semantically durable state can be implemented by reconstructing the object state after failure. The *init* operation simply creates an "empty" state object (the definition of "empty" is object specific), while the *recover* operation invokes the *init* opera-

tion. The *invoke* operation forwards operations to the state object and *startA* and *commitA* operations do nothing (i.e., return).

To implement transactional semantics—particularly, to be able to abort a transaction—even this proxy has to implement the *startTA*, *commitTA*, and *abortTA* operations. A checkpointing strategy similar to the replication proxy can be used, or alternatively, if aborts are unlikely, it would be semantically valid to abort to an "empty" state (i.e., *abortTA* simply calls *init*).

IV.C.3 Durability Mapping

The durability mapping describes for each state object the durability mechanism chosen for this object and the state object specific instructions. These specific instructions may be required for the following operations:

- Initialization and recovery,
- for each update method of this object, the state update instructions,
- checkpointing and restoring instructions.

While Java object serialization can be used to implement both the state update and checkpointing, more efficient methods are often available depending on the state object semantics. For example, if the state object operations are deterministic, the state update in the case of a replication proxy can be achieved by simply executing the same operation on the object replica(s). For example, for a state object that uses the database proxy, the durability mapping could include the following:

- Initialization: Create database table(s) for the state.
- Recovery: Database instructions to query database for the object state and instructions to assign the state.
- State update(s): Database instructions to update a table based on operation parameters.

- Checkpointing: mapping specifies a no op.
- Restoring: Call the *recover* operation.

IV.C.4 Durability Compiler

The durability compiler takes the service code (including the code for the state objects) and the durability mapping, and generates modified service code and customized durability proxies for the state objects. The durability proxies are generated by taking the generic proxy code and inserting the required customization code from the durability mapping; see section IV.D for an example. The output of the durability compiler is normal Java code that can be compiled and executed as a normal Java program.

The service code is modified as follows. The initialization code of the service is extended to create the durability proxies for each state object. All method calls to the state objects are replaced with calls to the corresponding durability proxy. For each method of the service, the durability compiler inserts calls to the state objects' *startA* methods at the beginning and *commitA* methods at the end for all state objects that may be updated by the service request. Note that the code for the backup service is the same as for the service except that it creates any replication proxies in the backup role and it does not call the *init* methods of the database proxies. It also provides a method that can be invoked to force the backup to assume the role of the primary service. Some additional code to interact with the system monitoring and registry system may also be necessary at the service and in the backups depending on the specifics of the system.

IV.D Examples

We use two services to illustrate our approach and evaluate the performance overhead imposed by the different durability mechanisms. The first service is the Counter service that is included as an example in the Globus Toolkit soft-

ware and is built using WSRF. We use this simple service for micro-benchmarking throughput and latency. The second service is the Matchmaker service again, with slightly different functions as the one we use in chapter III. We use this Matchmaker service to experiment with the performance tradeoffs of services using more than one state object.

Our performance evaluation is based on Globus Toolkit 4. Our configuration consisted of three dual-CPU Pentium III 2x2785 Mhz workstations connected to a 1 Gbit Ethernet and running RedHat 9. One machine ran the clients, a second machine ran the server, and the third machine hosted the database server and was used to run the service backup. We used MySQL 4.0 as our database servers. The durability compiler has not been implemented; the corresponding transformations were performed manually. Furthermore, since the Globus Toolkit does not yet provide support for WS Transactions, we did not implement transaction support.

IV.D.1 The Counter Service

The Counter service uses WSRF to maintain stateful information. This service has one operation "add", which increments the value of the counter state by one and returns the incremented value. The state object, Counter, provides two operations: "setValue" and "getValue", and only "setValue" updates its state. We compared the performance of three service configurations: the original Counter service, the service with the Counter object protected using the replication proxy (PB service), and the service with the Counter protected using the database proxy (DB service). Since the reconstruction proxy does not perform any operations except when failures occur or when the state object is involved in a transaction (neither of which were present in the experiments), its performance would be identical to the original Counter service. Our test client sends a specified number of "add" requests sequentially to the service. For the latency tests, we used one client that sends 40,000 requests, while for the throughput tests we used c concurrent clients, $c \in \{1, 2, 4, 8, 12, 16\}$, where each client sends $40000/c$ requests.

The average request round-trip time (RTT) of the original service was 139 ms (with 95% confidence interval ±0.3 ms). The RTT of the PB service is essentially the same, while the average RTT of the DB service is 171 ms (with 95% confidence interval ±0.3 ms), which is an increase of 23%. Figure IV.3 shows the throughput of the three services. The throughput of the PB service is close to the throughput of the original counter service, especially when the number of clients is small. The throughput of the DB service is lower than the other two. For all three services, the throughput increases when the number of clients increases from 1 to 8, and does not change much when the number of clients increases from 8 to 16. The 95% confidence intervals of all the throughput values are no wider than ±0.11. Thus, the confidence intervals would not be readable in the figure.

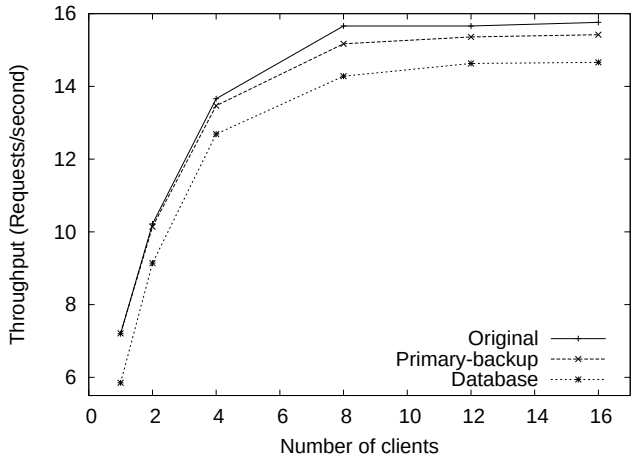


Figure IV.3: Counter Service Throughput

IV.D.2 The Matchmaker Service

The Matchmaker service is similar as the one we use in chapter III. It keeps track of machines available in a grid and allocates them to clients requesting computing resources. The service provides two methods: "machineAdvertise" and "jobSubmit". Each machine periodically advertises its availability by invoking "machineAdvertise" with information about the resources it has available (e.g.,

CPU speed, available memory, available disk, etc). A client that requires a machine invokes "jobSubmit" with requirements for the desired machine. If suitable machines are available, Matchmaker choose one and returns it to the client. Otherwise, it returns an error message indicating that machines of the requested type are currently not available. The scheduling policy of this Matchmaker service is different from the one in chapter III. If there are more than one machines suitable for a "jobSubmit" request, the Matchmaker service returns the first one it finds. After receiving a "jobSubmit" request, the Matchmaker exemines the resource pool and responds to the client immediately, instead of putting the request into a queue.

The Matchmaker service contains two state objects: MachineQueue and AccountSet. MachineQueue maintains a list of available machines and AccountSet maintains usage records for each client used to charge the user for the resources. If the MachineQueue is lost, it can be reconstructed based on the periodic "machineAdvertise" calls but there is no way to reconstruct AccountSet if it is lost. Thus, these two state objects have different durability requirements. Note that "machineAdvertise" updates the MachineQueue, while "jobSubmit" may update both state objects.

Tables IV.4 and IV.5 illustrate the durability mappings for MachineQueue and AccountSet. For MachineQueue, the mapping specifies that a replication proxy should be used. The state update method is to execute the same method (with the same arguments) on the backup state object (keyword **repeat**). No special initialization or recovery code is needed in this case and we omit checkpointing and restoration instructions. For AccountSet, this mapping specifies that a database proxy should be used. The initialization section specifies the instructions for creating a database table for the object state. The recovery section specifies the intructions for retrieving the object state from the database. The state update methods are based on the arguments of the state object operations and map to simple database update operations.

We compared the performance of four service configurations: the original

```
<durability proxy> <type> Replication </type>
  <update methods>
    <method>
      <name = insertNode>
      <state update> repeat </state update>
    </method>
    <method>
      <name = deleteNode>
      <state update> repeat </state update>
    </method>
  </update methods>
</durability proxy>
```

Figure IV.4: Durability mapping for MachineQueue

Matchmaker service, one using replication proxies for both states objects(PB), one using a replication proxy for MachineQueue and a database proxy for AccountSet (PB+DB), and one using database proxies for both state objects (DB). Our experiment has one client first advertise a large enough number of machines that all subsequent "jobSubmit" calls can be satisfied. Then, four clients generate load for the Matchmaker service. Each client executes the following sequence of actions 30 times: (1) pick a random number n between 1 and 5; (2) send MatchMaker n requests, with each request asking for one machine; (3) pick a random number C from a specified "computation interval"; (4) sleep for C seconds. The sleep period simulates the time during which the client is using the allocated machines.

The results are shown in figure IV.6. As expected, the average latency of PB is the closest to the original service: it is about 60 ms less than DB. The performance of PB+DB is between PB and DB. These results indicate that by using a cheaper durability mechanism for the less critical state object, the overall performance of the service can be improved compared to using a database for both. When the computation time interval increases, the average latency decreases, since the chance that two clients invoke the Matchmaker service at the same time is

```
<durability proxy> <type> Database </type>
  <initialization>
    CREATE TABLE bills (clientID INT, balance INT) ENGINE = INNODB;
  </initialization>
  <recovery>
    SELECT * FROM bills;
    for (each line) insertBill(clientID, balance);
  </recovery>
  <update methods>
    <method>
      <name = insertBill>
      <state update>
          INSERT INTO bills VALUES (arg[0], arg[1]);
      </state update>
    </method>
    <method>
      <name = setBill>
      <state update>
          UPDATE bills SET balance = arg[0] WHERE clientID = arg[1];
      </state update>
    </method>
  </update methods>
</durability proxy>
```

Figure IV.5: Durability mapping for AccountSet

reduced.

IV.E Related Work

The fault tolerance of services in distributed systems has been an issue since the inception of distributed computing. The basic techniques used to achieve such fault tolerance are fundamentally the same: the state of the service is secured against failure using some form of redundancy. We classify the related work based on the view taken on the service state.

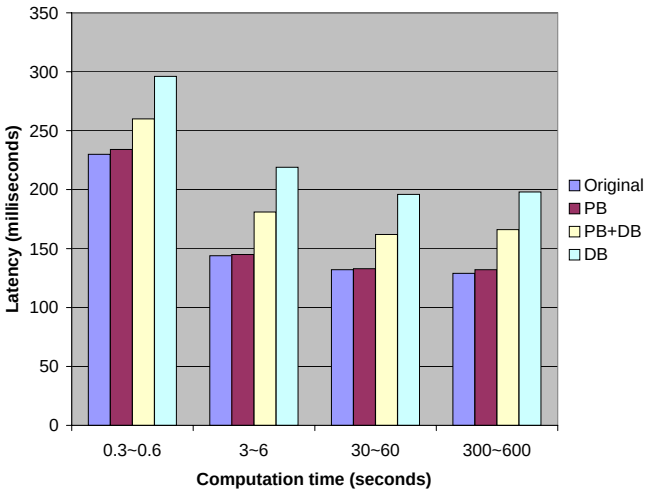


Figure IV.6: Matchmaker Service Performance

One traditional approach is to view the service state as a part of the server, and provide state durability by making the service fault tolerant via server replication. This approach was formalized by the *replicated state machine approach* [53]. Numerous distributed computing platforms based on group communication take this approach for replicating processes to provide fault-tolerant services. This approach was also standardized as the method for providing fault tolerance in CORBA [45] and it was also used in our earlier work on providing fault-tolerance for grid services based on the OGSI model [61]. Such an approach has also been proposed for services in the form of FT-SOAP [41].

Another approach taken to securing service state is to use stable storage provided by a file system or a database. In an SOA, each state change is typically stored to a database immediately before a reply is returned to the client. The Java J2EE [10] architecture for 3-tier e-commerce services provides a durability mechanism transparent to the programmer in the form of *entity beans*. The J2EE runtime environment (application server) provides database based persistence for entity beans transparently. Complete transparency has performance implications, and therefore, systems such as Hibernate have been designed to provide object persistence at a lower cost [7]. Hibernate is a framework that allows the mapping

of a data representation from a Java object model to a relational data model with an SQL-based schema. Hibernate maps Java classes to database tables at runtime. Finally, [59] proposes a replication algorithm that provides both state consistency and exactly-once semantics for stateful J2EE application servers.

The availability of stable storage implemented as a database can be increased using replication. Database replication at the middleware level has recently received considerable attention [43, 3, 5, 46, 52] since it can support a heterogeneous environment without the need to change the underlying database system.

Finally, [34] presents models for evaluating the dependability of data storage system, including both individual data protection techniques and their compositions. These models estimate storage system recovery time, data loss, normal mode system utilization, and operational costs under a variety of failure scenarios.

To our knowledge, no prior work provided the flexibility of allowing different techniques to be used transparently to improve the service state durability.

IV.F Summary

This chapter addresses the increasingly important issue of how to make services in an SOA highly available and fault tolerant. Our work is based on the observation that if the state of the service can survive failures (i.e., is durable), it is relatively easy to construct highly available services. However, durability may have a considerable performance overhead depending on the specific techniques used. Therefore, our approach allows the transparent customization of the durability techniques used for different parts of the service state, that is, different objects that store the service's state. The durability requirements and chosen techniques can be determined after the service and its state objects have been implemented. Our approach is based on using a durability compiler that takes the service implementation, a service-specific durability mapping specification, and reusable durabability proxies and generates code where each state object in the

service is protected by the chosen durability technique. To our knowledge, our proposed system is the first to offer such a level of customization of the durability/performance tradeoff. While we have not implemented the durability compiler, we anticipate that its complexity is comparable to a stub compiler for middleware platforms such as CORBA.

Our performance measurements show that being careful about choosing the appropriate durability techniques can significantly boost performance, with the gain increasing as the number of state objects accessed in a single client invocation increases. We also anticipate that the relative performance gain will improve further as more efficient implementations of services platforms are developed.

IV.G Acknowledgements

This chapter is, in part, reprints of material as it appears in "Customizable Service State Durability for Service Oriented Architecture," by Xianan Zhang, Matti A. Hiltunen, Keith Marzullo, and Richard D. Schlichting, in the Proceedings of the 6th European Dependable Computing Conference (EDCC-6), October, 2006. The dissertation author was the primary coauthor and co-investigator of this paper.

Chapter V

Practical Performance of Paxos

The previous two chapters explored how we can build highly available services in synchronous systems. However, some grid environments are asynchronous — the process execution speeds and message delivery delays are not bounded. We need additional techniques to make services highly available in asynchronous systems.

In this chapter, we evaluate practical performance of existing protocols for asynchronous systems. Based on this foundation, we propose a new set of protocols that support both asynchronous systems and nondeterministic services in the next chapter.

We consider two protocols here: Classic Paxos [35], which implements repeated consensus for crash failures, and Fast Paxos [37], which implements the same protocol under the same failure model, but does so using shorter message chains. Indeed, Fast Paxos is optimal in time (as measured by the length of message chains) for reaching consensus. Fast Paxos does this by offloading dissemination of values from servers to the clients. Of course, this does not mean that Fast Paxos may be faster than Classic Paxos in practice. For example, in the paper that presents Fast Paxos, the author points out that when there are "collisions" (defined later in this chapter), then Fast Paxos can be significantly slower than Classic Paxos.

In this chapter we take a more careful look at the performance of Classic versus Fast Paxos. We then consider two realistic system configurations assuming that servers are collocated: one in which the clients and the servers are all in the same local-area network (as one might have in a private cluster service, such as a resource manager) and one in which the servers are all in one local area network but the clients are attached via a wide area network (as one might have in a replicated Web service). Although the case in which servers are not collocated is also interesting, we concentrate in this work on the cases in which replicas are part of the same local-area network. A discussion of settings in which servers are spread across a wide-area network and how the choices of replicas impact on the availability of a system implementing Classic Paxos appears in [33].

From an analytical point of view, we uncovered several facts that surprised us. For example, increasing replication can, in some circumstances, reduce the chances of collisions; in some networks, the larger replication requirements of Fast Paxos can hurt its performance; in a wide-area setting, Fast Paxos's offloading to the clients the dissemination of values can have a serious impact on performance. In other realistic situations, though, Fast Paxos does indeed have better performance than Classic Paxos.

The chapter proceeds as follows: in section V.A, we give a simple factors model of Classic Paxos and Fast Paxos to describe what are the system parameters that can affect the relative performance of the two approaches. In section V.B, we look at the two protocols using an analytical probabilistic model to understand the relative effects the different system parameters have. In section V.C, we describe a set of experiments we conducted running Classic Paxos and Fast Paxos in two different systems to validate the observations in section V.B. We conclude with some final observations and directions for further research in section V.D.

V.A Classic Paxos versus Fast Paxos: Basics

In this section, we review the message flow of Paxos to gain an understanding of the factors affecting the performance of the two versions. We concentrate on the failure-free case, since in any practical system this should also be the common case.

Both in Classic Paxos and in Fast Paxos, servers have *roles*: there are *proposers* that propose values for consensus, *acceptors* that accept proposals (as well as make promises not to accept certain proposals) and *learners* that learn the outcome of consensus. In our case, clients are proposers and servers are acceptors and learners. An acceptor can have a special role, which is as a *leader*.[1] If there is only one leader and enough acceptors are not faulty, then the protocols are live.[2]

When a new acceptor arises as a leader, the new leader and the acceptors execute a part of the protocol to guarantee that the acceptors accept proposals from the new leader. This is called *the prepare phase*. We assume (again as part of the common case) that there is only one leader whose identity does not change, and so the prepare phase need not be run repeatedly. Thus, the message flow of the two protocols are, from proposing to the servers learning the consensus value:

Classic Paxos proposer $\longrightarrow$ leader $\xrightarrow{p2}$ acceptors $\xrightarrow{p2}$ learners

Fast Paxos proposer $\xrightarrow{p2}$ acceptors $\xrightarrow{p2}$ learners

In this diagram, messages labeled $p2$ constitute *phase 2*. Briefly, the proposer sends a request to the leader (Classic Paxos) or to all of the acceptors (Fast Paxos), and a learner accepts a value when it receives a message from a quorum of acceptors. A quorum in this case is a minimal subset of acceptors necessary to ensure safety. It is important to notice that the message chain for Fast Paxos is one shorter

[1] Although the leader is not necessarily an acceptor, we assume so in this chapter as we assume that every server is an acceptor.

[2] These terms are used slightly differently for the two different papers. Readers who are familiar only with Classic Paxos may wonder why an acceptor is the leader rather than a proposer. The terms we use here are from Fast Paxos.

Table V.1: Classic and Fast Paxos

	Number of replicas	Quorum size	# Length of msg chain
Classic Paxos	$2t+1$	$t+1$	4
Fast Paxos	$3t+1$	$2t+1$	3

than that for Classic Paxos. In using one of Classic or Fast Paxos to implement a replicated state machine, it is often necessary to have an extra message back to the client giving the result of the computation. Such a reply is the same for both variants of the Paxos algorithm, and none of the algorithms imposes constraints on how to implement it (the server that sends such a reply or its form). For this work, we therefore choose to ignore it, and we concentrate on the time to learn.

The two protocols have different replication requirements: Classic Paxos requires $2t+1$ acceptors and Fast Paxos requires $3t+1$ acceptors, where t is the maximum number of faulty acceptors. For Classic Paxos, a quorum consists of $t+1$ acceptors and in Fast Paxos, a quorum consists of $2t+1$ acceptors. Table V.1 presents a summary of the requirements of both algorithms.

Fast Paxos can exhibit a behavior called a *collision*. This occurs when two or more proposers send proposals at approximately the same time and the acceptors receive those proposals in different order. When a collision occurs, a collision resolution protocol is run to determine the outcome of consensus (in fact, this resolution protocol can be run even if $2t+1$ receive the message from one of the clients first, but the learners will learn the consensus value even if this resolution protocol is not run). The equivalent of a collision in Classic Paxos can occur only when there is more than one leader. Although a leader oracle is an important part for a real deployment of Paxos, in this chapter we concentrate on the cases in which the communication with the leader is reliable enough so there are no false suspicions.

Looking at this more carefully, let $R(Src, Dst, Mid, Q)$ denote the elapsed time between when Src sends a message to all servers in the set Mid, and when

the set *Dst* receives the triggered messages from Q replicas in *Mid*. Note that each replica sends *Dst* a message after receiving from *Src*, and some replicas can start sending before others receives from *Src*. Consequently, $R(Src, Dst, Mid, Q)$ is not necessarily the sum of the individual message latencies as there can be overlapping.

Let *prop* be the proposer, *leader* be the leader, A be the set of acceptors, and *learner* be a learner. In addition, let $latency(prop, leader)$ be the message latency from *prop* to *leader*. The latency of Classic Paxos is $latency(prop, leader) + R(leader, learner, A, t+1)$ while the latency of Fast Paxos is $R(prop, learner, A, 2t+1)$ assuming no collisions.

Ignoring collisions, Fast Paxos has a lower latency than Classic Paxos only when $R(leader, learner, A, t+1) + latency(prop, leader) > R(prop, learner, A, 2t+1)$. There are two reasons why this may not be the case:

Wide Area Network Variance When the client is not in the same local area network as the servers, then the variance of messages from the client to acceptors can be quite large. Classic Paxos sends only one such message, while Fast Paxos sends $3t+1$ messages and $2t+1$ of them are in the critical path. The client sends these messages in parallel, and by sending more messages, there is a higher chance of having at least one substantially slower message in the critical path.

Quorum Size A large message variance in the local area network containing the servers can also affect the latency because Fast Paxos requires a larger quorum of acceptors.

We quantify these effects in the following sections.

Collisions also increase the latency of Fast Paxos. How much of an effect this has depends on the probability of collisions and the latency in resolving them. We look at this issue in the following sections.

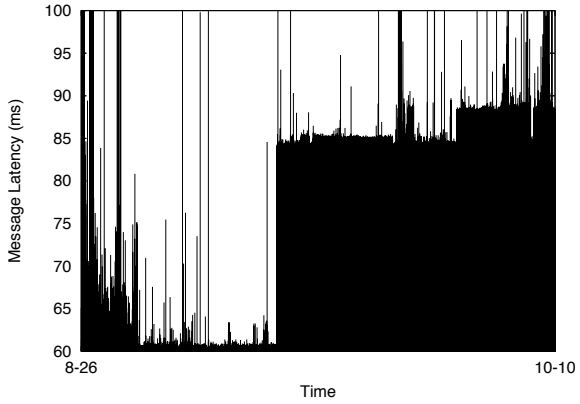
V.B Probabilistic Analysis

The performance of both variants of the Paxos algorithm depends upon how fast the network delivers messages to receivers. Their relative performance depends strongly on the variance of message latency. In many real networks, the distribution of message latencies is often heavy tailed due to traffic variations and non-deterministic scheduling of processes in a single computer. Informally, this implies that most of the time messages are delivered fast, but occasionally messages take one or two order of magnitude more time to be delivered at the receiver.

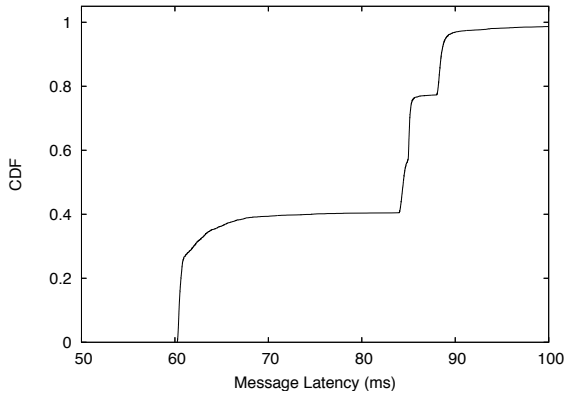
To illustrate this observation, we use traces obtained with NWS (Network Weather Service [58]) in the GrADS testbed [8] over the period between August and October of 2002. These are traces of TCP connections between pairs of machines. Each trace contains the time to establish a TCP connection, send four bytes, receive four bytes, and close the connection. As there are hundreds of such pairs in these traces, we only present graphs for two pairs that we believe are representative of many of the traces in the data set. These pairs are: one pair of hosts connected through a wide-area network (the Internet) and one pair of hosts connected through a local-area network.

Figure V.1(a) shows a time series over the measurement period for a pair of hosts connected through a wide-area link, and Figure V.1(b) shows the cumulative distribution function (CDF) of the same data. Similarly, Figure V.2(a) shows a time series over the measurement period for a pair of hosts connected through a local-area link, and Figure V.2(b) shows the CDF of the same data. In both cases, it is easy to see that the connection times have a large variation, and the corresponding probability distributions have a long tail. The local-area distribution presents less variation, but the tail is still significantly long (indeed, some local-area pairs show much larger variations).

It is not our goal in this work to model distributions of message laten-



(a) WAN Time Series

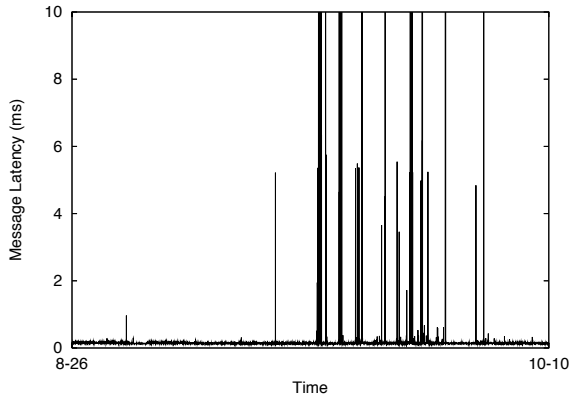


(b) WAN CDF

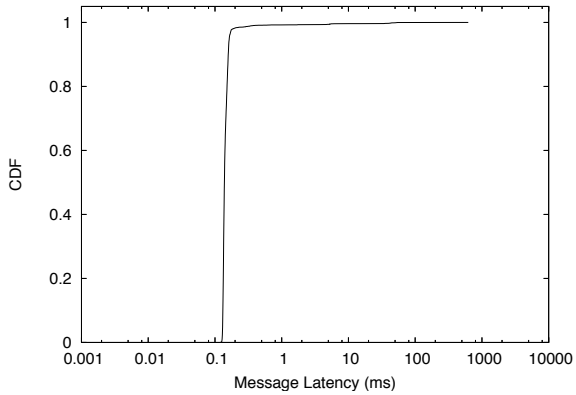
Figure V.1: WAN Trace

cies. In fact, modeling the distribution of message latencies is an open research problem [17]. The only two assumptions we make for the remaining of this chapter based on these data sets is that message distributions have long tail and that the minimum latency in a local-area network is at least one order of magnitude smaller compared to latencies in a wide-area network.

To analyze the performance of Classic and Fast Paxos, we use both synthetic distributions and distributions from our data set. The synthetic distributions use a Pareto model [51]. We decided to use a Pareto model not because it is a par-



(a) LAN Time Series



(b) LAN CDF

Figure V.2: LAN Trace

ticularly accurate model to use, but because they are simple to compute and have a long tail which is adjustable by a parameter of the model. For the distributions from the GrADS traces, we consider four of the large number of possible cases: two cases in which the proposer is connected to the servers through a wide-area link and two cases in which all participants are in the same local-area network. These four are sufficient to illustrate that the computed probabilities have a behavior similar to the synthetic distributions.

In the following probabilistic analysis, we assume that the time of local

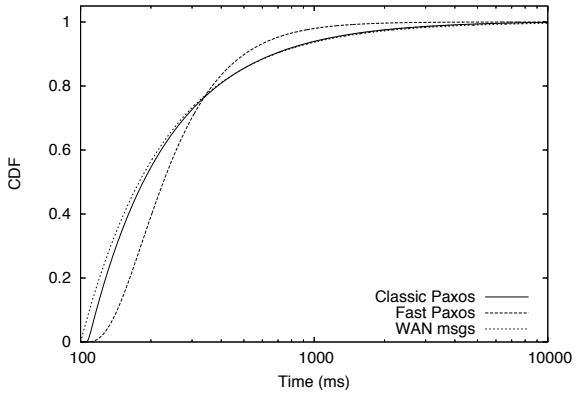
computations is negligible, and that message latencies are independent for distinct messages.

V.B.1 Pareto Distributions

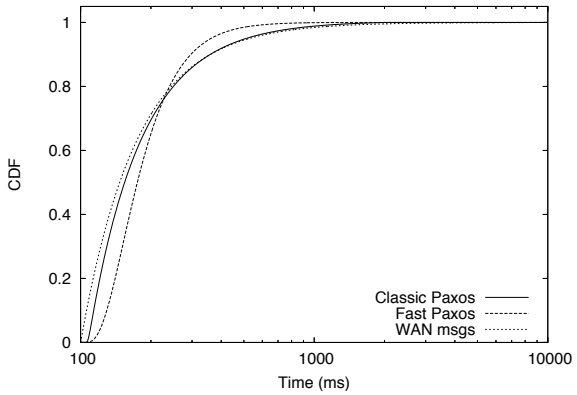
A Pareto distribution has two parameters: $\alpha > 1$ and b. The parameter α determines the shape of the distribution. As we decrease the value of α, we obtain a distribution with a longer tail. The parameter b gives the minimum value in the distribution.

Figures V.3(a), V.3(b) show the probability that a learner learns by time L for different sets of parameters, assuming that there is a single client and that there are no failures. More formally, let L_c be a random variable corresponding to the time that a given learner learns using Classic Paxos, and L_f be a random variable corresponding to the time that a given learner learns using Fast Paxos. We then have that the each of the graphs in Figures V.3(a) and V.3(b) show $P\{L_c \leq x\}$ and $P\{L_f \leq x\}$. Additionally, we plot the distribution of wide-area messages for comparison with the other curves.

For the message distributions, we assume one wide-area distribution (α_w and b_w) and one local area distribution (α_l and b_l). The values we assume for this graph are the following: $\alpha_w \in \{1.2, 1.8\}$, $b_w = 100ms$, $\alpha_l = 2.0$, $b_l = 3ms$, and $t = 1$. We chose α_w and α_l arbitrarily, under the constraint that $\alpha_w < \alpha_l$. This is because we assume that wide-area distributions are more skewed and therefore have a longer tail. We have also chosen the values of b_w and b_l arbitrarily, but following our assumption that b_l is at least one order of magnitude smaller than b_w. A natural choice is the minimum in graph V.2(b). To make the influence of the local-area communication observable, we use a value one order of magnitude larger. Even with such a value, as we can see from the graphs in Figures V.3(a) and V.3(b), the latency of wide-area messages dominates, although the difference between the curve for the distribution of wide-area latency values and the curve for Classic Paxos is noticeable.



(a) $\alpha_w = 1.2$



(b) $\alpha_w = 1.8$

Figure V.3: Time to learn using Pareto distributions: $b_w = 100$, $a_l = 2.0$, $b_l = 3$, $t = 1$

There are a few important observations to make on Figure V.3(a) and Figure V.3(b). First, the Classic Paxos curve follows closely the distribution of wide-area messages. This implies that the local communication has negligible influence on the performance of Classic Paxos. Second, although the latency of wide-area messages also has a higher impact on performance compared to the latency of local-area messages for Fast Paxos, the shape of the curve does not closely follow the one of the distribution of wide-area messages. This is for two

reasons.

1. For a short interval of time, the probability of a learner learning within that interval is smaller for Fast Paxos because the client has to communicate with multiple acceptors. There is consequently a higher probability that one of these messages is slow;

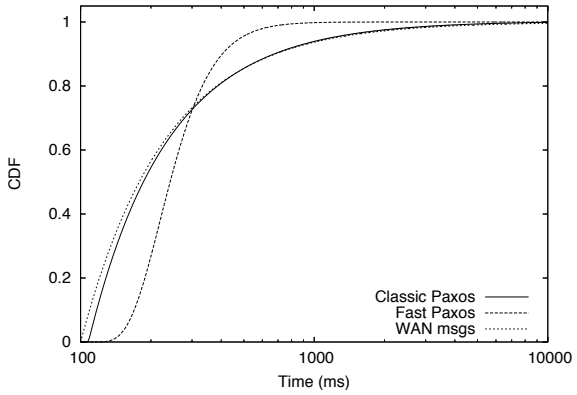
2. For longer intervals of time, the probability that a learner learns within this time interval is higher for Fast Paxos because messages have to be fast only from the client to a quorum of acceptors, and hence a few slow messages do not hurt performance.

This is different of what happens with Classic Paxos: if messages between the client and the leader are fast, then learning is fast, otherwise learning is slow.

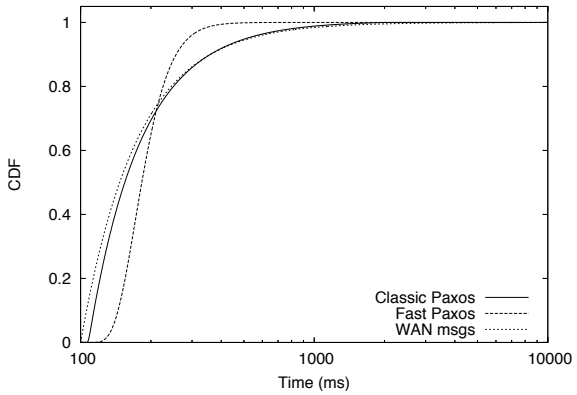
In the graphs, the previous observations map to two regions in the graph: one in which the Fast Paxos curve is under the Classic Paxos curve ($P\{L_c \leq x\} > P\{L_f \leq x\}$), and another in which the Classic Paxos curve is under the Fast Paxos curve ($P\{L_c \leq x\} < P\{L_f \leq x\}$). The division between the two regions is around $300ms$, and the values of $P\{L_c \leq x\}$ and $P\{L_f \leq x\}$ are both over 0.7 for both values of α_w.

One way to interpret this is as follows: asynchronous systems can still have deadlines, but such deadlines can be violated. For deadlines shorter than $300ms$, Classic Paxos is more likely to reach consensus than Fast Paxos. This deadline is longer than the median time to reach consensus, which is closer to $200ms$. If a longer deadline is acceptable, then Fast Paxos is a better choice since it is more likely to reach consensus. For longer deadlines, however, the difference in likelihood to decide between the two Paxos protocols is smaller than the difference for shorter deadlines.

We also present a similar set of CDF graphs for $t = 3$ (Figures V.4(a) and V.4(b)). Comparing to graphs with $t = 1$, we observe that the curve for Fast Paxos is steeper, which implies simultaneously a shorter tail and lower probabilities



(a) $\alpha_w = 1.2$

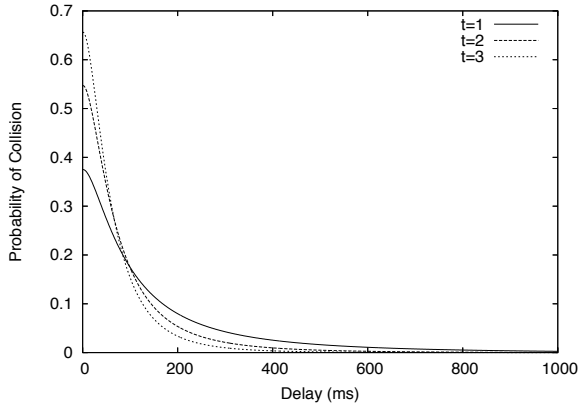


(b) $\alpha_w = 1.8$

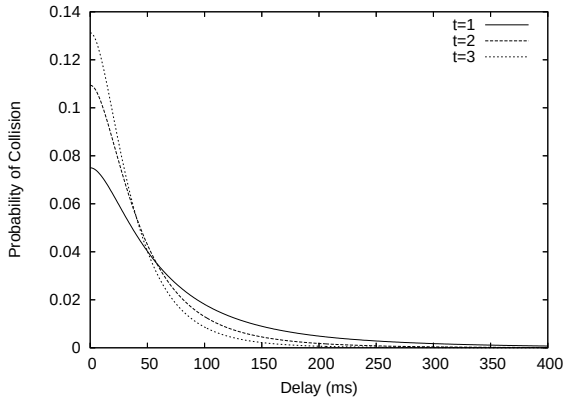
Figure V.4: Time to learn using Pareto distributions: $b_w = 100$, $a_l = 2.0$, $b_l = 3$, $t = 3$

for smaller time values. Note that when we increase the value of t, we are increasing the size of the quorums of acceptors.

As noted before, collisions hurt the performance of Fast Paxos. In Figures V.5(a) and V.5(b), we show the probability of collision for a distribution of message latencies with parameters $\alpha_w = 1.2$ and $b = 100$, and two clients. For this analysis, we assume that the first client sends a request at time r and that the second client sends a request at time $r + \delta$, and the values in the x-axis correspond



(a) $\alpha_w = 1.2$



(b) $\alpha_w = 1.8$

Figure V.5: Probability of a collision - Pareto distributions, $b_l = 100$

to this delay δ. The two requests conflict if none of them can obtain a quorum of acceptors.

As we increase the value of t, the probability is higher for small delays. For example, the probability of collision for simultaneous requests is approximately 0.38 for $t = 1$, whereas this same probability is approximately 0.65 for $t = 3$. For longer delays, however, as we increase the value t, the probability of having a collision decreases faster. This is consistent with our previous observations. As we increase the value of t, the tail of the time-to-learn distribution is shorter.

Consequently the distribution of time values for the first client request to obtain a quorum of acceptors before the second client request can reach enough acceptors to cause a collision must also have a shorter tail.

If one increases the number of clients, then the probability of collisions will increase because there are more ways that no proposer obtains a quorum of acceptors. These curves will still steepen as t increases.

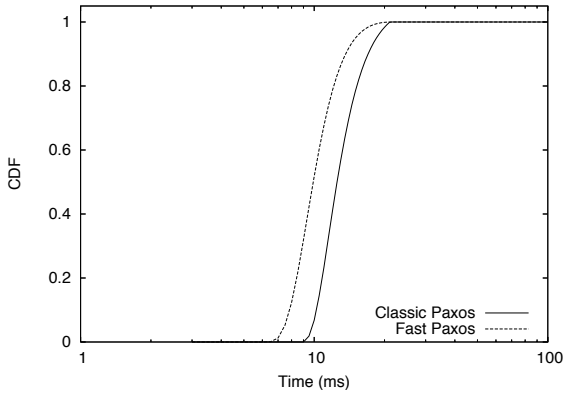
To finish this section, we show a case in which Fast Paxos clearly performs better than Classic Paxos. This happens when clients and replicas are in the same local area network. To illustrate this case, we use $\alpha_w = \alpha_l = 2.0$ and $b_w = b_l = 3ms$. We show distributions of time to learn for this set of parameters in Figures V.6(a) and V.6(b). From these graphs, we have that for any time value x, $P\{L_c \leq x\} \geq P\{L_f \leq x\}$. Increasing the value of t has the effect of making the tail shorter for both Fast and Classic Paxos.

V.B.2 Empirical Distributions

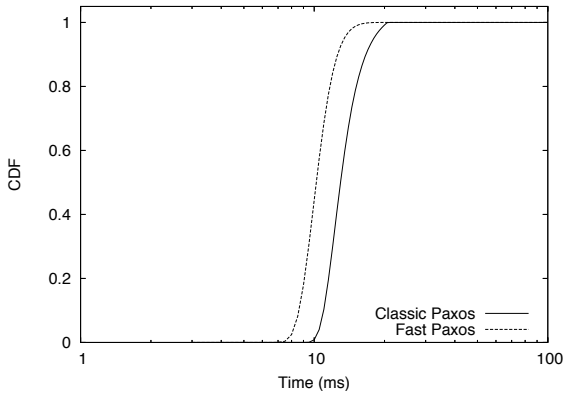
We consider four sets of distributions from the NWS/GrADS traces, both sets containing one wide-area distribution and one local-area distribution. For this analysis, we assume that the time to send a message corresponds to the time of a connection as we described for the traces we have.

The first set, which we call "set 1", uses the distributions in Figures V.1(b) and V.2(b). The graphs in Figures V.7(a) and V.7(b) show distributions of time for a given learner to learn a client request, assuming no competing requests and no failures.

From these graphs, we observe a similar behavior as the one for the Pareto distributions. This behavior corresponds to having two regions, one in which the CDF curve for Fast Paxos is under the curve for Classic Paxos, and another region in which we have the opposite. In the first region, for the same time value x, we have that $P\{L_c \leq x\} - P\{L_f \leq x\} \approx 0.2$. The second region, however, is not as pronounced as the first one, which signifies that, for the same time value x, the



(a) $t = 1$

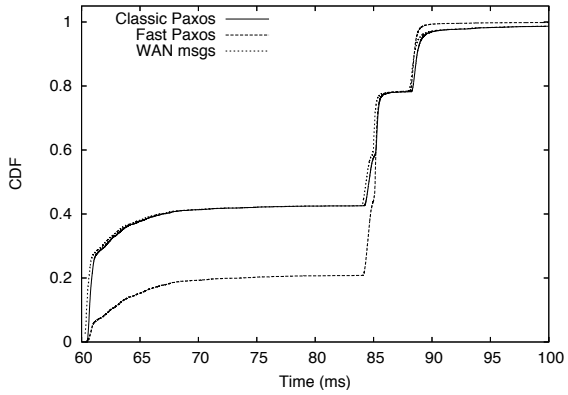


(b) $t = 3$

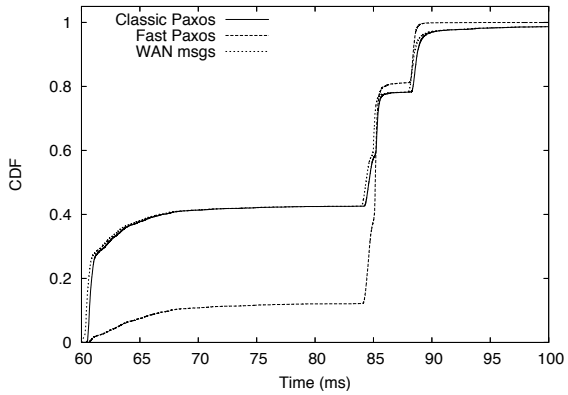
Figure V.6: Time to learn, single Pareto distribution, $a_l = 2.0$, $b_l = 3$

probability of Fast Paxos is not significantly higher than the probability of Classic Paxos. It is also interesting to observe that the two curves overlap for a small range. This implies that the probability is the same for time values in this range. For such a scenario, we conclude that using Fast Paxos as opposed to Classic Paxos is not a good choice.

The second set of distributions, which we call "set 2", uses a less skewed wide-area distribution, but that still has a long tail. We show CDF's, equivalent to the ones for "set 1", in Figures V.8(a) and V.8(b) for different values of t. We can



(a) $t = 1$

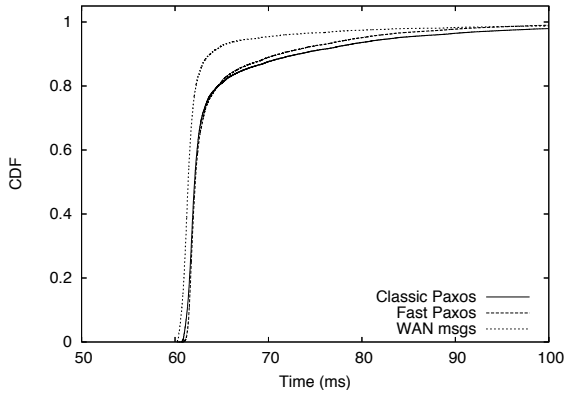


(b) $t = 2$

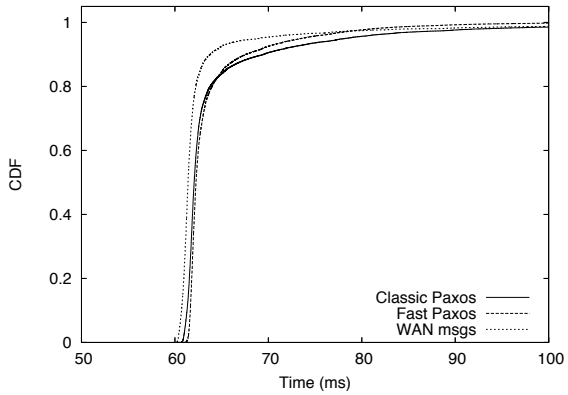
Figure V.7: Time to learn, empirical distributions, set 1

see from these graphs that they also have two regions as in the first case and as in the Pareto cases. In the curves for "set 2", however, the difference between the curves is small. That is, for every possible value of x, $|P\{L_c \leq x\} - P\{L_f \leq x\}|$ is small. In this case, we also conclude that using Fast Paxos is not beneficial, as there is no substantial gain in performance.

As we did for Pareto distributions, we now show cases in which all the communication is local. In Figure V.9(a), we plot $P\{L_c \leq x\}$ and $P\{L_f \leq x\}$ using a moderately skewed distribution. Because the distribution is only moder-



(a) $t = 1$

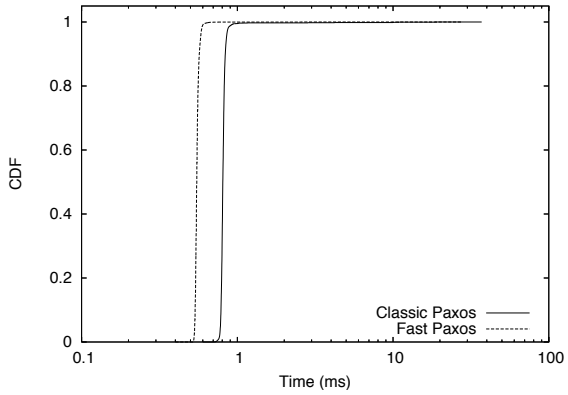


(b) $t = 2$

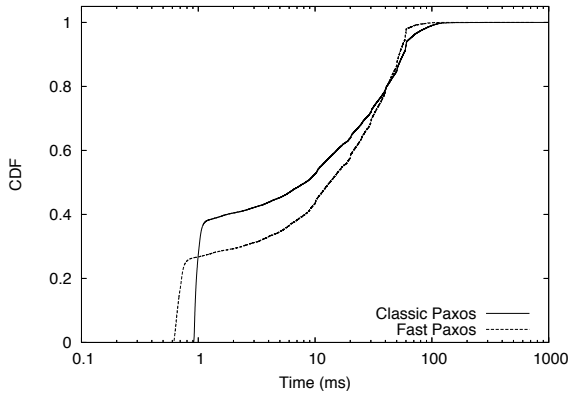
Figure V.8: Time to learn, Empirical distributions, set 2

ately skewed and all the communication is local, Fast Paxos outperforms Classic Paxos. To show another instance of when Fast Paxos may have a poor performance compared to Classic Paxos, we use one trace in which the distribution is unusually skewed to compute the distribution of time to learn. Figure V.9(b) shows the distributions for both Fast and Classic Paxos.

From the figure, for time values under $1ms$, a learner has a higher probability of learning of a request when using Fast Paxos. This probability, however, is at most 0.26. For time values up to $14ms$, the probability is higher for Classic



(a) Moderately skewed



(b) Highly skewed

Figure V.9: Time to learn, Empirical distribution, local-area communication

Paxos, and this probability is roughly 0.8 around $14ms$. For larger time values, Fast Paxos presents a higher probability. The difference, however, is small: the maximum is 0.048 at $60.203ms$.

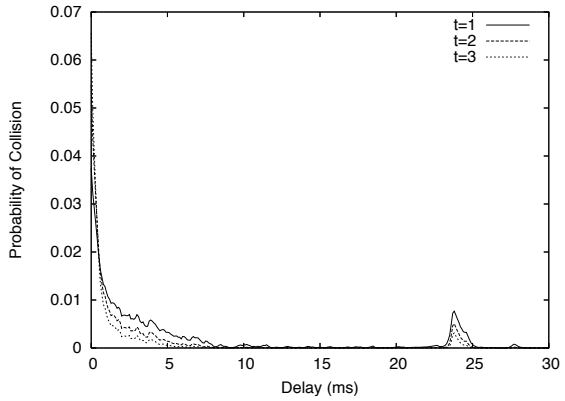
Finally, we computed the probability of collisions assuming that two clients send requests concurrently, exactly as we did for Pareto distributions. The graphs in Figure V.10(a) and V.10(b) show the density probability function for both "set 1" and "set 2" for different values of t. As we observed for Pareto distributions, the probability of a collision decreases faster for larger values of t, and

is higher for small delays and larger values of t. One interesting observation compared to the results for Pareto distributions is that the probability values we see in these graphs are substantially smaller. This is due to the longer tail of Pareto distributions (Pareto distributions are defined in the range $[b, \infty]$) and due to the choices of parameters. Equally interesting is the raise of the probability of collision for "set 2" around $25ms$. This happens due to the nature of the wide-area trace we used. From Figure V.1(a), during the initial period of the trace, the time values are mostly around $60ms$, whereas during the final part, the time values are mostly around $85ms$. For this reason, when the second client sends a request with a delay with respect to the first client around $25ms$, the collision probability increases.

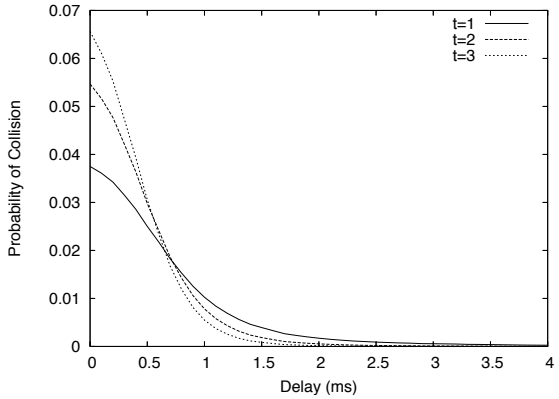
V.C Experiments

In this section, we compare the performance of Classic Paxos and Fast Paxos by measuring the request latency in three different settings. Messages are not written to stable storage here to simplify the implementation. The first setting consists of machines in a local cluster. The other two settings use machines spread across the Internet. We also present figures on the probability of collisions, collected in experiments conducted in the local cluster. The main goal is to show evidence that our results obtained by using probability models hold in practice, and to complement the collision results shown in section V.B.

To tolerate one server failure, we ran three server processes for Classic Paxos, and four processes for Fast Paxos. The server processes were all executing on different machines. Each process served as both an acceptor and a learner, and one of them also served as the leader of Classic Paxos and Fast Paxos. In the following discussion, although every server is simultaneously an acceptor and a learner, we use the terms "acceptor" and "learner" to refer to the particular role of a server in specific cases. We also used another four machines to host the client processes, and each machine hosted no more than two clients.



(a) Set 1



(b) Set 2

Figure V.10: Probability of collision - Empirical distributions

After a client process starts, it waits for a message from the leader before it starts to send requests. The leader sends the signal to the clients simultaneously to ensure the client processes start at (roughly) the same time. After receiving the signal, each client process sends a sequence of requests one by one. Each request is sent to all the acceptors. After receiving a reply from any learner, the client starts a new request. Every learner sends a reply to the client process after learning the chosen value; the client accepts the first reply and ignores the others. When collisions happen, the servers use the coordinated recovery method [37] to resolve

them – the leader chooses a value based on what it receives from the quorum and proposes it to all the servers.

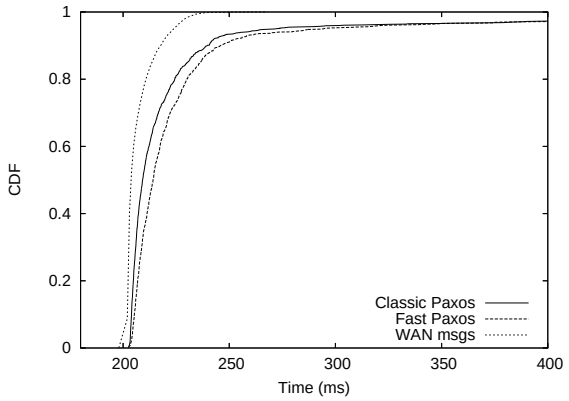
When running Classic Paxos, as long as the leader process receives a request and does not fail before proposing the request to other servers, the request will be chosen and replied if there are enough non-faulty servers. For Fast Paxos, however, it is not as simple. Assuming there are four clients A, B, C and D proposing at the same time – A proposes v_a, B proposes v_b, C proposes v_c and D proposes v_d, the four acceptors can receive these messages in the following order: Replica 1: $< v_a, v_b, v_d, v_c >$; Replica 2: $< v_a, v_b, v_d, v_c >$; Replica 3: $< v_a, v_d, v_c, v_b >$; Replica 4: $< v_d, v_b, v_c, v_a >$.

Assume the leader receives from the quorum consisting of Replica 2, 3, 4 for these four requests. Following the Fast Paxos algorithm, the leader eventually detects collision for the first three requests, and proposes v_a as the first, v_b as the second and v_c as the third request. As a result, v_d will never be learned. Thus, the client process needs to time out and resend the request. However, "time out" is not easy to implement in asynchronous environments since it is tricky to decide a good "time out" value. Even if we could determine a good value, it may degrade the performance because the client has to wait much longer than in the average case. To overcome this problem, we implemented an extra function at the leader process. Besides handling collisions, the leader keeps a counter for each request tracking the number of acceptors that have already accepted this request. When the leader observes that all the servers have already accepted the request but the request has not been learned, it will send the client a message informing the client that the request might not be chosen. The approach does not always work if the message among the servers can be lost, but it can reduce the need of a client time out.

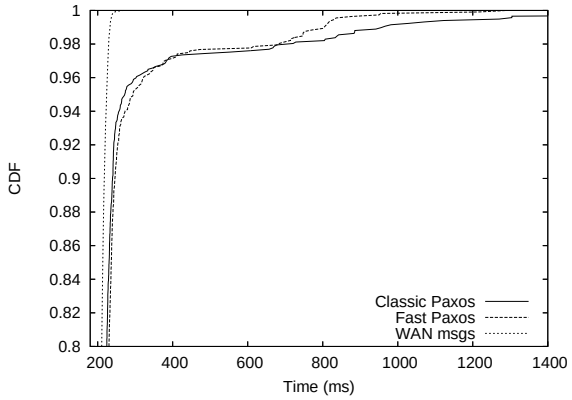
We measured the CDF of single client latency on three settings. The first one is a local cluster called Sysnet, which belongs to the University of California, San Diego (UCSD). Both the server and the client processes ran on Pentium IV

2.8 GHz workstations connected through a Gigabit Ethernet network and running Linux 2.6.11. For the second setting, the server processes ran on PlanetLab machines located at the Max Planck Institute for Software Systems, Germany (MPG), while the client process on a local machine at UCSD. In this case, the client communicates with the servers through a wide-area network, whereas the servers are connected to a local-area network. The third setting is similar to the second one, but the servers are at UCSD and the client is one of the PlanetLab nodes at Princeton University. For approximately 24 hours, we ran Classic Paxos and Fast Paxos approximately every two minutes. In Figure V.11, Figure V.12 and Figure V.13(a), we show the CDF for the latency of requests.

Fast Paxos has better performance compared to Classic Paxos on Sysnet. This is consistent with what we found in Figures V.6(b), V.6(a), and V.9(a). Since the network between the client and the servers is the same as among the servers, saving one message delay does boost the performance of Fast Paxos. For the experiments between UCSD and MPG, Classic Paxos presents smaller latency values. The reason is that the variance of WAN latency between the client and the servers increases the overhead of Fast Paxos and the communication overhead among server processes is relatively small. As we observed in Figures V.8(a), V.8(b), V.7(a), and V.7(b) the curves for Fast and Classic Paxos cross over. Although the cross over point appears at a higher probability value, this is consistent with the predictions of our model. The discrepancies we observe are likely due to process scheduling and execution time, which our model does not account for. We indeed observe that some PlanetLab nodes often present unusually high processing latency, on the order of tens of milliseconds. We attribute this to the high demand on these nodes. At the same time, it is interesting that such factors impact negatively on the request latency for Fast Paxos, which implies that our model is in fact conservative in comparing Fast and Classic Paxos. For the experiments between Princeton and UCSD, Classic Paxos and Fast Paxos behave similarly, since the network variance between Princeton and UCSD is not as big as between UCSD



(a) 0% to 100%

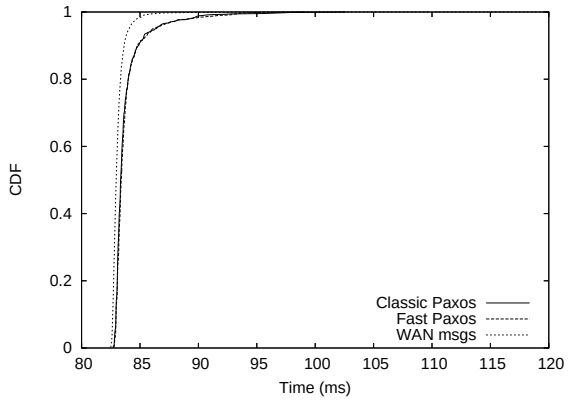


(b) 80% to 100%

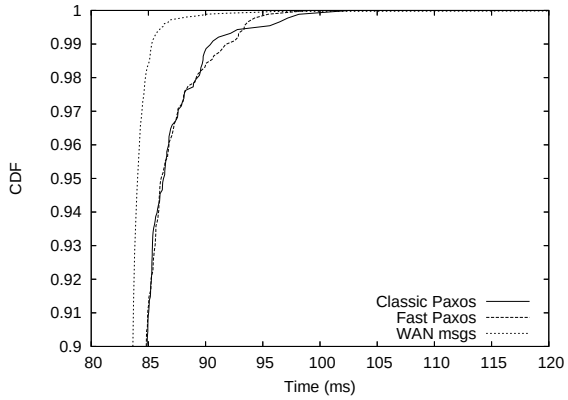
Figure V.11: Request CDF: UCSD – Max Planck Institute

and MPG.

To understand the collision overhead of Fast Paxos, we measured the probability of collisions on the Sysnet cluster for 2, 4, 6 and 8 clients. Figure V.13(b) shows the results. Calibrating a value of δ to try to match experimental results with the curves of section V.B is not possible unless we have perfectly synchronized clocks. We instead start multiple clients simultaneously and have these clients send requests one after another without any waiting interval, where "simultaneously" refers to the time that the clients receive a start message from



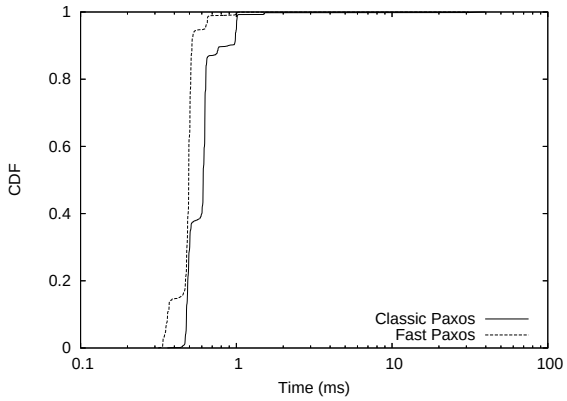
(a) 0% to 100%



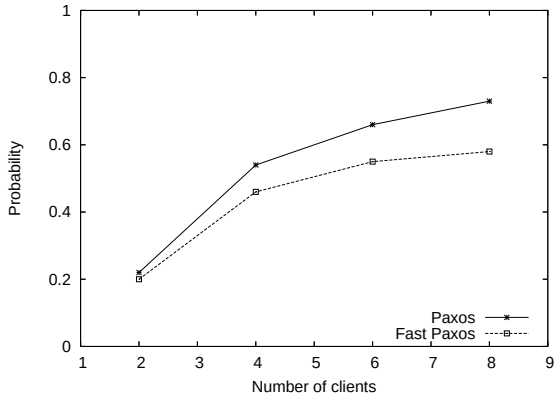
(b) 90% to 100%

Figure V.12: Request CDF: Princeton – UCSD

the leader. Recall from graphs V.10(a) and V.10(b) that the cross over point for different values of t is at the minimum message delay. As the servers we use for this experiment are in the same cluster, this point must have a small value of δ (on the order of microseconds). By increasing the number of clients, we have a higher chance of getting larger values of δ, and consequently, the probability of collision increases with the number of clients.



(a) Request latency CDF



(b) Collision probability

Figure V.13: Sysnet request CDF and collision probability

V.D Summary

Classic Paxos and Fast Paxos are algorithms that implement repeated consensus, and enable the implementation of efficient replicated state machines. Theoretically, the message chain of Fast Paxos has one fewer message, and so by that metric Fast Paxos is indeed faster. In this chapter we looked at the question of when Fast Paxos is faster in practice.

We showed in this chapter that there are some structural details about

Fast Paxos that can impact latency in the common case of no failures. Using a probabilistic model, we found that:

- While it is true that collisions have a negative impact on Fast Paxos, under some circumstances, this impact can be reduced by increasing the quorum size (that is, increasing t), since doing so reduces the probability of collisions.
- If the clients communicate with the servers over a wide-area network, then the variance of communications associated with the WAN can result in Classic Paxos having a lower latency than Fast Paxos.
- Under some circumstances, the larger quorum size of Fast Paxos can lead to Fast Paxos performing worse than Classic Paxos.

As well as our results with synthetic message distributions, we have identified realistic scenarios in which Classic Paxos outperforms Fast Paxos using empirical distributions and in experiments with real networks. When clients (or proposers using the Paxos terminology) are connected to the servers playing the leader, the acceptors, and the learners through a wide-area network, Classic Paxos outperforms Fast Paxos if the communication cost among the servers is negligible compared to the cost of the wide-area communication. Even if all the communication is local, but the distribution of message latency values is highly skewed, Classic Paxos outperforms Fast Paxos. Fast Paxos outperforms Classic Paxos, however, when the message latency is comparable for all pairs of participating processes and the distribution of message latency values does not have a large variance.

The work in this chapter implies that when deploying a replicated service based on Paxos, the choice of Fast Paxos versus Classic Paxos is not necessarily straightforward.

Chapter VI

Primary-backup Paxos

The primary-backup approach is a traditional way to replicate nondeterministic services [11]. In this approach, a primary replica executes client requests and sends the resulting state updates to the backups. Due to the special role of the primary, reliable failure detection becomes essential.

One can implement a perfect failure detector using heartbeats on synchronous systems where the primary periodically sends out "alive" messages to the backups or to some system monitor service [58, 31]. If there is no "alive" messages received for a certain period of time, then the primary is assumed to have failed and a new primary is elected from the backups. However it is impossible to implement perfect failure detectors in asynchronous systems [25, 16], where there is no bound on machines' speed and message delivery time. No receipts of "alive" messages from the primary may only mean that the primary is slow for being overloaded or the messages are delayed because of network congestion. In reality, the grid systems often consist of thousands of heterogeneous machines which belong to different organizations and are connected through unreliable networks. The machines' speed and message delivery time can vary dramatically and many things can go wrong. Therefore, the grid systems are usually not synchronous and we can not use the simple primary-backup approach to replicate services.

This chapter addresses the problem of replicating nondeterministic ser-

vices in asynchronous environments, such as grid systems, by using and extending the Paxos algorithm [35]. We organize the rest of this chapter as following. In section VI.A we introduce the basic protocol and optimizations that deliver better performance for practical applications. In addition, we also discuss the impacts of leader switches on the optimizations. The proof of the protocols is represented in section VI.B. And we evaluate the performance with a local cluster and on PlanetLab in section VI.C. Finally we describe the related work in section VI.D and conclude our work in section VI.E.

VI.A The protocol

We describe a new protocol based on Paxos [35, 36]. In this section, we first introduce the system model (section VI.A.1) followed by the description of our new protocol (section VI.A.2). The performance of the basic protocol can be further improved with two optimizations for practical applications as presented in sections VI.A.3 and VI.A.4. The first optimization improves the performance of the requests that do not change the service's states while the second optimization reduces the overhead of services that support transactions. Furthermore, we discuss the performance impacts of leader switches in section VI.A.5.

VI.A.1 System model

A system is a set of processes that are pairwise connected through channels that are used to send and receive messages. Processes are subject to crash failures: a faulty process executes no steps of the protocol while in a crashed state and it executes correctly the steps of the protocol if it is not faulty. We assume that faulty processes can recover. Once a process recovers, it executes the steps of the protocol correctly. We say that a process is correct if it behaves according to the specification of the protocol.

A process is either a *service process* or a *client process*. A service process

is also called a *replica* or a *service replica* in the following sections of this chapter. We use n to denote the number of service processes. Then, our protocols require that at most $\lfloor (n-1)/2 \rfloor$ service processes are crashed, and thus, at least $\lceil (n+1)/2 \rceil$ service processes are correct.

We assume that the communication channels are reliable in the sense that messages sent between correct processes are eventually received, and that the system is asynchronous. This last assumption implies that there is no bound on the amount of time to deliver a message. Thus, not receiving a message from a process does not necessarily imply that the process has crashed.

A *leader* is a service process that is elected to examine and reply to client requests. A client process sends each request to all service processes, while only the leader sends a reply. A *read* request does not change the service state, while a *write* request changes the service state. As in the Paxos algorithm, we assume that there is an underlying leader election service.

Every service process has data structures that it reads from (read request) and writes to (write request), and a log of commands that it uses throughout an execution to remember executed commands. This log is important to guarantee that once a new leader emerges, this leader learns about all previously accepted requests.

VI.A.2 The basic protocol

To synchronize the replicas of deterministic services, one can implement a series of separate instances of the Paxos consensus algorithm and the proposal chosen by the i^{th} instance is the i^{th} executed request. Therefore, it can be assured that all the service replicas execute the same sequence of requests. However, it is more complicated to synchronize the replicas of nondeterministic services since the replicas may behave differently even if they execute the same sequence of requests.

To guarantee the consistency of nondeterministic service replicas, we let the proposal chosen by the i^{th} instance be a tuple $< req, state >$, where req is

the i^{th} request executed and *state* is the leader's state after executing *req*. Clients send requests to all service replicas so that they do not need to know which replica is the current leader.

We make the usual assumption that the common case is the one of no suspicions and no failures in the system. In executions with no failures and no suspicions, there is a single leader. After receiving the requests, the leader assigns a position in the sequence to each request. If the leader decides that a certain request should be the i^{th} request, it executes the request and then tries to have this request and the new state chosen as the proposal of the i^{th} instance of the Paxos consensus algorithm. Typically, we do not need to run the prepare phase often since the identity of the leader rarely changes. Figure VI.1 shows an example execution of this protocol.

After accepting a proposal, a replica keeps the proposal in its log. Each replica needs to remember all the requests in the accepted proposals, while it only needs to keep the state of the latest proposal. The proposals are ordered by *proposal numbers*, which are pairs of numbers consisting of the ballot number and the instance number associated with the accept request of each proposal. Proposal numbers are ordered lexicographically, first by the ballot number and then by the instance number. When a replica learns that a proposal has been chosen, it applies the state associated with the proposal to its own state.

The leader never tries to propose more than one proposal simultaneously. Although it can start executing the i^{th} request, it will not propose the i^{th} request and the corresponding state until the $(i-1)^{th}$ request commits. Otherwise, if the i^{th} proposal is chosen but the $(i-1)^{th}$ is not, the leader generates a gap in the sequence of chosen proposals. Such a gap makes the service state inconsistent because the state after executing the i^{th} request depends on all the requests executed previously. It does not make sense to commit the state after executing the i^{th} request and abort the $(i-1)^{th}$ request. If the leader fails to receive the expected response to its proposal for the i^{th} instance, it retransmits those messages. If all goes well,

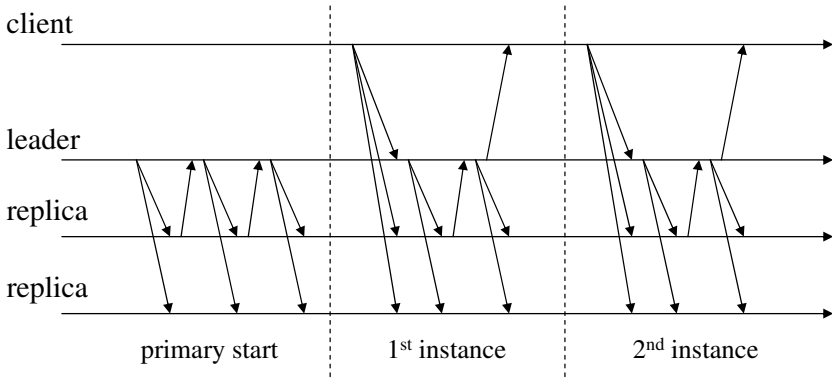


Figure VI.1: The Basic Protocol

then the proposal will eventually be chosen.

If the leader fails, a new leader is elected by an underlying leader election service. It typically knows most of the requests that have already been committed. Assume the leader knows requests 1–87 and 90 and the state after request 90 has been served. Then, the leader executes the prepare phase of instances 88, 89, and of all instances greater than 90. It can do this by sending a single message to all the other replicas that does not include the states after executing 88 or 89 since the replicas are only interested in the latest state. A replica responds with a simple OK unless it knows any of these instances from some other leader. If the replica knows 88 or 89, it only needs to send back the request and the associated proposal number. If the replica knows any instance greater than 90, it sends the leader not only all the requests greater than 90 with the proposal numbers but also the state of the latest proposal it knows.

Suppose the outcome of this execution determines the requests in positions 88, 89, 91 and the state upon execution of request 91. The leader executes the accept phases of instances 88, 89, and 91 by sending one single message to all the replicas, and makes requests 88, 89, and 91, as well as the latest state, chosen and learned.

In practice, service state can be large, and there may be a significant overhead in transferring the whole service state among replicas. It is possible

to reduce this overhead with two approaches. If the nondeterministic operation can be reproduced with the client request and some additional information, the replicas only need to exchange the request and the additional information, and each replica can generate the state itself. For example, in the grid scheduling service of section I, the primary only need to send the state of its queue when it selects a new request, assuming that requests have deterministic priorities. In doing so, replicas know exactly what request comes next in the schedule. Even if the service state is difficult to reproduce, the replicas may be able to exchange only the updated state if they already agree on the previous state. Therefore, the overhead of transferring service state can usually be made small.

VI.A.3 X-Paxos for read requests

It is relatively expensive to run the basic protocol for every request due to the overhead of the accept phase. Although we do not believe one can relax the algorithm for writes, it is possible to improve the performance of reads. This section presents an optimization that benefits read requests. We call this optimized version X-Paxos.

The optimization takes advantage of the synchronization requirements of read requests. Unlike writes, the relative order of reads is not critical. For example, if the clients send two read requests, say r_1 and r_2, to the service at the same time, it makes no difference to the service state whether r_1 is executed before r_2 or after. Therefore, there is no need to maintain a total order of all the requests. On the other hand, there is a consistency requirement for reads: the value that the service returns as a response to a read must reflect the latest update. In other words, the relative order between reads and writes is important. If w is the last write request before the read request r, the returned value of r should be consistent with the service state after executing w. Therefore, a service process p should not respond to any read request if there is another service process q that has already executed and replied to a write request w, while p does not know w. This requirement can

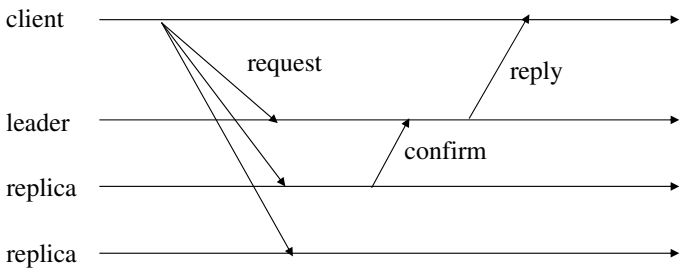


Figure VI.2: X-Paxos

be satisfied if only the leader process that proposed the highest accepted ballot number responds to read requests from client processes.

Based on these observations, the X-Paxos protocol optimizes the performance of read requests while still satisfying the consistency requirements of replicated grid services. X-Paxos is a simple majority-voting protocol and is not a real consensus protocol. It assures that only the latest leader can reply to the requests and that the replicas do not need to agree on the order of the read requests. Figure VI.2 shows how X-Paxos works in a scenario with one client and three server replicas.

If the request sent by the client is a read, the service replicas coordinate using X-Paxos. Otherwise, the replicas coordinate using the protocol described in section VI.A.2. When serving a read request, the leader starts executing it and waits for the confirm messages from the majority at the same time. Every other service process sends a confirm message to the process with the highest ballot number it has accepted after receiving the read request. The leader sends its reply to the client only after it gets the confirmations from a majority of the replicas (including itself). Since a service process becomes the leader only after a majority of service processes accept its ballot number, only the leader with the highest accepted ballot number can receive confirms from a majority and respond to read requests.

Assume E is the execution time of the request, M is the message latency

between a client and a service replica, and m is the message latency between two service replicas. Then, the request latency of X-Paxos is $2M + \max(E, m)$, assuming that all values are constant and that processes are able to send messages in parallel. The first M is the message latency for the client sending the request to the service replicas. $\max(E, m)$ is the time the leader spends on executing the request and waiting for confirming messages concurrently. The second M is the time for the reply message sent back to the client. The request latency of the basic protocol is $2M+E+2m$ assuming we can ignore the overhead of checkpointing. $2M$ is sum of the message latencies for the request sent from the client to the service replicas and the reply sent from the leader to the client. $2m$ is the overhead of running the accept phase of Paxos among the service replicas.

When read requests are predominant, X-Paxos achieves better performance compared to Paxos, since X-Paxos saves one message delay and performs the request execution and the wait for confirming messages in parallel. However, if the message latency variance of the network between the clients and the service replicas is much larger compared to the message latency of the networks interconnecting service replicas, X-Paxos may not achieve better performance because it may take longer for the request to reach a majority than reach only the leader. This special case is out of the scope of this chapter.

VI.A.4 T-Paxos for transactions

The second optimization reduces the synchronization overhead of requests that use transactions. We call the protocol with this optimization T-Paxos. T-Paxos does not implement atomic transactions. Instead, if the client uses transactions when invoking the service, T-Paxos improves the service performance.

If clients use transactions when submitting requests to the service, the overhead of replication can be reduced further. When a client uses transactions, it can rollback and discard the data from the service if the transaction is aborted. Assume a transaction consists of three requests: r_1, r_2, and r_3. The client commits the

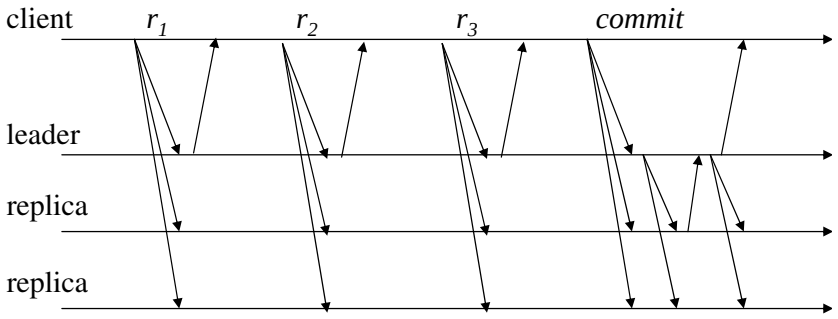


Figure VI.3: T-Paxos execution

transaction after successfully executing these three requests. Then, the sequence of messages the client sends to the service is: $r_1, r_2, r_3, commit$. The leader does not need to coordinate with other service replicas until it sees the *commit* message, and it can reply to each client request immediately. With this optimization, the response time of individual requests is the same as for an unreplicated service, but the overhead is paid at the commit phase. Figure VI.3 shows how service processes coordinate using T-Paxos when serving the transaction $r_1, r_2, r_3, commit$.

If the service handles more than one transaction at a time, the service may have an inconsistent state when some transactions commit and others abort. For example, transaction T_1 consists of $r_1, r_3, commit$, and transaction T_2 consists of $r_2, r_4, abort$. If the leader receives and executes the requests in the order $r_1, r_2, r_3, commit(T_1), r_4, abort(T_2)$, then the service cannot abort r_2 without introducing inconsistency with the results of T_1. Note that this problem is not introduced by replicating the service—any service that supports transactions needs to deal with concurrency of this type using locks or other mechanisms.

VI.A.5 Leader switches

Theoretically, there must eventually be one server that becomes the leader for X-Paxos to be live, just as for Paxos. In practice, both Paxos and X-Paxos require that one server remains the leader long enough for the protocol to termi-

nate. "Long enough" is longer for X-Paxos than for Paxos, because in X-Paxos, the leader needs to receive messages confirming that it is the leader from a majority of processes, while in Paxos, a process accepts any proposal with a ballot number no smaller than the ones it has already accepted. "Long enough" is even longer for T-Paxos; if the leader switches during the transaction, the previous leader that executes the requests of the transaction cannot commit, and the transaction has to be aborted. Thus, X-Paxos and T-Paxos are more sensitive to leader switching than Paxos.

Leader stability is a characteristic of asynchronous leader election protocols, like those that underlie practical implementations of Paxos. Leader stability characterizes under what conditions the leader changes. Recent work on leader election (*e.g.*, [42]) has concentrated on having good leader stability. Hence, these recent leader election protocols satisfy the requirements of our protocols as well.

VI.B Proof

We prove the proposed protocol implements the property of atomic registers. The definitions of different registers are as below.

Safe register A safe register is one in which a read not overlapping any write returns the most recently written value. A read that overlaps a write may return any value from the domain of the register.

Regular register A regular register is a safe register in which a read that overlaps a write obtains either the old value or the new value.

Atomic register An atomic register is a safe register in which reads and writes behave as if they occur in some total order, which is an extension of the precedence relation.

Since T-Paxos only treats transactions as individual requests (committing all the requests in one transaction together), and the commits are coordinated using

either the basic protocol or X-Paxos, we do not prove transactions seperately with requests in our proof. We only need to prove that the combination of the basic protocol and X-Paxos implements atomic registers. The formal description of the combined protocol is:

Protocol 1. *A new leader is established by executing the prepare phase and the accept phase as described in Section VI.A.2. Then the leader handles read requests using X-Paxos described in Section VI.A.3, and handles write requests according to the basic protocol.*

New leader – After a new leader is elected, it proposes its current state and the set of commands it knows to all the replicas by executing both the prepare phase and the accept phase. The new leader will not executes and reponds to any new request until both two phases complete successfully.

Read – After the leader receives a read request, it executes the request and waits for the confirm messages from the replicas. The leader responds to the client process if and only if the request has been executed completely and the leader receives the confirm messages from a majority of the replicas.

Write – After the leader receives a write request, it executes the requests. After the execution completes, the leader sends all the replicas a proposal with the leader's current state and the set of commands the leader knows. The leader responds to the client process if and only if the proposal has been accepted by the majority of the replicas.

We prove Protocol 1 implements Automic Register semantics as follows.

Definition 1. *We say a request r happens at $[t_1, t_2]$, if the first time that r is submitted is at t_1, and r is received by the client at t_2.*

Definition 2. *We use $< p, b >$ to represent a primary replica. p is the process ID and b is the ballot number p proposes and uses. Let $< p_1, b_1 >$ and $< p_2, b_2 >$ be two primaries. We define $< p_1, b_1 > \prec < p_2, b_2 > \equiv b_1 < b_2$.*

Definition 3. *A state sequence is a list of states and write requests served, with the format like:* $s_0, w_1, s_1, w_2, s_2, ...w_i, s_i, ...w_n, s_n$. s_0 *is the initial state,* w_i *is the* i^{th} *write request served,* s_{i-1} *is the state before executing* w_i*, and* s_i *is the state after executing* w_i*. We use* $S(< p, b >, t)$ *to represent primary replica* $< p, b >$*'s state at time* t*.*

Lemma 1. *There is a linear order of all the primaries, according to* $\prec$ *relationship of the primaries.*

Proof. This is true, because according to our protocol, a replica can only become a new primary by choosing a balnum bigger than all the previous primaries' balnums. □

We define the functions *succ* and *prec* of primaries as below:

$succ(< p, b >)$ the next primary after $< p, b >$ in the linear order.

$prec(< p, b >)$ the last primary before $< p, b >$ in the linear order.

Theorem 1. *The protocol implements the property of safe registers. That is: for any read request* r*, if* r *does not overlap with any write request, and write request* w *is the last served write request by a service replica, then the returned value of* r *is equal to the written value of* w*.*

Proof. Assume $< p, b >$ is the service replica which replied to r, r happens at $[t_1, t_2]$, and w happens at $[t_3, t_4]$. We know $t_2 < t_3$.

If $< p, b >$ is also the replica which served w, then p's state will clearly be equal to the written value of w when responding to r. We know that the returned value of r is the returned value of w.

If $< p, b >$ is not the replica which served w, then we assume $< p_0, b_0 >$ is the service replica serving w, and s is $< p_0, b_0 >$'s state sequence after serving w.

Now we want to prove: For any primary $< p_i, b_i >$, if w and s are in $< p_i, b_i >$'s state sequence at t and $< p_{i+1}, b_{i+1} >= succ(< p_i, b_i >)$, then w and

s are in $< p_{i+1}, b_{i+1} >$'s state sequence at t', for any $t' > t$ and $< p_{i+1}, b_{i+1} >$'s primaryID $= < p_{i+1}, b_{i+1} >$ at t'.

1. w and s are in $< p_i, b_i >$'s state sequence at t, only if more than half of the replicas send it phase 2b messages containing w and s before t.

2. $< p_{i+1}, b_{i+1} >$ can only become a primary after receiving phase 1b and phase 2b messages from more than half of the replicas.

3. From above, we know there exists a replica R, such that R accepts the phase 2a message containing w and s from $< p_i, b_i >$, and $< p_{i+1}, b_{i+1} >$ receives the phase 1b messages from R.

4. From the protocol, we know R can't accept the phase 2a message from $< p_i, b_i >$ after sending a phase 1b message to $< p_{i+1}, b_{i+1} >$, since $b_{i+1} > b_i$. Then R sends the phase 1b message to $< p_{i+1}, b_{i+1} >$ after accepting the phase 2a message from $< p_i, b_i >$

5. So we know $< p_{i+1}, b_{i+1} >$ learns w from R. Then $< p_{i+1}, b_{i+1} >$ knows w and s at t', for $\forall t' > t$, $< p_{i+1}, b_{i+1} >$'s primaryID $=< p_{i+1}, b_{i+1} >$ at t'.

Using induction, we have w are s are at the end of $< p, b >$'s state sequence when $< p, b >$ serves r. And the theorem is proved. □

Definition 4. *Each request r has a sequence number $s(r)$, and $s(r) =< b, i >$ if and only if $\exists$ a primary $< p, b >$: r's successful reply is from $< p, b > \wedge$ r is the i_{th} request $< p, b >$ replied. We let $bal(r) = b$, and $ind(r) = i$.*

Definition 5. $r_1 \prec r_2 \equiv [bal(r_1) < bal(r_2)] \vee [(bal(r_1) = bal(r_2)) \wedge (ind(r_1) < ind(r_2))]$.

Lemma 2. *All the successfully replied requests are linearly ordered according to $\prec$ relationship of the requests.*

Proof. From Lemma 1, we know all the primaries are linearly ordered. So we know all the successfully replied requests are linearly ordered according to $\prec$ relationship. □

We define the functions *succ* and *prec* of requests as below:

$succ(r)$ the next request after r in the linear order.

$prec(r)$ the last request before r in the linear order.

Lemma 3. $\forall$ *write requests* w_1 *and request* r*, if* w_1 *is the last write request before* r *according to* $\prec$ *relationship, the state before serving* r *is the same as the state after serving* w_1*.*

Proof. If w_1 and r are served by the same primary, then this is trival.

If w_1 and r are served by different primary $< p, b >$ and $< p', b' >$, we know $b < b'$. $< p, b >$ must receives phase 2b messages from more than half of the replicas before serving w_1. And $< p', b' >$ must receives phase 1b messages from more than half of the replicas before becoming a primary and serving w_2. So there must be some replica which replies both the phase 2b message to $< p, b >$ and the phase 1b message to $< p', b' >$. Then we know it must replies the phase 2b message to $< p, b >$ before replying to the phase 1b message to $< p', b' >$. Otherwise, the protocol is violated. So we know $< p', b' >$ would know w_1 and the state after serving w_1 from this phase 1b message. Since w_1 is the last write before r, $< p', b' >$'s state before serving r should be the same as $< p, b >$'s state after serving w_1. □

Theorem 2. *The protocol implements the property of atomic registers.*

Proof. Theorem 1 has proved the protocol implements safe registers. We only needs to prove all requests behave as if they occur in some total order.

From Lemma 2, we know all requests are linearly ordered according to $\prec$ relationship. From Lemma 3, we know all requests behave as if they occur in this order. □

VI.C Evaluation

In this section, we evaluate the performance of our protocol by comparing request response time and service throughput of the basic protocol and X-Paxos, as well as transaction response time and service throughput (transactions per second) of T-Paxos. In the experiments, the service replicas first receive the client requests, then coordinate with each other using the protocols described in the previous section, and finally reply to the clients. The service executes no actual operation for serving the requests, so the CPU cycles consumed by the replicas for this step are minimal; thus our experiments highlight the replication overhead. The communication between service replicas, and between clients and service replicas, uses TCP sockets. Although we could have integrated our prototypes into a grid middleware such as the Globus Toolkit [27], we chose not to do so, since the overhead of XML-based Web service requests in Globus is considerable [54] and it would have dominated the replication overhead. Thus, by minimizing the communication overhead between clients and the replicas by using TCP, we once again highlight the replication overhead.

We use three different configurations in our experiments. The first one is a local cluster called Sysnet that belongs to the Computer Science Systems Group at the University of California, San Diego (UCSD). Both the service and the client processes ran on Pentium IV 2.8 GHz machines connected through a Gigabit Ethernet network and running Linux 2.6.11. Since the Sysnet network is fast and reliable, the main reasons for the system to behave asynchronously in this configuration are process scheduling and machine load (computers under a heavy load can slow down processes). Both the second and third configurations use PlanetLab [18] machines and represent the deployments of replicated services on wide-area networks. Our second configuration represents the case where the clients are remote from the service replicas but the service replicas are located relatively close to one another. Specifically, all service processes were located on different machines at

Princeton University, while the client processes were located on machines at University of California, Berkeley. Our third configuration represents the case where a service is replicated across a wide-area network to tolerate correlated failures [33] (*e.g.*, power outage, network outage). Specifically, the leader replica ran on a machine at the University of Illinois at Urbana-Champaign, and the other service replicas ran at the University of Utah and the University of Texas, Austin. The client processes ran on machines locating at the University of California, Berkeley and Intel Labs Oregon.

We ran three service replicas ($t = 1$) in all of our experiments, since we believe this is the most common case in practical systems. The server replicas executed on different machines in order to tolerate one server failure. The service takes three kinds of requests: **read**, **write**, and **original**. All three kinds of requests invoke an empty method and do not trigger any actual operation. However, a **read** represents a request that does not change the service state, while a **write** represents one that changes the service state. Furthermore, both **read** and **write** requests require that the service replicas coordinate with each other. Thus, the service replicas use the basic protocol to coordinate for **write** requests and use X-Paxos for **read** requests. An **original** request represents the request sent to the original non-replicated service. The leader replies immediately after receiving an **original** request without coordinating with the backup replicas. The size of service state is small (a few bytes) in our experiments. Although a larger state might result in a higher overhead, the size of the state can often be kept small, as described in section VI.A.2. The overhead of transfering larger size of state was analysized in [61].

We used another eight machines to host the client processes. Each test client sends a specified number of one kind of request sequentially to the service replicas. After a client process starts, it waits for a message (start signal) from the leader before it starts to send requests. The leader sends this signal to all the clients simultaneously to ensure that the client processes start at (roughly)

the same time. After receiving the signal, each client process sends a sequence of requests one by one. Each request is sent to all service replicas, and only the leader replica sends a reply to the client process. A client does not send a new request until it receives the reply associated with the previous one. For the response time tests, we used one client that sends 20 requests in one sample test, while for the throughput tests, we used c concurrent clients, $c \in 1, 2, 4, 8, 16$, where each client sends exactly $1000/c$ requests. For each value, we collected hundreds of samples to measure the confidence intervals.

VI.C.1 The basic protocol and X-Paxos

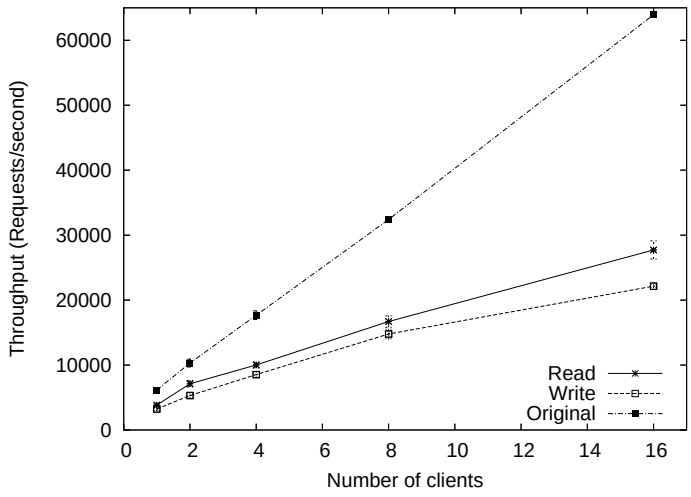


Figure VI.4: Service throughput on Sysnet

On Sysnet, the measured average request response time (RRT) of **original** requests was 0.181 ms (with 99% confidence interval ±0.002 ms). The average RRT of **read** requests was 0.263 ms (with 99% confidence interval ±0.02 ms), while the average RRT of **write** requests obtained in the experiments was 0.338 ms (with 99% confidence interval ±0.003 ms). This shows that X-Paxos reduced the RRT 22% compared with the basic protocol. Figure VI.4 shows the service throughput when serving different requests. Although the throughput of both "reads" and "writes" are lower compared to **original** requests (as expected), the

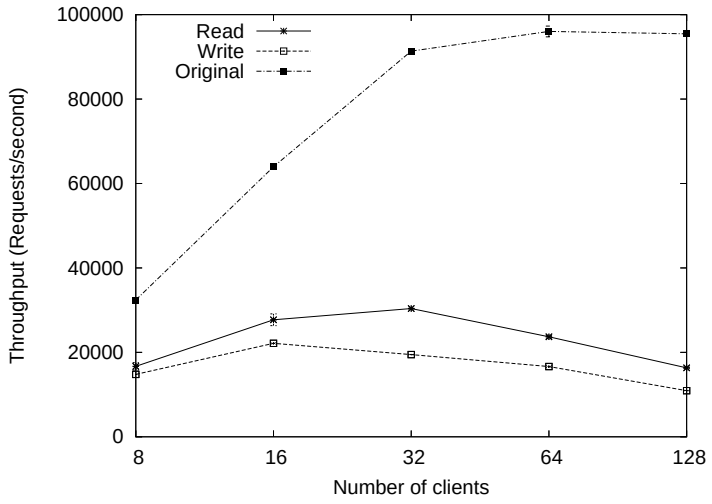


Figure VI.5: Service throughput – more clients

throughput of "reads" was at least 13% higher than that of "writes". The 99% confidence intervals of all the throughput values were no wider than 16% of the average value, and some of the confidence intervals are not readable in the figure. Our experiments indicate that on local-area settings the performance of the basic protocol described in section VI.A is acceptable, and X-Paxos significantly reduces the request response time (22%) and improves service throughput (at least 13%) for **read** requests.

Figure VI.5 shows the throughput with the number of clients ranging from 8 to 128 in log scale. All clients are processes that execute across eight machines and the number of clients on each machine range from 1 to 16. The basic protocol achieves the highest throughput when the number of clients is between 8 and 32, while X-Paxos achieves the highest throughput when the number of clients is between 16 and 64.

For the second configuration (Berkeley to Princeton), the RRT of **original** requests was 91.85 ms (with 99% confidence interval ±0.18 ms). The RRT of **read** and **write** requests are close to that of **original** requests—92.79 ms (with 99% confidence interval ±0.48 ms) for **reads** and 93.13 ms (with 99% confidence interval ±0.27 ms) for **writes**. The throughput of **original**, **read**, and **write**

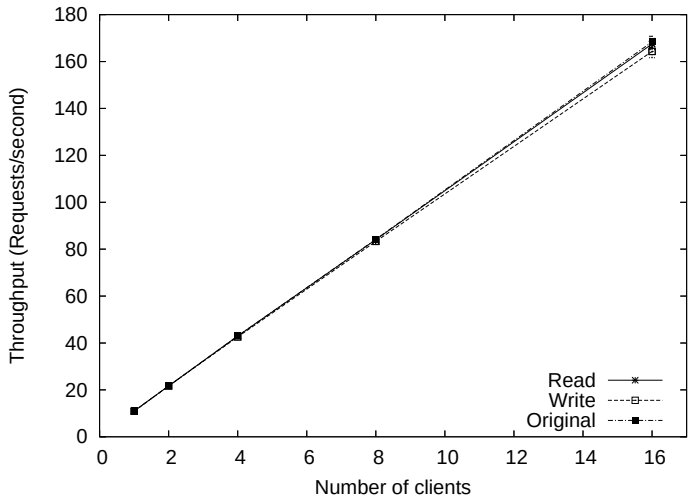


Figure VI.6: Service throughput from Berkeley to Princeton

requests are also close, as shown in figure VI.6. This indicates that if service processes are located at the same site and client processes are on different sites, the basic protocol achieves performance roughly the same as a non-replicated service and the X-Paxos optimization does not improve RRT and throughput much. The reason is that communication among service processes is relatively cheap compared to communication between client processes and service processes.

For the third configuration, the RRT of **original** requests was 70.82 ms (with 99% confidence interval ±0.25 ms), **read** requests was 75.49 ms (with 99% confidence interval ±0.25 ms), and **write** requests was 106.73 ms (with 99% confidence interval ±0.32 ms). Figure VI.7 presents the throughput of **original**, **read** and **write** requests. We can see that when service processes are located on different sites, X-Paxos achieves better performance than the basic protocol.

VI.C.2 T-Paxos

To evaluate the performance improvement of T-Paxos, we measured the transaction response time (TRT) and throughput (number of transactions per second) of the following operations on the Sysnet cluster with the number of requests per transaction equal to 3 and 5:

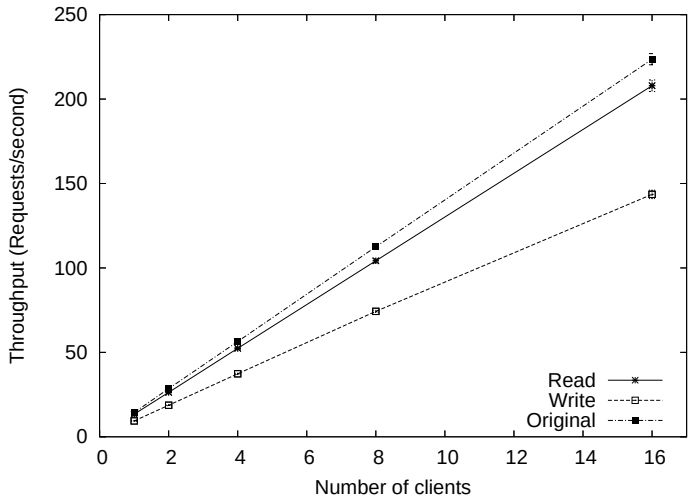


Figure VI.7: Service throughput on WAN

- **Read/write**. Transactions consist of both read and write requests. T-Paxos was not used in this case: X-Paxos was used to coordinate for reads and the basic protocol was used to coordinate for writes and commits. We assume a 3-request read/write transaction consists of 2 reads and 1 write, while a 5-request read/write transaction consists of 3 reads and 2 writes.

- **Write-only**. Transactions consist of only write requests. The basic protocol was used in this case to coordinate for all requests and commits.

- **Optimized**. Transactions consist of the same number of requests. The optimized protocol (T-Paxos) was used, and the replicas only coordinate with each other during commits.

Service processes coordinate for commits even for unoptimized read/write and write-only transactions since processes perform functions that affect service state when committing transactions, such as deleting checkpoints and logs.

Table VI.1 summarizes the transaction response time with 99% confidence intervals of different operations. T-Paxos reduced the transaction response time by 28% for 3-request read/write transactions and 34% for 3-request write-only transactions. When the size of a transaction gets larger, the performance improve-

Table VI.1: Transaction response time

Operation	Req/tran	Avg. TRT (ms)	99% CI (ms)
Read/write	3	1.17	±0.02
	5	1.79	±0.02
Write-only	3	1.29	±0.02
	5	2.01	±0.03
Optimized	3	0.85	±0.02
	5	1.23	±0.02

ment becomes even more significant: the reduction in the transaction latency was 31% for 5-request read/write transactions and 39% for 5-request write-only transactions. The 99% confidence intervals are no wider than ±0.03 ms.

Figure VI.8 shows the throughput of read/write, write-only, and optimized transactions and the 99% confidence intervals (some intervals are again too small to read). T-Paxos increases the service throughput by 42%, 43%, 45%, 47% and 57% when the number of clients was equal to 1, 2, 4, 8 and 16, respectively, compared to 3-request read/write transactions. Compared with 3-request write-only transactions, the transaction throughput increases by 52%, 53%, 77%, 88% and 97% with the number of clients equal to 1, 2, 4, 8 and 16, respectively. Throughput improves even more significantly for 5-request transactions: a $53\% - 90\%$ increase for read/write transactions and a $69\% - 138\%$ increase for write-only transactions.

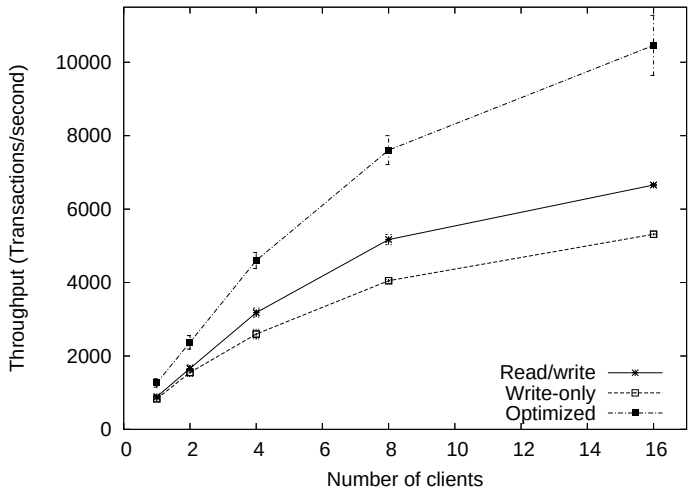
Our experiments with transactions show that T-Paxos reduces the transaction response time and increases the service throughput for applications that support transactions. The impact on both throughput and response time becomes more significant as the number of operations in each transaction increases.

VI.C.3 Tolerating multiple failures

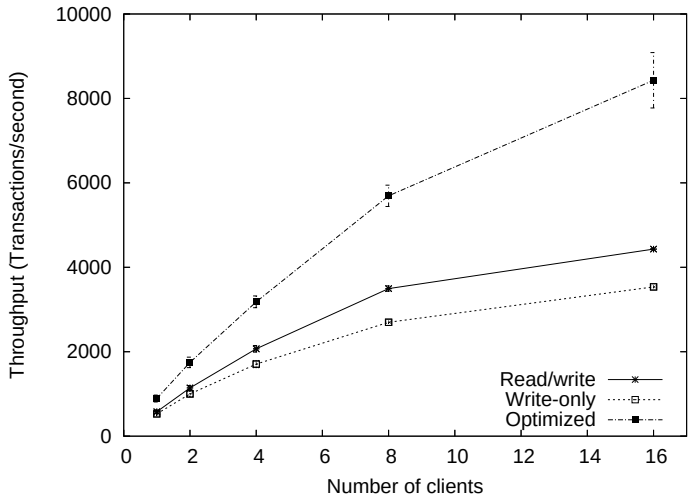
For $t > 1$, the behavior of our protocols depends strongly on the network setting. We can give some insight into the behavior of one network setting, where

the server replicas are on one local area, low latency network, and the clients are in other networks connected to the servers' network via a wide-area, higher latency network with a large variance in message delivery time. This is a typical setting for both Grid and commercial services.

In such a setting, with Paxos the latency of a client's request is determined primarily by the wide area network over which the client exchange messages only with the proposer (backup replicas ignore messages received directly from the client). Increasing t has a small effect on client latency. With X-Paxos, however, a client sends multiple requests across the wide-area network. Even when the requests are sent concurrently, the increased variance could result in performance degrading when t is increased. Experimental results bear out this intuition [33].



(a) 3 requests per transaction



(b) 5 requests per transaction

Figure VI.8: Transaction throughput on Sysnet

VI.D Related Work

Semi-passive replication, a variant of passive replication that can be implemented in the asynchronous system model without requiring an agreement on

the primary, was presented in [22]. This protocol uses the Chandra-Toueg $\diamondsuit S$ consensus algorithm [16] to implement the primary-backup approach. It uses the same idea of running consensus on both the command and the state update, but its practical implementation and performance remains uninvestigated.

Paxos [35, 36] provides the capability to replicate an arbitrary deterministic state machine in asynchronous message-passing systems. It does so by providing a safe, if not necessarily live, implementation of repeated consensus. Fast Paxos [37] saves one message delay compared with Paxos by having clients send commands directly to the acceptors, bypassing the leader. An acceptor interprets clients' messages as if it were an accept request from the leader for the next unused command number, that is, the command number that the acceptor believes to be the next unused one. Fast Paxos works well if all acceptors assign the same command number to a client's command. Otherwise, the processes may not choose any command, forcing the leader to intercede. Fast Paxos requires more replicas than Paxos to mask the same number of failures. Finally, Generalized Paxos [38] is a version of Paxos that solves the generalized consensus problem: agreement upon increasing partial orders of commands. By generalizing the Paxos algorithm, one can implement a system in which commands issued concurrently execute in two message delays if they are commutative, that is, it does not matter in which order the commands are executed. To our knowledge, no previous work has investigated the use Paxos for nondeterministic services.

The BFT protocol by Castro and Liskov (also known as Byzantine Paxos [40]) shares several goals and properties with the Classic Paxos algorithm, but tolerates Byzantine failures as opposed to only crash failures [15]. The BFT protocol proposes an optimization for reads, which consists of simply reading from a quorum of replicas. In contrast with X-Paxos, such an optimization only works when a read does not overlap any writes and all the replicas return the same value (the client does not read from a Byzantine replica), thus implying that read requests may not complete. If a read request does not complete, then the client has to re-issue the

read request using the regular, non-optimized BFT protocol. Although we have not considered thoroughly variants of the Paxos algorithm for Byzantine failures, an optimization similar to the one of X-Paxos should be applicable to protocols such as BFT.

[42] described a realization of distributed leader election. Progress is guaranteed in the weak setting where eventually one process can send messages such that every message obtains f timely responses (f is a resilience bound). An extension of this protocol provides leader stability which guarantees against arbitrary demotion of a qualified leader and avoids performance penalties associated with leader changes in schemes such as Paxos.

VI.E Summary

High availability of services, including grid and Web services, is important not only for business services (*e.g.*, e-commerce) but also for many high performance and scientific computing applications. Replication is a proven technique for ensuring high availability, but nondeterminism and the asynchronous nature of many such computing environments make the problem challenging. In this chapter, we present a replication protocol that can handle nondeterminism and asynchrony of the computing nodes and the underlying network. We show using experimental results that the overhead of replication is often reasonable, both in terms of response time and service throughput. Furthermore, we also present two optimized protocols, X-Paxos and T-Paxos, that reduce service response time and increase throughput by taking advantage of application-specific characteristics, namely, service requests that do not modify the service state (read-only) and applications that use transactions. X-Paxos and T-Paxos reduce replication overhead for configurations in which communication overhead between service replicas is not ignorable compared with that between client processes and service replicas. One case of such a scenario is when all client and service processes are located at

the same site. Another case is when service replicas are spread across wide-area networks to mask correlated failures. Experiments on a local cluster and PlanetLab demonstrate that X-Paxos and T-Paxos deliver significantly better performance in such settings.

VI.F Acknowledgements

This chapter is, in part, reprints of material as it appears in "Replicating Nondeterministic Services on Grid Environments," by Xianan Zhang, Flavio Junqueira, Matti A. Hiltunen, Keith Marzullo, and Richard D. Schlichting, in the Proceedings of the 15th IEEE International Symposium on High-Performance Distributed Computing (HPDC-15), June, 2006. The dissertation author was the primary coauthor and co-investigator of this paper.

Chapter VII

Conclusion

Given the fact that the grid resources are heterogeneous and often belong to multi-organizations, it is critical to build reliable grid services based on unreliable resources. Grid services are often nondeterministic and the primary-backup is the traditional approach to make nondeterministic services highly available. My thesis investigates how to replicate services using the primary-backup approach in grid environments.

First, my thesis presents the design of a primary-backup protocol using the grid standard – OGSI, and the implementation based on Globus Toolkit 3. The design and implementation shows that it is not hard to accommodate primary-backup on grid middleware, and the solution is simple and requires only small changes to the service. The use of the OGSI notification interface to handle replica updates is the key distinguishing feature of this approach. Using a simple example grid services, we compare the performance of this notification-based approach with variants in which replica update is done using standard grid service method calls and TCP. We found the overhead of using GT3 implementation of the OGSI notification is quite high. The overhead is particularly large in the cases where the state data is small or the number of clients is large.

Then my thesis addresses the issue of integrating the primary-backup replication with other techniques. We presented an architecture, in which the ser-

vice designer can control which portions of a service's state are persistent, the tradeoff between the degree of durability and the performance, and the atomicity of accesses to the service state with respect to failures. The performance measurements show that being careful about choosing the appropriate durability techniques can significantly boost performance, with the gain increasing as the number of state objects accessed in a single client invocation increases.

Finally my thesis studies how to replicate grid services on asynchronous systems. We evaluated the performance of the existing protocols – Classic Paxos and Fast Paxos. Although theoretically Fast Paxos is faster when there is no collisions, our simulation and experiments show that there are some structural details about Fast Paxos that can impact latency. We identified realistic scenarios in which Classic Paxos outperforms Fast Paxos. Based on our observation, we propose a new set of protocols that support both asynchronous systems and nondeterministic services. Using experiments on a local cluster and PlanetLab, we shows that our new protocol provides good performance for replicated nondeterministic grid services.

Bibliography

[1] P. Alsberg and J. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 562–570, Oct 1976.

[2] Y. Amir, B. Awerbuch, and R. S. Borgstrom. Managing checkpoints for parallel programs. In *Proceedings of the 1st International Conference on Information and Computation Economies (ICE-98)*, 1998.

[3] Y. Amir and C. Tutu. From total order to database replication. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, July 2002.

[4] A. Amoroso and K. Marzullo. Multiple job scheduling in a connection-limited data parallel system. *IEEE Transactions on Parallel and Distributed Systems*, pages 125–134, February 2006.

[5] C. Amza, A. L. Cox, and W. Zwaenepoel. Distributed versioning: Consistent replication for scaling back-end databases of dynamic content web sites. In *Proceedings of the fourth ACM/IFIP/USENIX International Conference on Middleware*, June 2003.

[6] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1:11–33, Jan 2004.

[7] C. Bauer and G. King. *Hibernate in Action*. Manning Publishing Company, Aug 2004.

[8] F. Berman, A. Chien, K. Cooper, J. Dongarra, I. Foster, D. Gannon, L. Johnsson, K. Kennedy, C. Kesselman, J. Mellor-Crummey, D. Reed, L. Torczon, and R. Wolski. The GrADS Project: Software support for high-level grid application development. *International Journal of High Performance Computing Applications*, 15(4):327–344, December 2001.

[9] K. Birman, T. Joseph, T. Raeuchle, and A. Abbadi. Implementing fault-tolerant distributed objects. *IEEE Transactions on Software Engineering*, SE-11(6):502–508, Jun 1985.

[10] S. Bodoff, D. Green, K. Haase, E. Jendrock, M. Pawlan, and B. Stearns. *The J2EE Tutorial*. Addison-Wesley, Mar 2002.

[11] N. Budhiraja, K. Marzullo, F. Schneider, and S. Toueg. Primary-backup protocols: Lower bounds and optimal implementations. In *Proceedings of the 3rd IFIP Working Conference on Dependable Computing for Critical Applications*, pages 187–198. Springer-Verlag, Wien, 1992.

[12] L. Felipe Cabrera, G. Copeland, W. Cox, T. Freund, J. Klein, D. Langworthy, I. Robinson, T. Storey, and S. Thatte. Web Service BusinessActivity (WS-BusinessActivity), Nov 2004. http://www6.software.ibm.com/-software/developer/library/ws-busact200401.pdf.

[13] L. Felipe Cabrera, G. Copeland, M. Feingold, T. Freund, J. Johnson, C. Kaler, J. Klein, D. Langworthy, A. Nadalin, D. Orchard, I. Robinson, J. Shewchuk, and T. Storey. Web Service Coordination (WS-Coordination), Nov 2004. http://www6.software.ibm.com/software/developer/library/WS-Coordination.pdf.

[14] L. Felipe Cabrera, G. Copeland, M. Feingold, T. Freund, J. Johnson, C. Kaler, J. Klein, D. Langworthy, A. Nadalin, D. Orchard, I. Robinson, T. Storey, and S. Thatte. Web Service AtomicTransaction (WS-AtomicTransaction), Nov 2004. http://www6.software.ibm.com/software/-developer/library/WS-AtomicTransaction.pdf.

[15] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, November 2002.

[16] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[17] B. Choi, S. Moon, Z. Zhang, K. Papagiannaki, and C. Diot. Analysis of point-to-point packet delay in an operational network. In *Proceedings of IEEE INFOCOM*, volume 3, pages 1797–1807, March 2004.

[18] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. PlanetLab: An overlay testbed for broad-coverage services. *ACM Computer Communications Review*, 33(3), July 2003.

[19] K. Czajkowski, D. Ferguson, I. Foster, J. Frey, S. Graham, I. Sedukhin, D. Snelling, S. Tuecke, and W. Vambenepe. The WS-Resource Framework, Jan 2004. http://www.globus.org/wsrf/specs/ws-wsrf.pdf.

[20] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid information services for distributed resource sharing. In *Proceedings of the Tenth IEEE International Symposium on High Performance Distributed Computing (HPDC-10)*, 2001.

[21] K. Czajkowski, I. Foster, and C. Kesselman. Resource Co-Allocation in computational grids. In *Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing (HPDC-8)*, 1999.

[22] X. Defago, A. Schiper, and N. Sergent. Semi-passive replication. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems (SRDS)*, 1998.

[23] D. L. Eager, E. D. Lazowska, and J. Zahorjan. Adaptive load scharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering*, 12(5), 1986.

[24] E. N. Elnozahy, L. Alvisi, Y. Wang, and D. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, pages 375–408, September 2002.

[25] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, Apr 1985.

[26] I. Foster. The grid: A new infrastructure for 21st century science. *Physics Today*, 2002.

[27] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, 1997.

[28] A. Galstyan, K. Czajkowski, and K. Lerman. Resource allocation in the grid using reinforcement learning. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS-04)*, 2004.

[29] S. Graham, P. Niblett, D. Chappell, A. Lewis, N. Nagaratham, J. Parikh, S. Patil, S. Samdarshi, S. Tuecke, W. Vambenepe, and B. Weihl. Web Service Notification (WS-Notification), Jan 2004. http://www.ibm.com/-developerworks/library/ws-resource/ws-notification.pdf.

[30] Grid physics network. http://www.griphyn.org/.

[31] D. Gunter, B. Tierney, K. Jackson, J. Lee, and M. Stoufer. Dynamic monitoring of high-performance distributed applications. In *Proceedings of the Eleventh IEEE International Symposium on High Performance Distributed Computing (HPDC-11)*, 2002.

[32] K. Joshi, M. Hiltunen, W. Sanders, and R. Schlichting. Automatic model-driven recovery in distributed systems. In *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems (SRDS 2005)*, pages 25–36, Oct 2005.

[33] F. Junqueira and K. Marzullo. Coterie availability in sites. In *Proceedings of DISC*, pages 2–16, September 2005.

[34] K. Keeton and A. Merchant. A framework for evaluating storage system dependability. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN-2004)*, 2004.

[35] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[36] L. Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column)*, 32(4):18–25, 2001.

[37] L. Lamport. Fast Paxos. Technical Report MSR-TR-2005-112, Microsoft Research, July 2005.

[38] L. Lamport. Generalized consensus and Paxos. Technical Report MSR-TR-2005-112, Microsoft Research, July 2005.

[39] B. Lampson. Atomic transactions. In *Distributed System-Architecture and Implementation*, pages 246–265. Springer-Verlag, 1981.

[40] B. Lampson. ABCD's of Paxos. In *Proceedings of the 20th ACM Symposium on Principles of Distributed Computing (PODC)*, page 13, August 2001. Full technical report at http://research.microsoft.com/Lampson/65-ABCDPaxos/Abstract.html.

[41] D. Liang, C.-H. Fang, C. Chen, and F. Lin. Fault tolerant web service. In *Proceedings of the 10th Asia-Pacific Software Engineering Conference (APSEC'03)*, pages 310–319, Dec 2003.

[42] D. Malkhi, F. Oprea, and L. Zhou. Ω meets Paxos: Leader election and stability without eventual timely links. In *Proceedings of the 19th International Symposium on Distributed Computing (DISC)*, pages 199–213, September 2005.

[43] J. M. Milan-Franco, R. Jiménez-Peris, M. Patino-Martínez, and B. Kemme. Adaptive middleware for data replication. In *Proceedings of the fifth ACM/IFIP/USENIX International Conference on Middleware*, Oct 2004.

[44] M. Mitzenmacher. On the analysis of randomized load balancing schemes. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 292–301, 1997.

[45] Object Managemnt Group. Fault tolerant CORBA. In *Common Object Request Broker Architecture: Core Specification*, chapter 23, pages 955–1059. Object Management Group, Dec 2002.

[46] C. Plattner and G. Alonso. Ganymed: Scalable replication for transactional web applications. In *Proceedings of the fifth ACM/IFIP/USENIX International Conference on Middleware*, Oct 2004.

[47] Quest Software. Big brother professional edition, 2005. http://www.quest.com/bigbrother/.

[48] A. Rajasekar, M. Wan, R. Moore, W. Schroeder, G. Kremenek, A. Jagatheesan, C. Cowart, B. Zhu, S. Chen, and R. Olschanowsky. Storage resource broker - managing distributed data in a grid. *Computer Science of India Journal, Special Issue on SAN*, 33(4):42–54, Oct 2003.

[49] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed resource management for high throughput computing. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing (HPDC-7)*, 1998.

[50] M. Raynal. Consensus in synchronous systems: A concise guided tour. In *Proceedings of the 2002 Pacific Rim International Symposium on Dependable Computing (PRDC'02)*, 2002.

[51] W. Reed. The Pareto, Zipf, and other power laws. *Economics Letters*, 74:15–19, December 2001.

[52] L. Rodrigues, H. Miranda, R. Almeida, J. Martins, and P. Vicente. Strong replication in the globdata middleware. In *Proceedings of the Workshop on Dependable Middleware-Based Systems*, June 2002.

[53] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, pages 299–319, December 1990.

[54] F. Taiani, M. Hiltunen, and R. Schlichting. The impact of web services integration on grid performance. In *Proceedings of the Fourteenth IEEE International Symposium on High Performance Distributed Computing (HPDC-14)*, pages 14–23, 2005.

[55] The TeraGrid project. http://www.teragrid.org/.

[56] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, T. Maquire, T. Sandholm, D. Snelling, and P. Vanderbilt. Open Grid Services Infrastructure (OGSI), June 2003. http://xml.coverpages.org/OGSI-SpecificationV110.pdf.

[57] Veritas company homepage. http://www.veritas.com/index.html.

[58] R. Wolski, N. Spring, and C. Peterson. Implementing a performance forecasting system for metacomputing: The network weather service. In *Proceedings of the SuperComputing 1997 (SC97)*, 1997.

[59] H. Wu, B. Kemme, and V. Maverick. Eager replication for stateful J2EE servers. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA)*, Oct 2004.

[60] D. Zagorodnov, K. Marzullo, L. Alvisi, and T.C. Bressoud. Engineering fault-tolerant TCP/IP servers using FT-TCP. In *Proc. IEEE Intl. Conf. on Dependable Systems and Networks (DSN)*, pages 393–402, San Francisco, California, USA, June 2003.

[61] X. Zhang, D. Zagorodnov, M. Hiltunen, K. Marzullo, and R. Schlichting. Fault-tolerant grid services using primary-backup: Feasibility and performance. In *Proceedings of the 2004 IEEE International Conference on Cluster Computing (Cluster 2004)*, Sep 2004.

[62] S. Zhou. A trace-driven simulation study of dynamic load balancing. *IEEE Transactions on Software Engineering*, 14(9), 1988.