AN ADAPTIVE QUALITY OF SERVICE AWARE MIDDLEWARE FOR REPLICATED SERVICES

BY

SUDHA KRISHNAMURTHY

B.S., Bangalore University, 1990
M.E., Indian Institute of Science, 1994
M.S., Ohio University, 1996

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2002

Urbana, Illinois

Abstract

A dependable middleware should be able to adaptively share the distributed resources it manages in order to meet diverse application requirements, even when the quality of service is degraded due to uncertain variations in load and unanticipated faults. We have addressed this issue in the context of a dependable middleware that adaptively manages replicated servers to deliver a timely and consistent response to time-sensitive client applications. These applications have specific temporal and consistency requirements, and can tolerate a certain degree of relaxed consistency in exchange for better response time.

We propose a flexible QoS model that allows clients to specify their temporal and consistency constraints. Since unanticipated faults and transient overloads cause uncertainty in most distributed environments, it is hard to provide deterministic guarantees for meeting the timeliness requirements of applications. Hence, our approach provides probabilistic timeliness guarantees. The approach dynamically selects replicas to service a client's request based on the prediction made by probabilistic models. These models use the performance history collected by monitoring the replicas at runtime to predict the ability of a replica to meet a client's QoS specification.

In this thesis we describe the adaptive framework and present the probabilistic models we have developed for the cases in which the replicated content is either static or dynamic. We discuss experimental results that validate the models, and compare the timeliness/consistency tradeoffs under different scenarios. The experimental results demonstrate the role of feedback and the efficacy of simple analytical models for adaptively sharing the available replicas among the users. Given the increasing demand for different services and the diversity of the requirements of clients, such adaptive frameworks that rely on feedback-based resource allocation are likely to play an increasing role in a range of problems related to building dependable systems.

To my parents

Acknowledgments

Life in graduate school has been a long, but adventurous journey that has taken me through numerous valleys, several plains, and a few peaks. I would now like to acknowledge the people whom I have met during the course of this journey and who have enriched my experience.

I would like to begin by thanking my thesis advisor, Prof. Bill Sanders, for giving me the opportunity to work in his research group. Although I approached him rather late in my graduate life, I am thankful that he was able to coordinate so I could graduate in a timely manner. I thank him for his feedback, cheerfulness, generous support, and encouragement. More importantly, I am grateful to him for giving me the opportunity to think independently and present my work on several occasions. I would also like to express my thanks to the rest of my committee members, Prof. Geneva Belford, Prof. Klara Nahrstedt, and Prof. Vikram Adve, for agreeing to be on my committee, for their insightful questions, and for their feedback. I would like to thank Michel Cukier for participating in the research meetings and for his timely feedback.

I have been fortunate to work in a group filled with amiable individuals. It has truly been a pleasure interacting with the ”impeccable” Jenny Applequist; her quiet dedication, efficiency, and attention to detail continue to amaze me. I am very grateful to her for reviewing all of my documents and giving me careful feedback. I thank both Bill and Jenny for maintaining an excellent collection of books in the group library, which provided easy access to information. It is rare to find someone who is willing to spend the time listening to half-baked ideas and critiquing them. I have been fortunate that my group member, Kaustubh Joshi, was willing to do that for me on many occasions. His insights and constructive feedback have always helped clarify my understanding. I would like to thank Jennifer Ren and Paul Rubel for getting me started with AQuA. I am grateful to Mouna Seri and Tod Courtney, for

their contributions to the AQuA project. Salem Derisavi, David Daly, and Graham Clark have helped clarify my probability-related doubts on many occasions, and I owe them my thanks for the many useful discussions I have had with them. I would also like to thank Vinh Lam, Ryan Lefever, James Lyons, Hari Ramasamy, Vishu Gupta, and Sankalp Singh for their friendship, and for making life in the PERFORM group an enjoyable and fun experience.

Working in Prof. Andrew Chien's group during the initial years of my graduate life was an educational experience in different ways. The weekly group discussions, which provided opportunities for graduate students to present and critique the latest research ideas, enabled me to be aware of the research being conducted in a range of different areas.

I owe my gratitude to my parents, sisters, and brother for their patience, support, and confidence in me through all these years. I am extremely thankful to my parents for allowing me the freedom to pursue the directions that I chose.

Finally, the research described in this thesis was funded by DARPA grants F30602-98-C-0187 and F30602-00-C-0172, and I am grateful to DARPA for their support.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

The last thing one knows when writing a book is what to put first.

- Blaise Pascal, Pensees.

In the last few decades, significant advances in networking technology and networking protocols have spurred the growth of distributed computing and resulted in the emergence of a diverse range of useful distributed services, such as distributed directory services, search engines, remote patient monitoring applications, navigational systems, multimedia applications, online auctions, and stock trading applications. These services are distributed across either local area networks (LAN) or wide area environments. The users of distributed services deployed in either of these environments have to contend with different degrees of unpredictability. Unpredictability may arise because of several factors, such as unanticipated faults in the system or transient overloads caused by sharing the network and computing resources with other users and applications.

Owing to the unpredictability, the users of distributed services often benefit when, in addition to being allowed to access the services, they are also allowed to control the quality of service (QoS) that they receive. A user may be interested in quality of service along multiple dimensions, depending on the specific service being accessed. For example, multimedia applications may be interested in specifying fine-grained QoS measures, such as network bandwidth, error rates, latency, and jitter, that pertain to network-level resources. Some users may be interested in controlling the QoS they receive from operating system resources, such as the memory bandwidth, disk bandwidth, and processor cycles. Finally, many applications that do not have detailed knowledge about the underlying resources may find

it more useful if they are allowed to request end-to-end quality of service guarantees using higher level measures, such as timely response, dependability, security, precision, and resolution [45]. In the last case, the burden of meaningfully mapping the requirements specified at a higher level of abstraction onto specific properties of the underlying resources falls on a layer, called a *middleware* layer, that acts as an interface between the application and the resources.

1.1 QoS-Aware Middleware

A significant amount of research and commercial activity in the area of middleware computing in the last decade has made the use of middleware in developing distributed applications more viable. However, the focus of most of the well-known middleware, like CORBA [10], DCOM [31], and Java [35], has been mostly on supporting the needs of general-purpose distributed applications by providing location transparency and supporting heterogeneity and interoperability. While those middleware simplify the task of developing distributed applications by hiding low-level details about the underlying communication subsystem, it is only recently that efforts have been made to enhance these middleware with features to support special-purpose applications. For example, the CORBA specification has been enhanced to include multithreading, priority-based scheduling, and controlled priority inversion at the ORB level to support real-time and embedded applications [74, 68]. Similar efforts have been made to augment Java to support real-time features [66, 27]. A recently adopted CORBA specification also provides directives for constructing fault-tolerant CORBA objects using replication [24]. Orthogonal to the above efforts, which accommodate the needs of modern applications by enhancing existing standards, are efforts that are being made to build newer middleware and enhance existing middleware to support novel features. One such effort that is relevant to this thesis is that of building middleware that is *QoS-aware*. There are at least three main challenges involved in building a QoS-aware middleware:

- The middleware should allow the clients to express their application-specific requirements using the right level of abstraction. Choosing the right level of abstraction, in a way that would allow the applications enough control, without overburdening them with details about the low-level

resources, is important.

- The middleware should be able to relate the application-specific requirements to the properties of the resources it manages, so that those requirements can guide the middleware in effectively sharing the resources among the users.

- Owing to the unpredictability of the environment, the middleware should be able to adapt the allocation of resources to the users to meet their requirements, based on feedback from the environment.

1.1.1 QoS-Based Resource Management

We now briefly compare and contrast some of the related research efforts that have proposed middleware-based solutions for QoS-aware resource management in the context of different applications. Some of these efforts have built stand-alone middleware, whereas others have enhanced the capabilities of existing middleware with additional layers in order to provide QoS-based resource management. While some of these approaches manage the allocation of a single resource, other strategies co-allocate multiple resources. Furthermore, some of the solutions use resource reservation and admission control for providing QoS guarantees. In contrast, others use a more flexible and adaptive approach. In some cases, the adaptation may be done in an application-transparent manner, while in other cases, the middleware may inform the users about changes in the resources and adapt based on user-level feedback.

Agilos [46] is a CORBA-based middleware whose objective is to adaptively allocate resources, such as network bandwidth and CPU cycles, to meet application-specific requirements as well as system-specific requirements, like stability and fairness. It uses a generic, control-theoretic framework to realize its objective. Mobiware [8] is a QoS-aware middleware that leverages CORBA technology to support mobile multimedia applications. It provides QoS-controlled handoff algorithms, and adaptively scales the transport of continuous media over wireless networks, based on the application's traffic class, delay, and bandwidth requirements.

Unlike Agilos and Mobiware, which leverage existing middleware technology, the Compose|Q [54]

is a standalone middleware framework that uses a two-level scheduler to provide timeliness and QoS guarantees for soft real-time applications. A global scheduler schedules tasks across nodes in a distributed environment and a local scheduler schedules tasks on a node. Both the schedulers collaborate to adapt to changing system and network conditions by runtime monitoring. The middleware supports standard real-time scheduling algorithms, such as rate-monotonic and earliest deadline first algorithms, as well as priority-based scheduling. Adaptware [1] is another QoS-adaptive middleware that uses a feedback-based control-theoretic approach to schedule resources. Clients specify a range of acceptable QoS through QoS contracts with the server. Adaptware uses those contracts to adapt the QoS delivered to a client, so as to maximize the aggregate utilization of resources. Globus [16, 17] is a more ambitious QoS-aware middleware that is geared towards large-scale deployment. Globus manages the network, CPU, and disk resources for a variety of distributed applications. It manages the co-allocation of multiple resources by combining features of a reservation-based approach and an adaptation-based approach. Applications specify their QoS requirements at a fairly low level of abstraction, which allows a straightforward mapping between the requirements and the resources. Each resource is associated with a resource manager that monitors and controls the reservation of that resource. Sensors monitor the different resources at runtime and provide the feedback that initiates adaptation.

1.2 Problem Description

In this thesis, we are interested in the design of middleware-based solutions for providing access to distributed services based on the QoS requirements of the clients. Unlike some of the well-known middleware that allow access to services using a directory lookup, based on static properties such as name, interface, and location, we are particularly interested in the quality of service requirements that depend on *dynamic* properties of the servers. Specifically, we are interested in targeting *time-sensitive* clients. Our goal is to develop a middleware-based approach to mediate a client's access and to allocate servers based on their ability to meet the quality of service requirements of the client. Simple as the goal seems, the problem is challenging, because the timeliness of a service depends

on the performance characteristics of the servers, the distributed environment in which these services are deployed, and the number of users accessing a service. All of these factors vary with time in an unpredictable manner. As such, access to servers that is based on a simple directory lookup will not suffice for meeting the temporal constraints. Rather, the lookup has to be based on actively monitoring the changes in the dynamic properties of the servers. Further, in order to cope with the unpredictability, the middleware has to be designed to meet the demands of the clients under stable conditions as well as when there is a change in the availability of a service due to transient overloads and server failures. In short, the middleware has to be adaptive in order to provide both fault tolerance and timeliness. Fault tolerance and timeliness are important for many performance-critical systems, such as missile control units, sensor-based tracking applications, and navigational systems. Fault tolerance and timeliness guarantees are also beneficial to users of many non-critical services, such as directory and database servers, web servers, and multimedia applications.

1.2.1 Challenges in Replica Selection

A well-known approach for providing fault-tolerant and responsive services is *replication*. Replicating the servers provides robustness in times of failure by allowing access to the service even when some of the servers are not functioning and improves the response time by allowing multiple clients to be serviced concurrently. However, replication by itself is not a solution for meeting the different QoS requirements. Rather, the available replica resources have to be *managed* and *allocated* to service the clients, based on the QoS requested by the clients. This requires an understanding of the tradeoffs between the different quality of service measures and an ability to map the requirements appropriately onto properties of the replicated resources. This mapping is often not straightforward, especially when some of the QoS requirements may be conflicting. For example, in order to provide good fault tolerance, we could allocate all the available replicas to service a client (e.g., [4, 25, 47, 11, 55, 15]). However, such an approach would not be scalable, as it would increase the load on all the replicas and result in higher response times for the remaining clients. On the other hand, assigning a single replica to service each client would allow multiple clients to be serviced concurrently. However, if the replica

failed while servicing a request, the failure might result in an unacceptable delay for the client being serviced. Hence, neither approach is suitable when a client has specific timing constraints and failure to meet those constraints results in a penalty for the client.

Furthermore, when the replicated state is modified by the clients, there is the additional challenge of permitting client operations to execute with the greatest possible concurrency to provide good response times, while ensuring that the replicated state does not diverge in an uncontrolled manner. We can ensure that the replicated state remains coherent by forcing all the replicas to commit the modifications at the same time. However, such a strategy limits the degree of concurrency and results in reduced responsiveness.

Finally, when the currently available replicas are insufficient to meet the requirements of the clients, the middleware has to decide how to react appropriately. For example, should the middleware inform the client applications about the insufficiency and leave it to them to adapt? Should it limit the number of clients that it admits? Should it increase the size of the replica pool? If the middleware decides to add more replicas, it has to also decide how many to add and where to place them.

To summarize, in order to build a dependable, QoS-aware middleware for meeting a client's QoS specification, we need an approach that adaptively selects the replicas from the available replica pool. The replicas must be chosen to service the client, based on an understanding of the client's requirements and the dynamic properties of the replicas. Furthermore, we also need to enable the middleware to react appropriately when the available replicas are insufficient to meet the demands of the clients.

1.3 Our Solution: QoS Model for Accessing Replicated Objects

The approach we use to address the aforementioned issues is based on a QoS model. The QoS model we propose allows a broad spectrum of applications to express their requirements at a fairly high level of abstraction using a uniform interface. The model does not require the applications to provide detailed specifications about the underlying resources (e.g., network bandwidth, storage, and CPU cycles). In fact, even replication is provided in an application-transparent manner, and clients are given the per-

ception that they are communicating with a single server, although their requests may in reality be processed by multiple replicas. Applications may either specify their QoS requirements at start-up time or negotiate them at runtime as often as they want. In order to provide access to replicated servers, we are mainly interested in providing quality of service along two dimensions: timeliness of response and consistency of replicated data.

Timeliness: Time-sensitive applications require timely execution of operations and a timely responses to their requests. To support such clients, our QoS model allows a client to specify the time by which it wants to receive a response after it transmits its request to a service. If the requested *response time* is not met then it results in a *timing failure* for the client. In an unpredictable distributed environment, it is impossible to provide deterministic guarantees for meeting the temporal requirements. Instead, our aim is to provide probabilistic guarantees. To achieve this, our QoS model allows a client to specify its response time as well as the *minimum probability* with which it wants its time constraint to be met. The advantage of this probabilistic QoS model is that it allows the temporal requirements of applications to be treated as a continuous spectrum, instead of classifying them as hard real-time and soft real-time, as is done conventionally. In addition, the specified probability can be interpreted as a client's importance. For example, a time-sensitive application that wants its temporal requirements to be met with a high priority can specify a probability value close to 1, while applications that have lower priorities can specify a smaller probability value.

Consistency: Replica inconsistency may arise when multiple clients access an object concurrently and some of the accesses result in modifications to the replicated state. In order for the responses to be meaningful to the clients, it is important to bound the degree of inconsistency when the replicated information is time-varying. Traditionally, replicated systems provide either strong consistency (e.g., [55, 63, 67]) or weak consistency (e.g., [77, 37, 21]) guarantees for replicated data. Strong replica consistency requires that at the end of each method execution by a replicated object, all the replicas of the object have the same state. Weak consistency specifies that the replicated state will eventually converge, but does not provide any guarantees about when the convergence

will be achieved. We believe that instead of using qualitative measures, such as strong and weak consistency, several applications will benefit from intermediate degrees of consistency that can be more precisely quantified [76, 83, 60]. Therefore, our QoS model uses a combination of qualitative and quantitative measures that allows us to tune the consistency delivered to an application across a wider range of options.

Since different applications have different views of consistency, it is hard to capture the different consistency requirements using a single metric. Hence, the QoS model we propose treats consistency requirements along two dimensions: *order* and *staleness threshold*. The order guarantee prevents conflicts between operations, and bounds the degree of inconsistency in the replicated state in a way that is semantically meaningful for the users. The staleness threshold indicates the degree to which a client is willing to tolerate relaxed consistency, and quantifies the staleness in the information retrieved by the clients. Our approach measures staleness using "logical" time [44], since this obviates the need for synchronized clocks. However, it should be possible to develop an equivalent approach that measures staleness using real-time clocks. Like the timeliness QoS model described above, the consistency QoS specification accommodates the needs of a broad spectrum of applications. For example, a client that requires strong consistency can request sequential ordering with staleness 0. On the other hand, in a scenario in which the replicated state is either absent or static (for example, when the client transactions are read-only), clients can allow their accesses to be unordered, and ignore the staleness threshold.

We have used the AQuA middleware [11] to implement our work. Previous work in AQuA transparently replicates CORBA objects to cater to applications that require access to fault-tolerant distributed services. Users are allowed to specify the number of crash failures and value faults they want an object to tolerate. Using a rather straightforward mapping, AQuA maps these specifications to an appropriate degree of replication for the object. Our research leverages the AQuA infrastructure to make it a QoS-aware middleware. Since our motivation for replicating objects is to provide fault tolerance and timeliness, our goal is to meet the above QoS requirements even when the availability of a service is degraded due to the failure of a replica. Unlike the timeliness and consistency QoS dimensions, we

do not explicitly consider fault tolerance as a basis for accessing replicated resources. Rather, in this thesis, we address fault tolerance implicitly by developing protocols to ensure that the timeliness and consistency requirements can be met even when replica failures occur.

1.3.1 Application-Aware Adaptation

Given the diversity of the QoS requirements of today's distributed applications, rather than define a "one size fits all" approach for managing the replicated resources, we propose a middleware-based framework that allows us to construct customized protocols tailored to the semantics of specific applications. We then plug these protocols into a middleware to provide an integrated framework.

The framework we propose uses the aforementioned QoS specifications to choose the appropriate protocol for adaptively managing the replica resources. The chosen protocol uses simple analytical models to establish a relationship between a client's QoS specification and properties of the replicas. Although more detailed models would have defined this relationship more accurately, we decided to use simpler models because the overheads involved in computing more detailed models online may be considerable. These overheads may not justify the use of complex models, especially for time-sensitive applications.

Since we are interested in providing QoS guarantees that depend on the dynamic properties of the replicas, our framework supports online monitoring of these dynamic properties. When a client accesses a service, the middleware-based framework first transparently intercepts the request. It then supplies the information gathered from the online monitoring of resources as inputs to the analytical model used by the chosen protocol. The model then guides the middleware in adaptively choosing a set of replicas to service the request based on the ability of the replicas to meet the client's QoS requirements. We choose a set of replicas in such a way that we can improve the chances of satisfying a client's requirements despite the unpredictability of our environment. Hence, one of the key decisions our distributed request scheduling framework has to make is that of choosing the appropriate redundancy level for servicing a request. If the middleware is unable to meet the QoS requirements owing to insufficient replicas, it either creates additional replicas or informs the clients through a callback and

allows them to adapt.

In summary, we propose a framework in which we develop protocols that manage replicated resources using an application-aware model of adaptation [57]. Such a model, in which the middleware and application collaborate to adapt, is attractive because it supports application diversity and concurrency. We accommodate diversity by allowing the applications to determine the mapping between their QoS requirements and the replicated resources, and we support concurrency by controlling the resource arbitration and monitoring at the middleware layer.

1.3.2 Research Summary

Figure 1.1 puts our research into perspective. Previous work in AQuA has provided fault tolerance for applications that do not have specific time constraints. Our research has resulted in the following enhancements to the AQuA infrastructure.

We have designed and implemented an adaptive replica allocation scheme that uses a probabilistic approach for providing temporal guarantees to the clients, under the assumption that the clients do not have any explicit consistency requirements. Consistency guarantees are not useful when the replicas are stateless, as would be the case, for example, when the replicas are compute servers that perform large computations using the data supplied by the clients as inputs. Even when applications have replicated state, consistency is not useful if the servers do not permit update transactions, causing the replicated state to be static. For example, many of the search engines and directory servers export interfaces that allow a user to look up and download information from a large database. They seldom export interfaces that allow a client to modify the information in the database. The allocation scheme we have designed is useful in such scenarios. We have also experimentally analyzed the performance of the probabilistic scheme, and compared its performance with a static and round-robin allocation algorithm, in order to understand how the temporal guarantees are affected by factors such as transient overload, replica crashes, and the redundancy level chosen to process a request. The "timeliness profiles" so obtained can be used to predict the ability of the middleware to meet the temporal requirements for future requests.

We have also addressed the replica allocation problem when the replicated state is time-varying. The

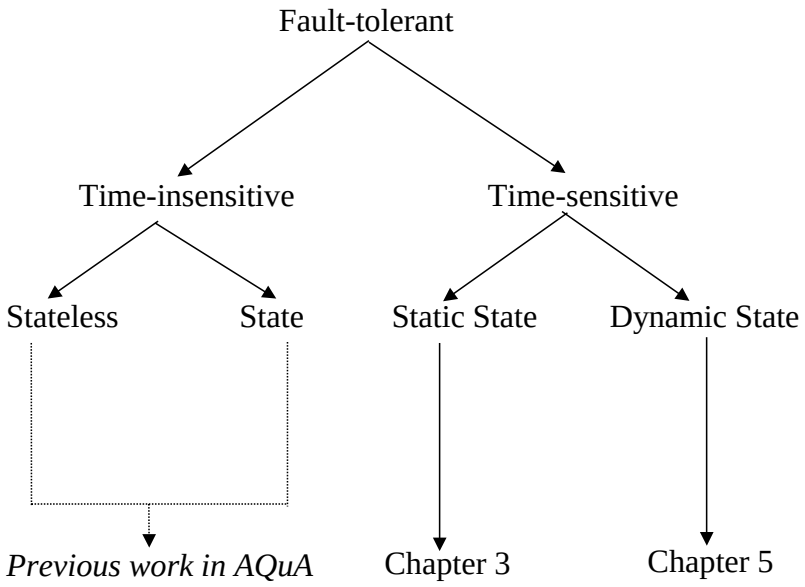


Figure 1.1: Research in perspective

time-varying nature of the replicated state may result in some of the replicas having obsolete state. We target time-sensitive applications that can tolerate a certain degree of relaxed consistency in exchange for better response time. Some of the motivating applications include real-time database applications, such as electronic patient recording systems and ticket reservation systems. Different clients that retrieve information from such services may have different consistency and timeliness requirements. In order to select replicas to meet those requirements, we need to take into account the state of a replica when estimating its responsiveness. To do this, we have developed an adaptive framework that supports tunable consistency and timeliness. The motivation behind this framework is that there is a fundamental tradeoff between timeliness and consistency when using a replicated service. We have used our framework to experimentally evaluate the effectiveness and adaptability of our replica allocation scheme. We have also conducted experiments to analyze the tradeoffs between timeliness and consistency, in order to address a number of pertinent questions. For example, our experimental study can help us determine how to predict the effect that selecting a particular consistency guarantee will have on the timeliness of a response.

Our research has also addressed the problem of creating replicas on demand, when the currently available replicas are insufficient to meet the QoS requirements of the clients.

1.4 Research Accomplishments

The research we have conducted has broadly focused on developing a set of QoS-aware protocols that adaptively manage replicated resources at the middleware layer. The main contribution of this thesis is the definition and experimental validation of a probabilistic model-based approach that uses runtime measurements to guide the adaptation. The research we have conducted has resulted in the following accomplishments:

QoS Model: We have defined a QoS model that allows applications to access replicated services based on their timeliness, consistency, and fault tolerance requirements.

Online Performance Monitoring: We have implemented a distributed framework that allows the performance of replicas to be monitored and recorded at the middleware layer. The performance updates are broadcast using a publish/subscribe model.

Probabilistic Temporal Guarantees: We have defined a probabilistic model that predicts the responsiveness of a replica based on its performance history. We have developed an adaptive replica selection algorithm that uses this prediction to select replicas with static state, in order to meet the timeliness requirements of applications. We have implemented the probabilistic selection algorithm, a round-robin selection scheme, and a static selection scheme as components of the AQuA middleware. We have experimentally validated the probabilistic model and conducted detailed experiments to compare the effectiveness of the probabilistic scheme with that of the static and round-robin schemes in different scenarios.

Tunable Consistency and Timeliness: We have designed and implemented an adaptive framework that allows clients to retrieve information based on their timeliness and consistency requirements. The framework allows applications to express their timeliness and consistency requirements at a fairly high level of abstraction. It supports tunable consistency by organizing the replicas into a hierarchical structure, with different tiers implementing different degrees of consistency. It bounds the divergence in the replicated state using a combination of immediate and lazy up-

date propagation. The framework also allows us to implement protocols tailored to different application-specific consistency semantics. As a proof of concept, we have used this framework to implement consistency semantics using sequential and FIFO ordering.

Analysis of Timeliness/Consistency Tradeoffs: We have used the above framework to experimentally analyze the tradeoffs between timeliness and consistency in a replicated service in different scenarios.

Dynamic Replica Creation: Our middleware framework has the ability to create replicas on demand. The key decisions involved are how many replicas to create, when to create them, and where to place them. The framework must also decide how to initialize the state of the new replica.

1.5 Chapter Organization

The remainder of this thesis is organized as follows.

In Chapter 2 we provide an overview of the AQuA architecture and describe previous work conducted using this infrastructure.

In Chapter 3 we describe our approach for providing probabilistic temporal guarantees in the absence of consistency requirements. We also present a detailed experimental analysis of this approach.

In Chapter 4 we outline the key challenges in maintaining replica consistency and survey some of the relaxed consistency models proposed by other researchers.

In Chapter 5 we describe the adaptive, fault-tolerant framework we have designed for providing tunable consistency and timeliness. We describe the QoS model, the hierarchical replica organization, and the probabilistic approach we have developed for meeting the timeliness and consistency requirements. We also describe how the framework allows us to implement different consistency semantics at the middleware layer, using sequential and FIFO ordering guarantees as examples.

In Chapter 6 we provide an experimental evaluation of our framework, and discuss the effectiveness of our probabilistic approach, by presenting results that compare the timeliness/consistency tradeoffs for sequential and FIFO ordering under different scenarios.

In Chapter 7 we describe the approach our middleware framework uses to create replicas on demand.

Finally, we present our conclusions and some key directions for future research in Chapter 8.

Chapter 2

Background : An Overview of AQuA

Though no one can go back and make a brand new start, anyone can start from now and make a brand new end.

- Carl Bard

The AQuA middleware leverages the capabilities of CORBA objects [10] to provide adaptive fault tolerance for distributed applications. Fault tolerance is provided by the transparent replication of objects using *active* [63] and *passive* replication [67] schemes. Replicas offering the same service are organized into a group. Communication between members of a group takes place through the Maestro-Ensemble [80, 28] protocol stack, above which AQuA is layered. The use of group communication in AQuA is transparent to the end applications. This is achieved using an AQuA gateway, which transparently intercepts a local application's CORBA message and forwards it to the destination replica group through Maestro-Ensemble. The different replication schemes supported by AQuA are implemented as *protocol handlers* within the gateway. These handlers are responsible for tolerating different kinds of faults. Previous work in AQuA has addressed the issue of tolerating crash failures using the active [69] and passive [67] handlers. [64] also discusses how AQuA simultaneously tolerates value faults and crash failures using an active handler. In this chapter, we discuss the architectural components of AQuA and describe the replication mechanisms supported by AQuA.

2.1 AQuA Application

An AQuA application is a typical client or server application, implemented as a CORBA object. The specific implementation of CORBA that we have used in this research is The ACE ORB (TAO) [33], which is an open-source, real-time, object request broker compliant with the OMG CORBA specification [10]. TAO is built on top of the Adaptive Communication Environment (ACE) [73], which is an object-oriented framework implemented in C++ that provides abstractions that allow an application to interface with operating system, networking, and interprocess communication facilities and also provides high level design patterns [18] that encapsulate common communication mechanisms. We chose TAO because it is open-source and enables portability to diverse target platforms. The AQuA project leverages the capabilities of this standard CORBA-compliant ORB to build a middleware that provides dependability to distributed objects using replication. Previous work in AQuA has addressed the problem of tolerating crash failures and value faults. Clients express their dependability requirements by specifying the number of crash failures and value faults they want to tolerate. The middleware then replicates the server objects to provide fault tolerance with a degree of replication that depends on the dependability requirements of the clients. The research described in this thesis has enhanced the AQuA infrastructure to tolerate timing faults by providing timeliness in addition to fault tolerance.

2.2 Group Communication Layer

The members of a replicated service in AQuA are organized into groups. The replication protocols in AQuA depend on the existence of a group communication layer that allows the replicas to communicate using both point-to-point and broadcast protocols. The AQuA replication protocols also require that a membership service keep track of the changes in the membership of a group and notify the group members when changes occur. Further, each replication protocol also requires certain messaging guarantees from the group communication layer, in order to provide fault tolerance to the end applications. We now discuss these guarantees.

2.2.1 Messaging Guarantees

The active and passive replication protocols in AQuA depend on the underlying group communication layer to reliably broadcast all the messages to the group members. This requires that the group communication layer be responsible for retransmitting messages that are lost due to congestion on the links and overflow of buffers in the protocol stack. Reliable broadcast ensures that all the non-faulty members deliver the same set of messages. It ensures that the set will include all the messages sent by non-faulty members and exclude any spurious messages. In addition to reliable broadcast, the active and passive replication schemes in AQuA also require stronger message-ordering guarantees, specifically, *total ordering* and *virtual synchrony*.

The active and passive replication schemes in AQuA assume that consistency among the replicas is determined by the set of messages they process and the order in which they process them. Hence, in order to ensure replica consistency, the active and passive replication schemes in AQuA make use of total-ordered broadcast, which ensures that all the non-faulty replicas that belong to a group receive all the messages sent to the group in the same order. More formally, total order guarantees that if two replicas, *A* and *B*, deliver messages *M1* and *M2*, then *A* delivers *M1* before *M2* if and only if *B* delivers *M1* before *M2* [56]. The messages *M1* and *M2* may be sent by different members of the same group.

Since group membership in the AQuA groups is dynamic, replicas may join or leave a group at any time. In addition to imposing total order among the regular messages broadcast to a group, the active and passive replication schemes in AQuA require that all the group members receive event notifications pertaining to events such as changes in the group membership in the same order. Virtual synchrony guarantees that membership change notifications within a group are observed in the same order by all the group members that remain connected. Moreover, membership change messages are totally ordered with respect to all regular messages that are broadcast to the group. Thus, any pair of replicas that observe the same two consecutive membership changes (or views) will also receive the same set of regular multicast messages between the two membership changes. However, the order of the regular messages is determined by the ordering guarantee provided by the replication scheme. For the active and passive replication schemes in AQuA, the regular messages are totally ordered.

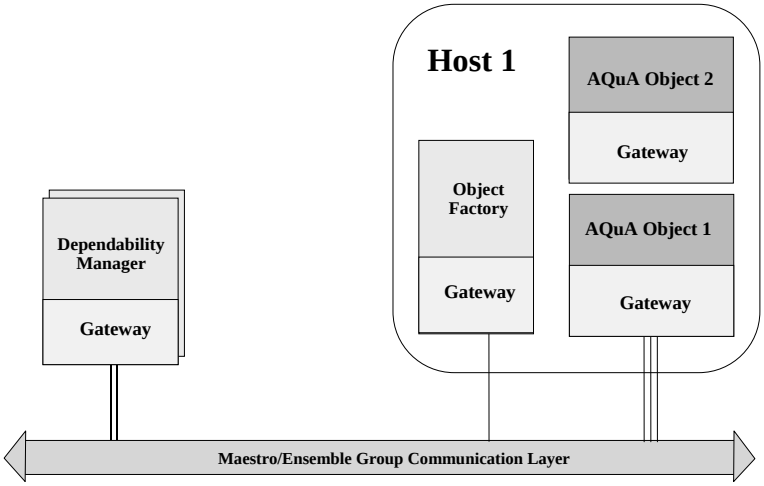


Figure 2.1: AQuA architectural components

2.2.2 Group Communication using Maestro/Ensemble

The Ensemble group communication system [28] provides many of the services and guarantees required by the replication protocols in AQuA. AQuA interacts with Ensemble using the Maestro object-oriented interface [80]. Ensemble implements a layered protocol stack. The layers are assembled at runtime according to the properties specified by the application. Applications are allowed to renegotiate these properties at runtime. As mentioned earlier, the properties specified by the active and passive replication protocols in AQuA include reliable, totally ordered message delivery along with guarantees of virtual synchrony. AQuA uses the *Gossip* name server, a component of Ensemble, to provide the membership service. AQuA leverages Ensemble's heartbeat mechanism to detect crash failures and notify view changes to the members of a group. The leader of a replication group in AQuA is the same as the leader of the corresponding Ensemble group. This allows AQuA to leverage the leader election algorithm of Ensemble to elect a new leader when the old leader fails. Finally, the replication protocols in AQuA make use of the state transfer mechanisms provided by Maestro to transfer state between two members of a group.

2.3 AQuA Architectural Components

We now describe the main components of the AQuA architecture, as shown in Fig. 2.1: a *gateway* that transparently handles all communication between two AQuA applications through the Maestro-Ensemble group communication layer, according to the replication protocol chosen by the applications; a *dependability manager* that manages the replication level of the servers based on the dependability requirements of the clients; and an *object factory* that monitors the load on the hosts and aids the dependability manager in choosing the machines to host the replicated servers. An AQuA application may be a CORBA client that requires fault tolerance, or a CORBA server that is replicated by AQuA to provide fault tolerance. It may also be a fault-tolerant component of AQuA, such as the dependability manager or the object factory.

2.3.1 Gateway

The gateway is one of the key components of the AQuA middleware. It serves as an interface between an AQuA application and the Maestro/Ensemble group communication layer, as shown in Fig. 2.2. AQuA provides fault tolerance by transparently replicating objects using different replication schemes, such as active and passive replication. Replicas that offer the same service are organized into a group. Communication between members of a group takes place through the Maestro-Ensemble protocol stack [80, 28], above which AQuA is layered. The use of group communication in AQuA is transparent to the end applications. Thus, each of the clients, all of which are CORBA objects, is given the perception that it is communicating with a single server object using CORBA's IIOP [10], although in reality, a client's request may be processed by a group of server replicas. This transparency is achieved using an AQuA gateway, which transparently intercepts a local application's CORBA message and forwards it to the destination replica group through Maestro-Ensemble, as shown in Figure 2.2. For the sake of clarity, in the figure we have illustrated a server replica group that has only a single member. In an actual scenario, the group may have multiple replica members.

The different replication schemes supported by AQuA are implemented as *protocol handlers* within

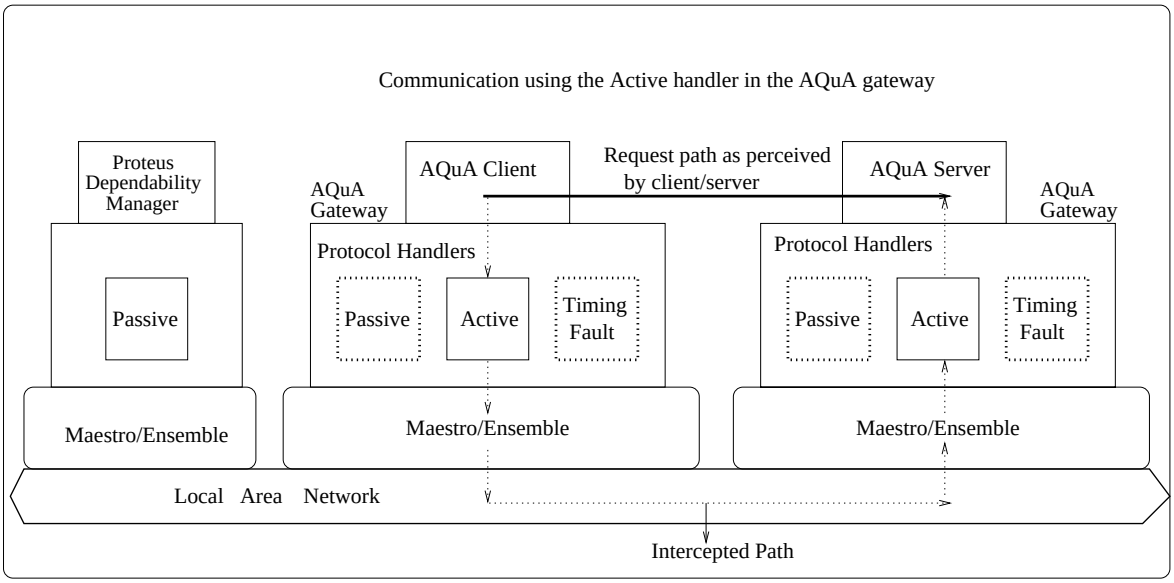


Figure 2.2: Communication using AQuA gateway handlers

the gateway. An AQuA client uses different gateway handlers to communicate with different server groups. For example, Figure 2.2 shows the case in which the AQuA gateway supports three types of handlers: Passive, Active, and Timing Fault. The client in Figure 2.2 has loaded the Active handler to communicate with the server group. The ACE *service configurator* [33] loads the gateway handlers as dynamically linked libraries at runtime by reading a service configuration file. This dynamic loading allows an application to take advantage of newly supported handlers by simply adding them in the configuration file, and thereby obviates the need to recompile applications when new handlers are added or old handlers are modified. This can prove to be a significant advantage to an application developer, especially for large applications.

The functions performed by the gateway handlers may be classified as *generic* or *specific*. The generic functionality is the set of tasks that is performed by all the gateway handlers and comprises the tasks that are responsible for handling the communication between AQuA application objects. The tasks include transparent interception of the CORBA requests or replies sent by the end applications, marshalling of these CORBA messages into Maestro messages, transmission of the messages to their respective destination groups using the Maestro-Ensemble communication protocol, demarshalling of each Maestro message into a CORBA message upon reception, and delivery of the demarshalled

CORBA messages to the end applications. Each gateway handler is also responsible for tolerating a specific type of fault. Previous work in AQuA has developed Active and Passive handlers for tolerating crash failures [69, 67] and value faults [64]. The research described in this thesis has resulted in the development of different handlers for tolerating timing faults.

Each type of gateway handler is associated with its own *handler factory*. The handler factory is responsible for constructing handlers of that type, initializing them, and registering them in the *Gateway Naming Service* at startup time. For example, an active handler factory is responsible for creating an active handler. The configuration used for initializing a handler is specified in the form of directives in the ACE service configurator file [33]. The ACE service configurator reads the file and processes the directives, which direct the configurator to dynamically load the different handler factories as service objects. Each handler factory that is loaded then creates the corresponding handlers using the configuration specified in the command line directive. The handler, which is implemented as a servant of the CORBA dynamic skeleton interface (DSI) [10], is then registered by the handler factory in the Naming Service associated with each AQuA gateway. The Naming Service provides a name-to-object binding for all the gateway handlers.

2.3.2 Dependability Manager

The dependability manager [11, 69], enhances the capabilities of CORBA objects to provide fault tolerance for distributed applications. The dependability manager manages the replication levels of different services based on the dependability requirements of the users of those services and the replication scheme selected by the users. The users may specify their dependability requirements by choosing the types of faults they want to tolerate (such as crash failures or value faults), and the number of faults of each type they want to tolerate. To avoid a single point of failure, the dependability manager is itself replicated.

When a replica crashes, the dependability manager creates a new replica to replace the failed one, if the existing set of replicas is not sufficient to satisfy the dependability requirements of the users. The manager creates the new replica on the machine with the least load, as determined by the periodic load

updates received from the object factories. If the manager is unable to find sufficient resources to meet the dependability requirements of a user, it communicates this to the user using callback functions.

2.3.3 Object Factory

An object factory component runs on every host in the system. It acts as a load sensor and is responsible for periodically sending updates about the host load to the dependability manager. Further, it is also responsible for creating and terminating replicas on the host on which it is running, when instructed to do so by the dependability manager.

2.4 Replication Protocols in AQuA

We now briefly compare the active and passive replication protocols implemented in AQuA. Active replication [63] tries to maintain a consistent state among all the replicas at all times. It achieves this by forwarding a client's request to all the replicas offering the service. Although all the replicas process the request, the client receives only a single reply. AQuA uses three different strategies for selecting the reply to be delivered to the client. In the *Leader-Only* scheme, only the leader of the server group sends a reply to the client. In the *Pass-First* scheme, all the replicas send their replies. However, only the earliest reply received by the client gateway is delivered to the client. Both of these schemes tolerate crash failures. In the *Majority Voting* scheme, which is used mainly for tolerating value faults, all the replicas send their replies. The client gateway votes on the replies and delivers to the client the reply that concurs with a majority of the replies received. Since all of the replicas process the requests of each client, active replication uses more computational resources under fault-free conditions than its passive counterpart. Further, active replication requires that all the replicas be connected all the time.

Passive replication [67] (also called the primary-backup approach) is another strategy for tolerating crash failures. The passive replication strategy also forwards a client's request to all the replicas. However, the request is serviced only by the primary replica, also called the *leader*. The backup replicas merely buffer the request in their local message buffers. Upon servicing the request, the leader first

multicasts its reply to the backup replicas in its group and then sends the reply to the client. The backup replicas store the reply in their local message buffers. The request and reply messages stored by the backup replicas in their local message buffers are used during recovery from the failure of the leader replica, as will be explained shortly.

Since the backup replicas do not directly service the requests, their internal states may diverge from that of the leader over a period of time. Therefore, to reduce the time needed to recover from a leader failure, the leader periodically advertises its state either by multicasting it to the backup replicas or by checkpointing it to a stable storage server. When the backup replicas receive the state update from the leader, they remove the messages that are part of the recently transferred state from their message buffers. This removal has two advantages. First, it minimizes the space required to maintain the different message buffers. Second, it reduces the number of residual messages that the newly elected leader has to replay during recovery, thereby reducing the recovery time.

2.4.1 Failure Recovery Time

In active replication, all the replicas process all the messages in the same order in order to maintain consistent state. As a result, upon the failure of the leader, any of the existing replicas can assume immediate leadership. Therefore, active replication incurs negligible recovery overhead following the failure of a leader replica, at the expense of using more computational resources under fault-free conditions.

Recovery from the failure of a leader in the case of passive replication involves two main steps. First, the newly elected leader has to set its state to the last transferred state of the previous leader. The new leader then has to replay the messages that were stored in its local message buffers since the last state transfer, in order to reach the latest state of the previous leader. The time to recover from the failure of a leader depends on two factors:

- the manner in which the state is transferred, and
- the frequency with which the state is transferred.

If the failed leader checkpointed its state to the stable storage server before crashing, the newly elected leader must set its state to the last transferred state by fetching it from the server. Thus, the time to recover using the checkpointing method includes both the disk latency involved in reading the state from the stable storage server, and the network latency involved in propagating the state across the network. If, however, the previous leader had multicast its state to the backup replicas before crashing, the state of the newly elected leader would already be set to the last transferred state. As a result, the recovery time would not include the overheads involved in retrieving the previous leader's state.

Next, the new leader has to replay the messages in its buffers by delivering the request messages in its local buffer to the application, and transmitting the reply messages in its local buffer to the client. The number of messages to be replayed depends on how frequently the previous leader has transferred its state before crashing. As mentioned earlier, with each state transfer, the messages in the buffers that constitute the recently transferred state are removed, because they are no longer required during recovery. Thus, the more frequent the state transfers, the fewer the messages that remain in the buffers at the time of failure, and hence the shorter the recovery time. However, the periodic state transfer messages compete with the application's messages for the network bandwidth, thereby reducing the fraction of bandwidth available for the application's messages. Hence, the more frequent the state transfers, the higher the communication overhead. Therefore, it is important to choose the state transfer protocol based on an understanding of the tradeoffs between the communication overhead incurred and the time it takes to recover from a leader failure.

Summary

In this chapter, we have provided an overview of the main components of the AQuA middleware and the replication protocols supported by AQuA to provide fault tolerance to distributed objects. The main goal of the previous work in AQuA was to address the dependability requirements of clients. AQuA can be extended to support protocols tailored to different application semantics by implementing the protocols as gateway handlers. In the following chapters, we describe how we have enhanced AQuA to

make it a QoS-aware middleware by building protocols that allow access to replicated objects based on the QoS requirements of the clients. We specifically focus on QoS along the timeliness and consistency dimensions.

Chapter 3

Probabilistic Temporal Guarantees

As far as the laws of mathematics refer to reality, they are not certain;

and as far as they are certain, they do not refer to reality.

- Albert Einstein, World Of Mathematics

Replication is a well-known approach used by many dependable middleware to provide fault tolerance and scalability. In Chapter 2, we described how AQuA, which is a dependable middleware, used replication to provide fault-tolerant distributed services. In this thesis, we address the issue of using replication to support time-sensitive applications. An important problem in the execution of time-sensitive applications in a replicated environment is that of preventing timing failures by dynamically selecting the replicas that can satisfy a client's timing requirement, even when the quality of service is degraded due to replica failures and transient overload on the server. In this chapter, we describe the framework we have developed to address this problem in the case when the replicated state is static. The framework makes use of a probabilistic approach to provide timeliness guarantees to applications. The approach we use estimates a replica's response time distribution based on performance measurements regularly broadcast by the replica. An online model uses these measurements to predict the probability with which a replica can prevent a timing failure for a client. A selection algorithm then uses this prediction to choose a subset of replicas that can together meet the client's timing constraints with at least the probability requested by the client.

The rest of this chapter is organized as follows. In Section 3.1, we introduce our research problem and provide the motivation for our work. Section 3.2 describes our assumptions and our system model. In Section 3.3, we describe the QoS guarantees requested by the clients. In Section 3.4, we describe the

parameters of the probabilistic model we use for the replica selection. In Section 3.5, we present the dynamic replica selection algorithm that we have developed to provide probabilistic timeliness guarantees, and follow this in Section 3.6 with details of our implementation of the framework in AQuA. In Section 3.7, we present experimental results based on our implementation. Finally, in Section 3.8, we compare our work with some related efforts.

3.1 Motivation

Server replication is a popular approach for building fault-tolerant distributed services (e.g., [4, 25, 47, 11, 55, 15]). Replication is also a commonly used solution for improving the scalability of a distributed service, i.e., to ensure that the response time of a service does not significantly degrade with an increase in the number of clients accessing the service (e.g., [38, 50, 58]). Providing good response times in the presence of replica faults, however, is a challenging goal, especially when the number of available replicas is constrained. We can achieve good fault tolerance by allocating all the available replicas to service a single client, but such an approach increases the load on all the replicas and results in poor response times for the remaining clients. On the other hand, assigning a single replica to service each client allows multiple clients to be serviced in parallel, but it may result in an unacceptable delay for the client being serviced, if the replica assigned to it fails while servicing the request. Hence, neither approach is suitable when a client has specific timing constraints and failure to meet the constraints results in a penalty for the client. Thus, in order to build a dependable service, we need a method that attempts to prevent the occurrence of such timing failures for a client by selecting replicas from the available replica pool based on an understanding of the client's timing requirements and the responsiveness of the replicas. The research described in this chapter presents the approach we have used to realize this goal.

Several other replica selection algorithms have been formulated with the objective of choosing the replica that can deliver the lowest possible response time. Those algorithms often target clients of stateless, distributed services, such as the World Wide Web, in which the servers do not maintain any

records of ongoing client transactions. Some of the algorithms choose the nearest replica based on a distance metric [29], and some choose the replica with the best historical average response time [72]. Some use regression analysis of previously collected data to predict the time to propagate a message to different replicas [9], and pick the replica that has the lowest future propagation time. Finally, some of them actively monitor both replica load and network delays, use them to estimate the response times of the replicas, and select the replica that has the smallest estimated response time [14]. All of these efforts assign a single replica to each client and do not consider the case in which a replica may fail while servicing a request. For that reason, it is the responsibility of the client to retransmit its request upon failure to receive a response. Such strategies based on temporal redundancy may be used when the timing requirements are large enough to allow retransmissions. However, they may not be suitable for clients with stricter time constraints and in cases when failure to meet those constraints may result in penalties for the clients.

In contrast, our work targets clients that have specific response time requirements and require that they be met with certain probabilities. Like the above efforts, our work targets applications, such as search engines and radar-tracking applications, in which clients retrieve information but do not modify the state of the servers. However, the server replicas in our case are distributed across a local area network. The approach we use first estimates a replica's response time distribution based on performance measurements regularly published by the replica. An online model uses these measurements to estimate the probability with which the replica can prevent a timing failure for a client. A selection algorithm uses this estimate to choose a subset of available replicas that can together meet a client's timing constraints with at least the probability requested by the client. Each replica in the selected set independently processes the request and sends its response. However, only the earliest reply is delivered to the client. The selected subset is chosen in such a way that the client's probabilistic timing requirement can be met even when one of the members in the selected subset crashes before responding to the request. We have implemented our algorithm in AQuA and analyzed its performance experimentally.

3.2 System Model and Assumptions

We now describe the system for which we want to solve the dynamic replica selection problem. The machines hosting replicated services in this system are distributed across a LAN. A machine may host multiple replicas. The services in the system are frequently accessed by several clients concurrently. Clients requesting the use of these services demand specific response time guarantees that have to be met with a certain probability. Failure to receive a response for a request within the specified time results in a *timing failure* for the client. We assume that the load on a replica may fluctuate and that periods of high load may make it less responsive. We also assume that while the links in the LAN connecting the system do not experience frequent fluctuations in traffic, they may experience occasional periods of high traffic, which may result in large delays in the message delivery time. Finally, a replica may crash, making it unresponsive. Any of these factors may contribute to a timing fault, resulting in missed deadlines for the client.

3.3 Problem Description

Given the above sources of timing faults, the problem we address is that of selecting replicas to meet the timing constraints of the clients in such a way that the timing failures experienced by a client will be within the threshold acceptable to it. Since we target applications with static state in this chapter, we do not address the issue of maintaining replica consistency. Therefore, we assume that there is no restriction on the order in which the replicas service the requests. We now state how a client expresses its timeliness requirements and then outline the decisions that have to be made when allocating the replicas.

A client that requires that a service respond to its request within a specific time expresses its requirements as a QoS specification. The client may either specify its QoS requirement at start-up time, or negotiate it at runtime as often as it wants. For each service, the client specifies the time by which it wants to receive a response after it transmits its request to that service, and the minimum probability with which it wants the time constraint to be met. If a response does not meet the specified time

constraint then it results in a timing failure for the client. If the frequency of timing failures is so high that the middleware is unable to deliver timely responses with at least the minimum probability that the client has specified, then the client is notified by the middleware through a callback.

Our research objective is to reduce the occurrence of timing failures under normal conditions as well as when the responsiveness of a service is reduced, due to either the failure of its replicas or transient overloads. We achieve this objective by designing a distributed request scheduler that transparently intercepts a client's request, estimates the response time of the different replicas offering the service that the client has requested, and selects a subset of available replicas that can meet the client's response time requirements with a probability at least as high as that requested by the client. Our approach associates each client with a local scheduling agent that makes the replica selection decisions on the client's behalf. The scheduler uses historical performance data collected at runtime as input to solve a probabilistic model, which estimates the probability that a response will be received on time. The scheduler then forwards the request to the selected replicas. Each of the selected replicas independently services the request and sends back its response. However, only the earliest response is delivered to the client.

In a system in which a replica's responsiveness may change unpredictably due to either transient overloads or crashes, like the system we have described in Section 3.2, it is impossible for the scheduler to predict with certainty whether any single replica can meet a client's timing constraint. In order to satisfy our goals of servicing multiple clients while at the same time providing a reasonable level of fault tolerance, an important decision our scheduler has to make is that of choosing the redundancy level with which a request has to be serviced. The scheduler we have designed makes its decisions adaptively based on the probability with which the individual replicas will meet a client's timing constraint. The higher the probability that the chosen replicas will meet the constraints, the lower the redundancy level.

3.4 Dynamic Replica Selection in AQuA

We now describe the dynamic replica selection algorithm that we have developed to address the timing failure problem in AQuA. First, we discuss the performance parameters we use to guide the replica selection. We then discuss the design of the information repository that stores the measured values of these parameters. Next, we describe our selection algorithm, which uses these experimentally measured parameters to build a model that guides the replica selection. We follow that with a description of the design and implementation of the timing fault handler that tries to meet a client's timing requirements using the selection algorithm.

3.4.1 Factors Influencing the Response Time

Figure 3.1 shows the stages along the path traversed by a typical request from an AQuA client that has specific timing constraints. In Stage1, the AQuA client invokes a remote method using CORBA's IIOP [10]. The request is then intercepted by a protocol handler in the AQuA gateway. The handler marshals the request into a Maestro message, and in Stage2 presents it to the Maestro/Ensemble protocol stack, from where it is transmitted across the network to the server gateway. This gateway-to-gateway communication may use point-to-point or multicast communication depending on the number of replicas to which the client request is forwarded. In Stage3, the protocol handler in the server gateway receives the Maestro message, demarshals it into a CORBA message, and enqueues it in the request queue. A gateway thread services the queue in FIFO order and in Stage4, delivers the requests to the server application using a CORBA upcall [10]. Upon servicing the request, the server sends its response to the client. The protocol handler in the server gateway transparently intercepts this response, and forwards it to the client gateway along the Maestro/Ensemble protocol stack. The client gateway delivers the earliest response it receives for a request by making a CORBA upcall to the AQuA client.

We conducted experiments to determine the factors that have a significant impact on a replica's response time in AQuA. Based on our off-line analysis, we concluded that the response time in AQuA is mainly affected by the following factors:

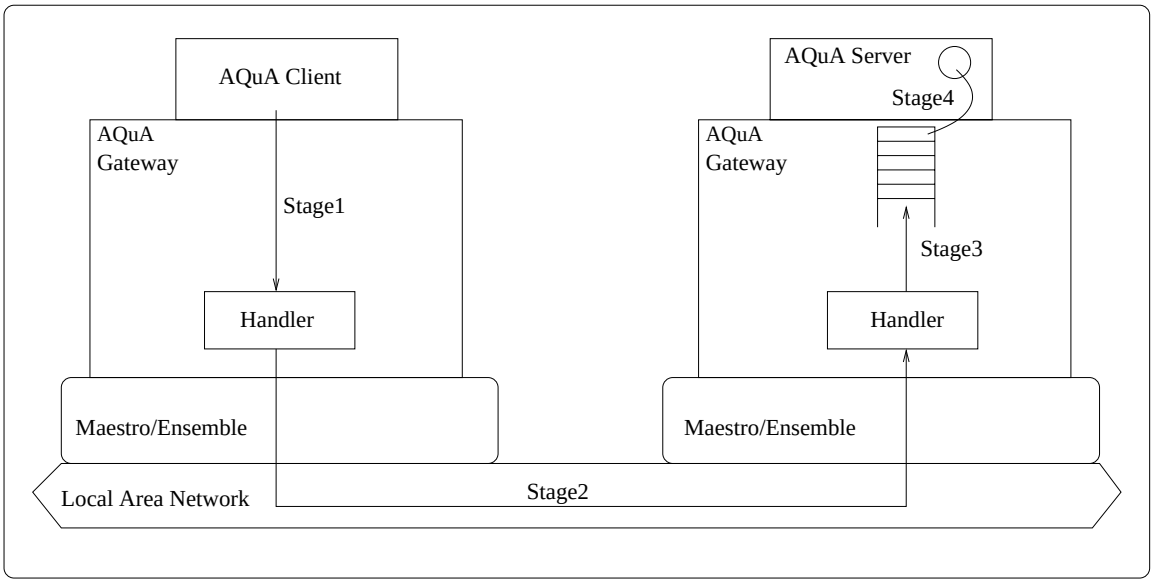


Figure 3.1: Stages along the path traversed by a request in AQuA

Gateway-to-Gateway Delay: the time it takes an AQuA request or response embedded within a Maestro message to travel between two AQuA gateways, as in Stage2 of Figure 3.1. From the figure, we see that this delay includes the time it takes a message to travel through the Ensemble/Maestro protocol stack and the time it takes the message to travel on the wire across the LAN. This delay is incurred on both the request and response paths, and the two delays together make up the two-way gateway-to-gateway delay. For a message of a given size, this delay varies mainly with the load on the network and the number of group members involved in the communication.

Queuing Delay: the time that a request spends waiting in the request queue of the server. This time varies with the rate at which the requests are serviced. It also varies with the number of previously outstanding requests in the queue, because the server uses FIFO scheduling to service its request queue.

Service Time: the time it takes a server to process the request after dequeuing it from the request queue. Differences among the service times of requests that are of the same kind arise mostly due to the load on the host.

In addition to the above sources of delay, a response from a replica may suffer an unacceptable delay if

the replica crashes before responding.

3.4.2 Gateway Information Repository

For each replica, we regularly monitor the above performance parameters at runtime, and maintain the recently measured values in a distributed information repository. An online probabilistic model then uses these measurements to estimate a replica's response time during replica selection. Since the information changes with time, the smaller the time to update the information repository, the more current and accurate the information provided by the repository. That in turn, facilitates better selection decisions. Further, since the information lookup is done by the scheduler for each request, it is important that the lookup time be as small as possible.

As mentioned in Section 2.3.1, an AQuA client uses different gateway handlers to communicate with different servers. In other words, the gateway handler identifies the server group with which a client is communicating. Thus, a client that is communicating with multiple servers would have multiple handlers loaded in its gateway. The handlers associated with a client gateway are responsible for intercepting a request, selecting the replicas to service the request, and transmitting the request to the selected replicas. To enable the selection, we associate an information service with each timing fault handler within a client's gateway. Although this distributed design has the drawback that the replica-related information is redundantly stored at multiple client gateways, it has several advantages compared to a global information service, which would avoid this drawback. First, having a repository local to each client handler avoids the possibility of a single point of failure. Second, it avoids the overhead that would result from making calls to a remote information service; that is important, as we want to minimize the selection overheads. Third, allowing each gateway handler to access its local repository avoids the need to enforce concurrency control, which would otherwise result in high access overheads. Finally, since a repository local to a handler only caches information relevant to the service associated with that handler, the search space is smaller, so it takes less time to access information.

The gateway information repository within each client handler stores the list of replicas that offer the service associated with the handler. For each of these replicas, it stores the current number of

outstanding requests in the replica's request queue and the most recently measured two-way gateway-to-gateway delay between the client and the replica. In addition, the repository also stores a *service time vector* and a *queuing delay vector* for each replica. The former records the service time while the latter records the queuing delay for the most recent l requests serviced by the replica. Thus, l can be considered to be the size of a sliding window of requests, and its value is chosen so that it includes a reasonable number of recent requests but eliminates obsolete measurements. The next section explains how these performance parameters are used in the selection of the replicas.

3.5 Probabilistic Replica Selection Algorithm

Using the performance measurements collected above as inputs, the local scheduler that is part of a timing fault handler builds a model to predict the probability that a subset of replicas will be able to meet a client's timing requirements. The replica selection is then done based on this resultant probability. We first define the notation we use to present the probabilistic model:

- $M = \{m_1, m_2, ..., m_n\}$ is the set of replicas offering the service requested by a client. The scheduler obtains this set from its local information repository.
- $R = \{R_1, R_2, ..., R_n\}$, R_i is the random variable denoting the time to receive a response from a replica $m_i \in M$, after a request was transmitted to it.
- $P_c(d)$ is the probability with which the client wants a response for its request by time d, as explained in Section 3.3.

We now need to determine the probability that a response from a subset $K \subseteq M$, consisting of $k > 0$ replicas, will arrive by the client's deadline, d, and thereby avoid the occurrence of a timing failure. This probability is denoted by $P_K(d)$. As stated earlier in Section 3.3, each replica in the subset independently processes the client's request and sends back its response. However, only the first response received for a request is delivered to the client. Therefore, a timing failure occurs only if no response

was received from any of the replicas in the set K within d time units after the request was sent. Computing the distribution of the time until a response is received is straightforward if we assume that the response times of individual replicas are independent of one another. While this is not strictly the case in a shared network in which the network delays may be correlated, we believe it is a reasonable assumption to make, since the network delay is usually a small fraction of the replica's response time in a LAN environment. Although we could have used a more detailed model that accounts for all the correlations more accurately, we chose a simpler model, because it is our opinion that the overhead involved in solving the more detailed models at runtime may not justify their use when time-sensitive applications are being used. We use the independence assumption to compute the probability, $P_K(d)$, for the replicas in subset K, as follows:

$$P_K(d) = 1 - P(\text{no replica in } K \text{ responds before d})$$

$$P_K(d) = 1 - \prod_{m_i \in K} P(R_i > d)$$

$$P_K(d) = 1 - \prod_{m_i \in K} (1 - F_{R_i}(d)) \tag{3.1}$$

where $F_{R_i}(d)$ is the response time distribution function for replica m_i.

3.5.1 Computing the Response Time Distribution

Given the above model, we now explain how we compute the value of $F_{R_i}(d)$ for a replica m_i. Henceforth, we will use the subscript i to refer to the replica m_i. We make use of the parameters explained in Section 3.4.1 to define the response time random variable, R_i, using Equation 3.2 below.

$$R_i = S_i + W_i + G_i \tag{3.2}$$

where S_i is the random variable denoting the service time for a request serviced by m_i; W_i is the random variable denoting the queuing delay experienced by a request waiting to be serviced by m_i;

and G_i is the random variable denoting the two-way gateway-to-gateway delay between the client and replica m_i. Our model assumes a server's request queue to be made of requests belonging to the same job class. For each request, we experimentally measure the values of the service time, S_i, as described later in Section 3.6, and record the values of the most recent l requests in the *service time vector* in the information repository. We do the same for the queuing delay, W_i, and record its recent values in the *queuing delay vector* in the information repository. Thus, these vectors represent a sliding window, L, of size l, over which the performance history is recorded. For the gateway-to-gateway delay, G_i, we decided to use its most recently measured value rather than record its history over a period of time. This decision was based on the observation that the traffic in a LAN does not frequently fluctuate as the other two parameters do. We verified that this observation is true for the environment we used. For environments in which this observation is not true, it would be simple to extend our approach to record the value of the gateway-to-gateway delay over a sliding window as we do above for the service time and queuing delay.

Given that we can measure the performance parameters and record them at runtime, we can now compute the value of the distribution function $F_{R_i}(d)$ for a replica m_i. To do this, we first compute the probability mass function (pmf) of S_i and W_i based on the relative frequency of their values recorded in the sliding window, L. We then use the pmf of S_i, the pmf of W_i, and the recently recorded value of G_i to compute the pmf of the response time R_i as a discrete convolution of W_i, S_i, and G_i. The pmf of R_i can then be used to compute the value of the distribution function $F_{R_i}(d)$.

3.5.2 Replica Selection Algorithm

Given the ability to compute the probability that an individual replica will meet a client's time constraint, we were able to develop an algorithm that applies Equation 3.1 to select a set of replicas that can meet this time constraint with the probability the client has requested. Algorithm 1 presents the main ideas of the selection algorithm. The algorithm first sorts the replicas in decreasing order of the probability that they can individually meet the client's response time requirement. In Line 4, the first element of the sorted replica list is included in the selected set, K. The algorithm then considers the

remaining replicas in the list in sorted order, including each replica in the candidate set X, until it has included enough replicas in X to satisfy the condition $P_X(d) \geq P_c(d)$, where $P_X(d)$ can be computed using Equation 3.1. In Line 11, we extend the candidate set X to include the first element, m_0, which was selected in Line 4, in order to form the final selected set of replicas, K. Thus, we include the replica, m_0, that has the highest value of $F_{R_0}(d)$, in the final selected set, although we do not consider it when testing the condition in Line 10. We now explain the reason for this.

Algorithm 1 Replica selection algorithm: static state

Require: $V =< i, F_{R_i}(d) >$ {set of replicas and their corresponding distribution function}
Require: Client Inputs:
d : client's deadline,
$P_c(d)$: probability that the deadline should be met
1: $X \Leftarrow \phi$
2: $prod \Leftarrow 1$
3: sortedList $\Leftarrow$ sort V in decreasing order of $F_{R_i}(d)$
4: $K \Leftarrow [first(sortedList)]$ {always include the replica that has the highest probability in the selected list}
5: newSortedList $\Leftarrow$ sortedList - K
6: **for all** i in newSortedList **do**
7: $X \Leftarrow X \cup i$
8: $g_i \Leftarrow 1 - F_{R_i}(d)$
9: $prod \Leftarrow prod * g_i$
10: **if** $1 - prod \geq P_c(d)$ **then**
11: $K = X \cup K$
12: return K {found an acceptable replica set}
13: **end if**
14: **end for**
15: return M {return the set comprising all the replicas}

Since replicas may crash, our goal is to choose a set of replicas that can meet a client's time constraint with the probability the client has requested, even when one of the replicas in the selected set crashes before responding to a request. Since we do not know a priori which one of the selected replicas is likely to crash, our intuition is that if we can choose a set of replicas that can satisfy the timing constraint with the specified probability despite the failure of the member, m_0, that has the highest probability of meeting the client's deadline, then such a set should be able to handle the failure of any other member in the set. The loop in lines 6-14 of Algorithm 1 attempts to find such a subset, X, that

satisfies the condition in line 10 by excluding the member m_0. If it finds such a set, it extends the set to include m_0 to form the final set, K. If, however, it is unable to find such a set, then it returns the complete set of available replicas, M. We now justify our claim that the set K found by Algorithm 1 can handle single replica crashes. Let $g_0 = 1 - F_{R_0}(d)$, where $F_{R_0}(d)$ is the distribution function of the first member in the sorted list. Since $F_{R_0}(d) \geq F_{R_i}(d), \forall i$, we have,

$$g_0 \leq g_i,\ 0 \leq g_i \leq 1,\ \forall i,$$

$$g_0 * (g_1 * \cdots * g_{i-1} * g_{i+1} * \cdots * g_x) \leq \prod_{i=1}^{x} g_i$$

$$1 - (g_0 * g_1 * \cdots * g_{i-1} * g_{i+1} * \cdots * g_x) \geq 1 - \prod_{i=1}^{x} g_i$$

$$1 - (g_0 * g_1 * \cdots * g_{i-1} * g_{i+1} * \cdots * g_x) \geq P_c(d) \tag{3.3}$$

Equation 3.3 follows from the condition in Line 10 of Algorithm 1. This equation shows that should any one of the members, i, belonging to the selected set K crash without completing its transaction, the other members would still be able to meet the client's timing constraint with the probability the client has requested.

We chose to address only single replica crashes in this work because we targeted an environment in which replicas offering the same service ran on different hosts. It is our observation that the chances of two hosts failing simultaneously during a single method invocation is fairly small. For that reason, we assume that the probability of simultaneous failures of two replicas offering the same service is fairly low. If this is not the case, it should be simple to extend the above algorithm to handle multiple failures by following a method similar to the one outlined above.

3.5.3 Algorithm Overhead

In a practical implementation, one must consider the overhead incurred by the selection algorithm. We do that by modifying Algorithm 1 to select those replicas that can respond within $d - \delta$ time units rather

than d time units, where d is the client's deadline as before, and δ is the overhead of the algorithm. As seen from Algorithm 1, the overhead depends mainly on the number of replicas, n, and the size of the sliding window, l, that we use to record the performance measurements broadcast by the replicas. In our implementation, we measure this overhead, δ, each time the selection algorithm is executed, and use the most recently measured value of δ to compute the value of $F_{R_i}(d - \delta)$. We then include this overhead by modifying Algorithm 1 to use the value of $F_{R_i}(d - \delta)$ wherever it formerly used the value of $F_{R_i}(d)$. The implementation of the rest of the algorithm remains unchanged.

3.6 Design of the Timing Fault Handler

Given a QoS specification from a client as described in Section 3.3, we now explain how the timing fault handler tries to meet the client's response time requirements by making use of the above selection algorithm. A client may either negotiate its QoS requirements at runtime or specify them in a configuration file, which is read by the timing fault handler when it is loaded in the client gateway. A sample configuration file is given in Appendix A. The quality of service a client requests from a service is stored in the handler the client uses to communicate with that service. When a client makes a request to that service, the handler uses this QoS specification to select the set of server replicas to process the request.

The timing fault handler uses the Maestro-Ensemble group communication layer to manage communication transparently between a client application and a replicated service. Before they can communicate, the client and all the replicas of an object should join the same *QoS group*. All of the clients that require the same kind of service guarantees (e.g., probabilistic timeliness guarantees in the absence of consistency guarantees) from a replicated object can be placed in the same QoS group. The timing fault handler uses this group to forward requests from a client to a selected subset of server replicas, as will be explained in the next subsection. The client handlers that are interested in receiving performance updates from the servers use the group to multicast their subscription request to the server replicas. Each server replica then keeps track of its subscribers and notifies them whenever it records

a new measurement for its performance parameters. The information, published by the server replicas, is then used to update the client's gateway information repository, as will be explained in further detail in the next subsection. Whenever the membership of a group changes, Maestro-Ensemble detects the change and notifies all the group members about the change. This notification allows the clients that are members of a QoS group to remove the entries for failed replicas from their local information repositories, when they are notified of replica failures. The failed replicas will therefore not be considered in the selection process for subsequent requests.

3.6.1 Request and Response Handling

A typical request is processed by the timing fault handler as shown in Figure 3.2. After transparently intercepting a request from a client, the client's timing fault handler records the interception time, t_0, and hands over the request to its selector module. The selector first retrieves the replica list for the service from its local gateway information repository. If the service has never been accessed before, the information repository would not contain any performance data for the replicas offering that service. In that case, the selection strategy selects all the replicas in the list. This allows the replicas to publish their performance updates to the clients, as described below, and thereby initialize the information repositories. During subsequent requests, the scheduler uses the performance history from its local information repository to choose the replicas based on the client's QoS requirements, using the selection strategy explained in Section 3.5. The handler then multicasts the client's request to the selected replicas using Maestro-Ensemble and records the sequence number of the message and its time of transmission, t_1.

Upon receiving the request, the timing fault handler at the server enqueues the request in the replica's request queue. It then records the time, t_2, at which the request is enqueued. A gateway thread asynchronously processes the request queue in FIFO order. When the request is dequeued for service, the gateway records the dequeue time, t_3, before using CORBA's invocation interface [10] to deliver the request to the server application. When the server sends its response back to the client, the timing fault handler intercepts the response at time t_4 and records the service duration, t_s, where

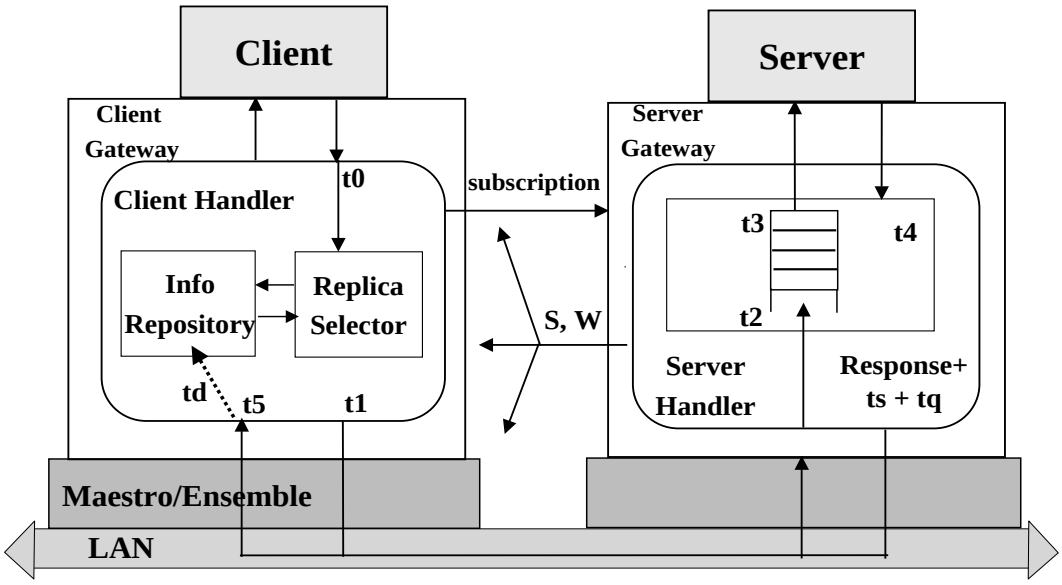


Figure 3.2: Timing fault handler

$t_s = t_4 - t_3$. The server's handler then forwards the reply back to the client gateway along with the performance data, which includes the service duration t_s and the time, t_q, spent by the request in the queue, where $t_q = t_3 - t_2$. The handler publishes this new performance data, along with the replica's current queue length, to all its subscribers. The information is used by the subscribers to update their local information repositories. In our current implementation, the server publishes its performance update to its subscribers each time it processes a request. It should, however, be simple to modify our current implementation so that the updates are published at other intervals, if required.

When the client handler receives a reply from a replica, it records the time of reception, t_5, and extracts the performance data embedded in the message. If the reply is the first one it has received for a request, the handler delivers the reply to the client. The handler then uses the extracted performance data to measure the new round-trip gateway-to-gateway delay, t_d, between the client and replica. This delay, t_d, is given by $t_d = t_5 - t_1 - t_q - t_s$, where t_q and t_s are obtained from the extracted data. The handler then updates the information in its local repository with the new value of the gateway-to-gateway delay.

Since we allow a request to be processed redundantly by multiple replicas, the client gateway may receive multiple responses for the same request. The client gateway does not, however, deliver any of

the redundant replies to the client. Instead, the gateway extracts the performance data contained in each reply message and uses it to update its information repository with the new value of the gateway delay between the client and replica, just as it did for the first reply.

3.6.2 Detecting Timing Failures

We now explain how the timing fault handler detects timing failures and handles them when they occur. The handler maintains a counter that keeps track of the number of times its client has failed to receive a timely response from a service. When the handler receives the first reply for a request sent by its client to a service, it checks whether a timing failure has occurred by computing the response time, $t_r = t_5 - t_0$, where t_5 is the time at which the first reply arrived at the handler and t_0 is the time at which the handler intercepted the request from its client. We measure the response time at the gateway instead of at the end client because the time it takes for the gateway to deliver the response to the end client application is negligible. Besides, measuring the response time at the gateway allows the middleware layer to detect and handle timing failures. A timing failure occurs if $t_r > d$, where d is the response time requested by the client. If the handler detects that a failure has occurred, it updates its counter. If the frequency of timely responses from the service is lower than the minimum probability of timely response the client has requested in its QoS specification, the handler issues a callback to notify the client about its failure to meet the client's QoS specification. The client can then choose either to renegotiate its QoS specification or to issue its requests to the service at a later time. Note that when we collect the timing data as explained above, we do not require that the clocks be synchronized, because we always measure the two end-points of a timing interval on the same machine.

3.7 Experimental Results

We now discuss the experiments we conducted using our implementation of the timing fault handler in AQuA to analyze the performance of the selection algorithm. Our experimental setup was composed of a set of 300 MHz uniprocessor Linux machines distributed over a 100 Mbps LAN. For a minimum-

sized request having negligible service time, the minimum value we achieved for the response time, t_r (defined in Section 3.6.2), was about 3.5 milliseconds. All confidence intervals for the results presented are at a 95% level, and have been computed assuming that the number of timing failures follows a binomial distribution [36].

3.7.1 Overhead of the Probabilistic Selection Algorithm

The overhead of the probabilistic selection algorithm depends mainly on the size of the replica selection pool and the size of the sliding window used to record the performance histories of the replicas. Figure 3.3 shows how the overhead of the model-based selection algorithm varies with the number of available replicas for three different sizes of sliding window: 5, 10, and 20. These overheads include the time to compute the distribution function and the time to select the replica subset and are incurred during each request. Computation of the response time distribution function contributes to 90% of the overheads, while selection of the replica subset using Algorithm 1 contributes to the remaining 10%. The selection overhead for a sliding window of size 5 varies from 180 microseconds to 730 microseconds as the total number of available replicas varies from 2 to 8. The larger the sliding window, the greater the number of data points used to compute the response time distribution, resulting in higher selection overhead. We can control the sensitivity of the selection scheme to changes in the responsiveness of a replica by choosing an appropriate size for the sliding window that stores the replica's performance history. The smaller the window size, the greater the sensitivity to changes in the replica's responsiveness. We used a sliding window of size 5 for our experiments below, unless otherwise stated.

3.7.2 Evaluation of the Probabilistic Model

We also conducted experiments to evaluate how effectively the subset of replicas chosen by the model-based selection algorithm was able to meet a client's deadline with the probability requested by the client. To do this, we used an interactive workload generated by two clients. The clients ran on two different machines and independently issued requests to the same service, with a 1000 millisecond delay between receiving a response and issuing the next request. We use the term *think time* for this

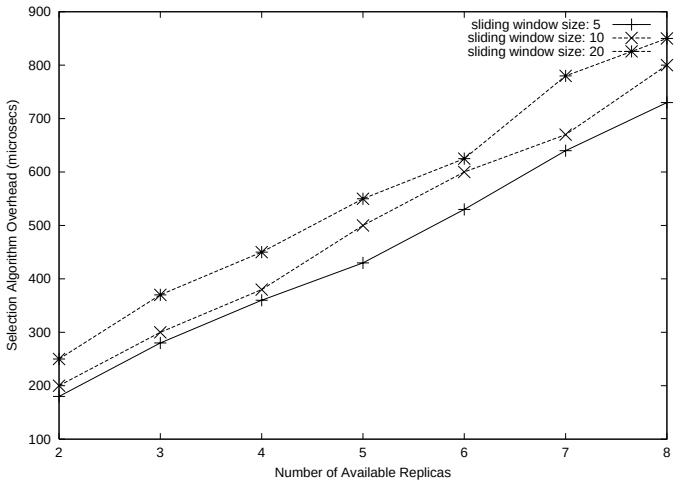
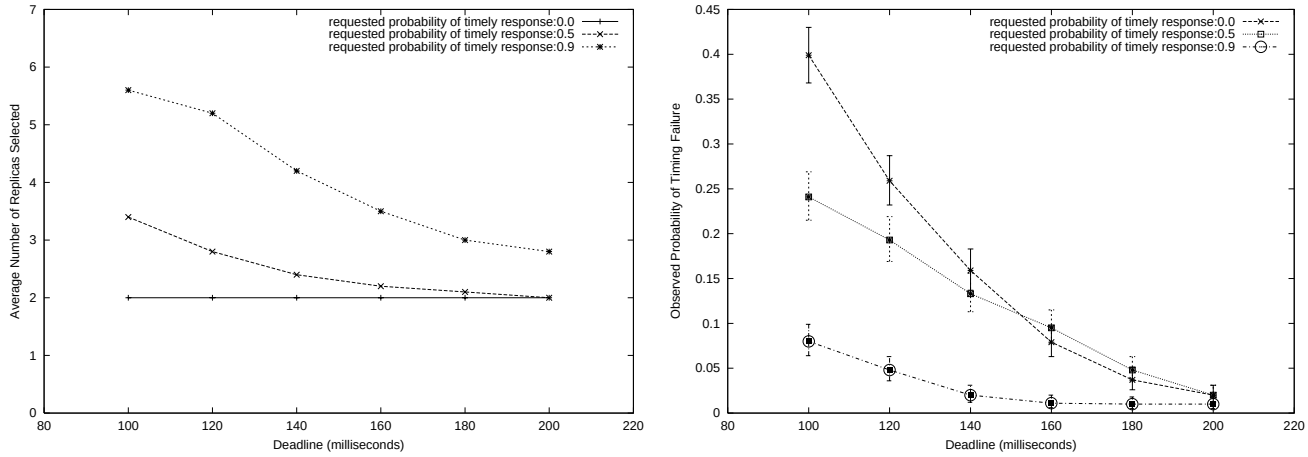


Figure 3.3: Overhead of model-based replica selection algorithm

duration that elapses before a client issues its next request to a service after it receives a reply for its previous request. The number of server replicas available for selection in each experiment was 7. Each server replica ran on a different machine and responded with an integer value. Since the machines in our testbed had insignificant background load on them, we simulated the background load on the servers by having each replica respond to a request after a delay that was normally distributed with a mean of 100 milliseconds and a variance of 50 milliseconds. In every run, each of the two clients issued 1000 requests to the service. One of the clients requested a deadline of 200 milliseconds in each run and specified that this deadline be met with a probability ≥ 0. The second client requested a different deadline in each run. For each of the deadline values of the second client, we computed the probability of timing failures in a run of 1000 requests by measuring the number of responses in the run that had failed to arrive by the deadline specified by the second client. In order to study the behavior of the probabilistic selection algorithm for different values of the probability of timely responses specified by a client, we repeated our experiments for the following three different probability values specified by the second client in its QoS specification:

1. a probability value of 0.9. This allows up to 10% of the client's requests to miss their deadlines, and hence tolerates a maximum timing failure probability of 0.1.

2. a probability value of 0.5. This allows up to 50% of the client's requests to miss their deadlines.



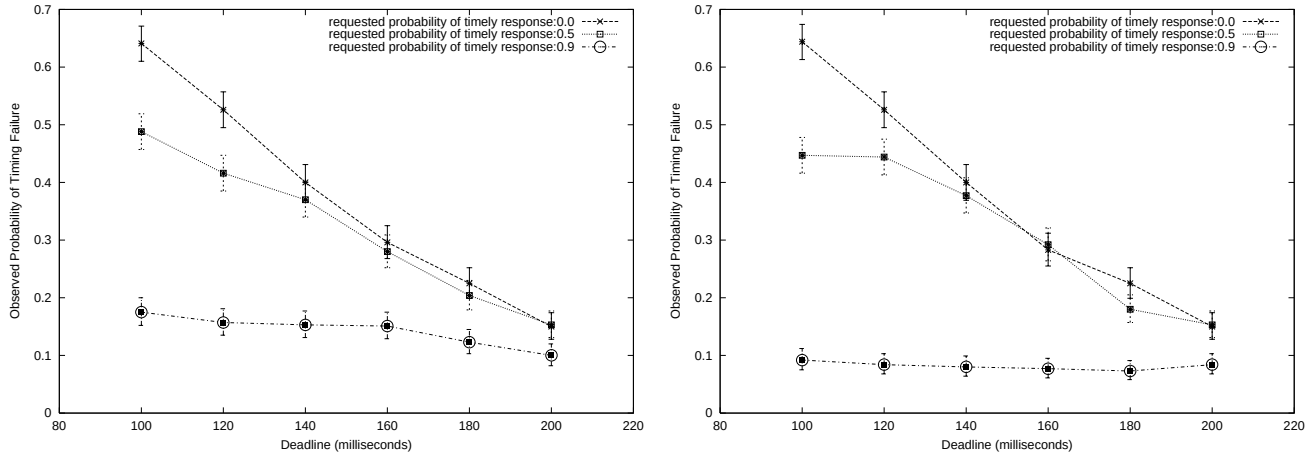
(a) Number of replicas selected

(b) Validation of model

Figure 3.4: Performance of model-based scheme in a crash-free scenario

3. a probability value of 0.0. This does not place any restriction on the number of allowed timing failures. We chose a probability value of 0 because it represents the case in which using the model-based selection algorithm would result in the highest timing failure probability. Hence, it provides a perspective on the worst-case behavior of the model-based selection algorithm.

Figure 3.4a shows the average number of replicas selected by the dynamic selection algorithm to service the second client for each of its QoS specifications. The first observation from this figure is that for higher values of the requested deadline, the algorithm chooses fewer replicas, on the average, to service the client. For example, for the case in which the client requests a probability of timely response of at least 0.9, the selection algorithm assigns nearly 80% of the available replicas to service the client when the requested deadline is 100 milliseconds, while it assigns only 40% of the available replicas when the client requests a less stringent deadline of 200 milliseconds. The second observation from the figure is that the algorithm chooses a lower redundancy level when the client requests a lower probability of timely response. For example, in the first case, in which the client specifies a 0.9 probability of timely response, the algorithm chooses redundancy values as high as 6 to meet some of the client's requests. However, in the case in which the client is willing to tolerate any number of timing failures, the algorithm chooses a redundancy level of only 2, which is the minimum number of



(a) Sliding window size = 5

(b) Sliding window size = 20

Figure 3.5: Performance of model-based scheme during a replica crash

replicas selected by Algorithm 1. These observations can be explained as follows. Algorithm 1 never selects more than the minimum number of replicas necessary to meet a client's QoS requirement. The higher the probability that the chosen replicas will meet a client's deadline, the lower the redundancy level. The less stringent a client's QoS specification, the higher the probability that a chosen replica will meet the client's specification. Therefore, as the client's QoS requirements become more flexible, the algorithm can satisfy them with fewer replicas.

Figure 3.4b shows how successful the selected set of replicas, shown in Figure 3.4a, was in meeting the QoS specifications of the second client. It shows the probability of timing failures observed experimentally, along with their 95% confidence intervals. Figure 3.4b shows that when the client specifies that the probability of timely response must be at least 0.9, the maximum probability of timing failures we observe experimentally is only 0.08, which is lower than the maximum allowed timing failure probability of 0.1. Similarly, for the cases in which the client allows failure probabilities of up to 0.5 and 1, we observe maximum timing failure probabilities of 0.32 and 0.36, respectively, for the deadline values we used. These results show that, in each case, the set of replicas selected by Algorithm 1 was able to meet the client's QoS requirements successfully by maintaining a timing failure probability lower than the failure probability that was acceptable to the client. Thus, for the experimental runs we conducted,

the model we defined in Equation 3.1 was successful in predicting the set of replicas that would be able to meet the client's deadline with at least the minimum probability of timely response requested by the client.

Performance During a Replica Crash

Figures 3.4a and 3.4b show the behavior of the model-based algorithm in the absence of a replica crash. Since we designed our algorithm with the goal of tolerating a single replica crash, we were interested in evaluating the algorithm when one of the selected replicas was unavailable. In Section 3.5.2 we claimed that if we choose a set of replicas that can satisfy the timing constraint of a client with the specified probability even if the member that has the highest probability of meeting the client's deadline crashes, then the set should be able to tolerate any single replica crash that occurs when the request is being serviced. We also provided a simple theoretical justification for this claim. We now describe the experiments we performed to verify whether this claim was valid.

We used the same experimental setup with two clients that we used for the experiments in Figure 3.4 for the crash-free scenario. We simulated the crash of a replica by having the timing fault handler forward a client's request to all the selected replicas except the one with the highest probability of meeting the client's deadline. Since the client never receives a response from the omitted member, this procedure effectively simulates the scenario in which the member may either have crashed before completing its service, or be completely unresponsive. Figure 3.5a shows the results of our experiments. The experiments were conducted using a sliding window of size 5. Based on a comparison of the plots in Figure 3.5a with the corresponding plots in Figure 3.4b for the crash-free case, we see that the crash of the replica that had the highest probability of meeting a client's deadline at the time of the selection results in more timing failures. From Figure 3.5a, we also see that despite the failure of a replica, the model-based algorithm is able to meet the client's QoS specification when the client specifies a value of 0.5 or lower for the probability of timely response. However, when the client requests a higher value, such as 0.9, for the probability of timely response, we see that the timing failure probability measured experimentally is about 0.2, which is higher than the 0.1 failure probability allowed by the client's

specification.

We conducted extensive experiments to investigate why the chosen replicas were unable to meet the QoS specification during a crash for cases in which the client requested a high value for the probability of timely response, such as 0.9. There are two possible explanations. The first hypothesis is that the model may have been unable to find a replica subset to meet the client's QoS requirement in the presence of a single crash fault. In that case, the handler would have sent the request to all the available replicas, as indicated in Line 15 of Algorithm 1. However, our experimental results did not support this hypothesis. We found that in each case the number of replicas selected for each QoS specification in the scenario with a crash was nearly the same as the number chosen in the crash-free scenario, as shown in Figure 3.4a. However, in the scenario with a crash, one less than the number of replicas selected actually processed the request. Thus, Algorithm 1 is indeed able to find a replica subset that is predicted to have the ability to meet a client's timing requirements with a probability ≥ 0.9, despite the crash of a replica. The first hypothesis is therefore ruled out.

The second hypothesis we proposed to explain the observed behavior focused on the experimentally collected performance measurements that our model uses as inputs to predict the probability with which a replica subset can meet a client's deadline. We proposed that these measurements may not have been accurate enough to predict the ability of replicas to provide a high probability of timely response during a crash, specifically values ≥ 0.9. To test this hypothesis, we repeated the experiments we used to obtain the results shown in Figure 3.5a, but used a larger sliding window. Figure 3.5b shows the results we obtained when we used a sliding window of size 20. From that figure we see that the observed probability of timing failures is now at most 0.1 for the case in which the client requests that its deadline be met with a probability ≥ 0.9. Thus, the use of more experimental measurements enabled the probabilistic model to improve the accuracy of its prediction about the set of replicas that can provide a high probability of timely response (≥ 0.9) for a client, despite the failure of a replica.

At this point, one might ask why the model was able to predict the set of replicas that would be able to meet a client's deadline with a probability ≥ 0.9, in the crash-free scenario in Figure 3.4, using a sliding window of size 5. We think the reason is that the inaccuracy in the experimental measurements

in that case was largely offset by the fact that there was one additional replica processing the request. More importantly, the additional replica was the one that had the highest probability of meeting the client's deadline.

We now summarize our conclusions from the above experiments. We have shown that our model can successfully predict the ability of a set of replicas to meet a client's QoS specification. However, the accuracy of the prediction depends on the accuracy of the performance measurements the model uses as its inputs. The accuracy of these measurements is especially important when the client desires a high probability of timely response. One way to improve the accuracy of the prediction, especially for a service that is accessed relatively frequently, is to use a larger number of sample points for computing the response time distribution of the replicas. One can control the number of sample points in our experiments by choosing an appropriate size for the sliding window that records the performance measurements broadcast by the replicas. A large sliding window has the disadvantage that it may allow obsolete performance information to be included when a service is not accessed frequently. On the other hand, a small window size may be more appropriate in environments in which a replica's responsiveness changes rapidly with time, as a smaller window allows quicker adaptivity to changes in a replica's responsiveness. Thus, the size of the sliding window has to be chosen based on the specific environment and access patterns of a service, through an assessment of the tradeoffs between the desired levels of accuracy and adaptivity.

3.7.3 Comparison of Replica Selection Algorithms

We also conducted experiments to compare the performance of the model-based replica selection algorithm with two other selection algorithms that are well-known and commonly used: a static algorithm and a round-robin algorithm. We implemented both of these selection schemes as separate gateway handlers in AQuA, and chose the scheme we wanted to use at the beginning of each experiment. Our goal was to compare the performance of the model-based selection scheme with the performance of the round-robin and static schemes for different client-induced load. We first compare the model-based and static schemes and then follow that with a comparison of the model-based and round-robin schemes.

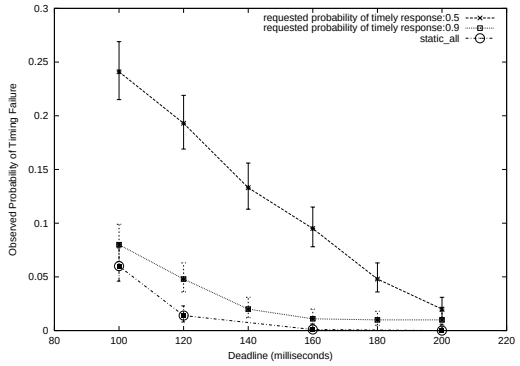
Model-Based Algorithm vs. Static Algorithm

The static scheme allocates all the available replicas to service a client. Hence, we call the static scheme `static_all`. Two sources of overhead that are present in the model-based scheme are absent in the `static_all` scheme. They are the computational overhead incurred by the replica selection algorithm, and the communication overhead incurred by the regular performance updates. Intuitively, it would seem that we may be able to reduce the probability of timing failures for a client by sending its request to as many replicas as possible. On the other hand, by having all the replicas process a request, the `static_all` scheme may result in higher queuing delays for the requests, leading to higher response times. Thus, we were interested in finding out how the two schemes performed under different workload intensities.

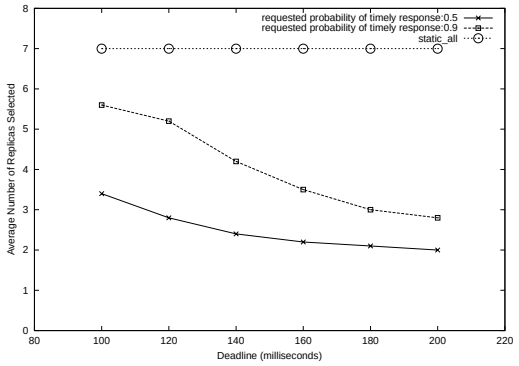
We carried out our experiments by varying the client-induced load in two ways. In the first case, we varied the client-induced load by varying the think time between the requests. In the second case, which we describe later in this section, we varied the client-induced load by varying the number of clients. For the first case, we used the same experimental setup with an interactive workload generated by two clients, that we described in Section 3.7.2 for the crash-free scenario. When we conducted experiments using the model-based scheme, all the clients involved in the experiment chose the model-based selection handler. Similarly, when we analyzed the `static_all` scheme, all the clients involved in the experiment used the static selection handler. Figure 3.6 compares the performance of the model-based scheme with that of the static scheme for different values of the think time. The graphs in the left-hand column of Figure 3.6 compare the observed timing failure probabilities for different values of the think time, while the graphs on the right compare the average number of replicas selected for the corresponding cases.

Varying the Think Time

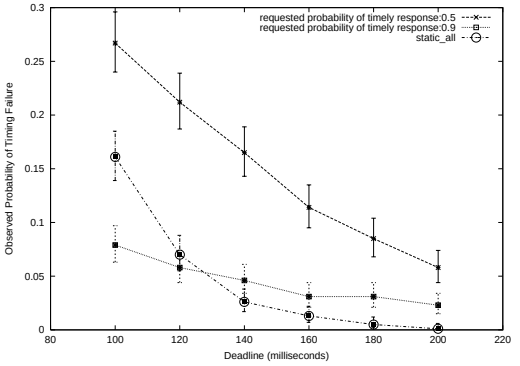
Our first observation based on the graphs in Figure 3.6 is that the timing failures observed using both the schemes increases as the think time is reduced from 1000 milliseconds to 0 milliseconds. The reason is that as the think time reduces from 1000 milliseconds and approaches values closer to the mean



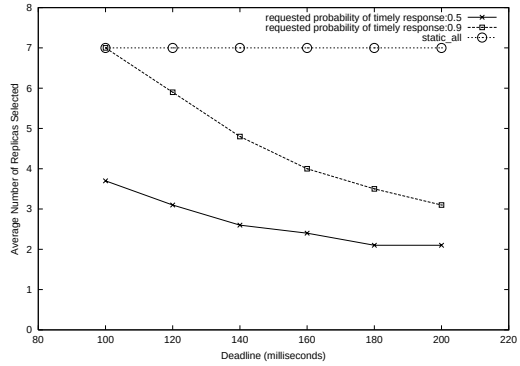
(a) Think time: 1000 ms



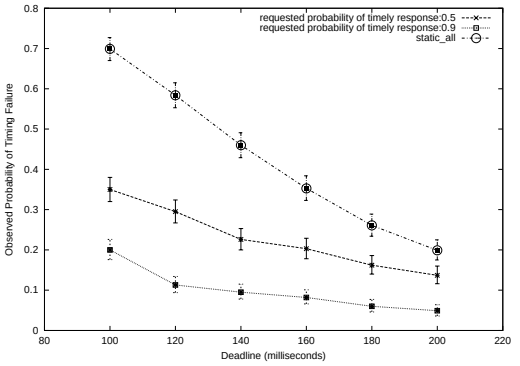
(b) Think time: 1000 ms



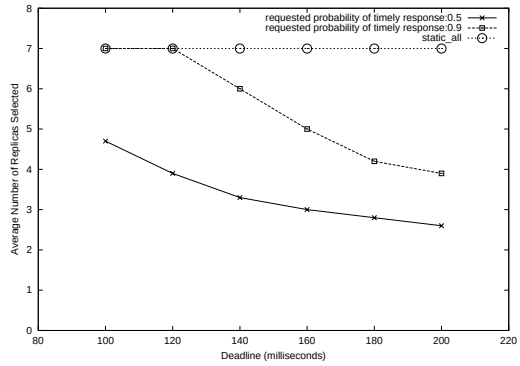
(c) Think time: 250 ms



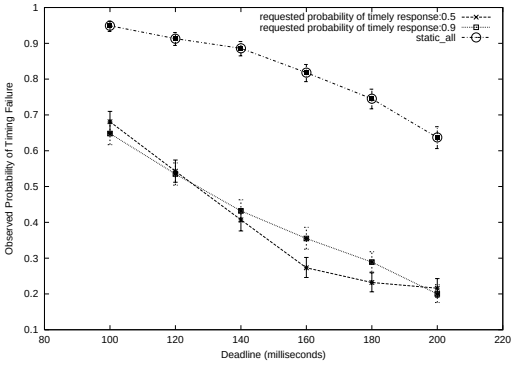
(d) Think time: 250 ms



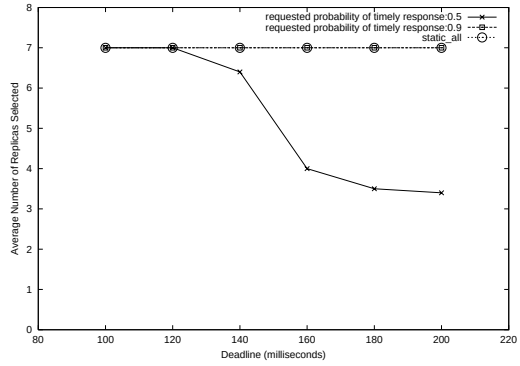
(e) Think time: 100 ms



(f) Think time: 100 ms



(g) Think time: 0 ms



(h) Think time: 0 ms

Figure 3.6: Model-Based scheme vs. static scheme: variable think time

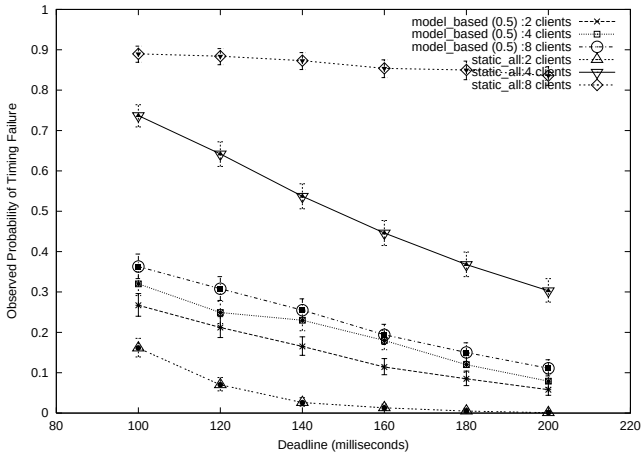
service time of 100 milliseconds and lower, the number of requests that experiences queuing delays at the servers increases. Since we used only two clients and all the method invocations from them were synchronous, the queue length in our experiments was at most 1. The queuing delay was around 68 microseconds when the queue length was 0 and corresponds to the interval that elapses before the invocation thread in the server gateway polls the request queue and dequeues the request. The queuing delay increased to as much as around 100 milliseconds for a request that arrived in the server's request queue just as the preceding request was beginning to be serviced. The reason for the increase is that the new request had to wait until the preceding request had been serviced. As mentioned earlier, the service duration for a request in our experiments was normally distributed with a mean of 100 milliseconds and variance of 50 milliseconds.

Our second observation from the graphs in Figure 3.6 is that the `static_all` scheme incurs fewer timing failures than the model-based scheme when the think time is significantly larger than the mean service time. The reason is that for larger think times ($\geq$ 250 milliseconds), the queuing delays are insignificant and the response time for a request is dominated by the service time, which, as mentioned earlier, is normally distributed with a mean of 100 milliseconds in our experiments. Hence, in cases in which the client-induced load is low, we increase the probability of timely response by having all the replicas service a client's request. However, when the interval between arrival of two successive requests at the replicas is comparable to or smaller than the mean service time, the increase in the probability of timing failures using the `static_all` scheme is considerably higher than for the model-based scheme. For smaller think time values ($<$ 250 milliseconds), the probability of a request experiencing a high queuing delay is higher in the `static_all` scheme than the model-based scheme, because in the `static_all` scheme all the available replicas service the requests of both clients. On the other hand, the model-based scheme selects only two replicas for the client that specified a probability of timely response of 0, while the number of replicas it selects for the second client, which requested a probability of timely response of 0.5 and 0.9, varies as shown by the graphs in the right-hand column of Figure 3.6. The above analysis leads us to conclude that the `static_all` scheme performs well when the interval between arrivals of requests at the servers is much larger than the mean

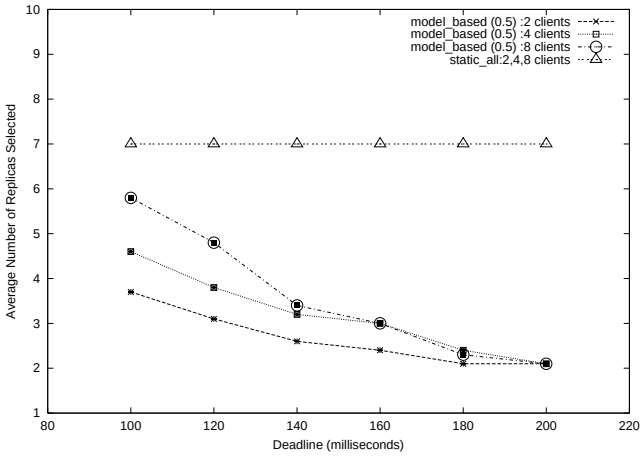
service time of a request. For smaller think times, as the queuing effects become dominant, a more dynamic selection scheme, such as the model-based scheme we have proposed, performs better. The reason is that a dynamic scheme adapts the replica assignment to meet a client's timing requirements by monitoring the changes in the responsiveness of the replicas.

Having compared the performance of the `static_all` scheme with that of the model-based selection scheme, we will now discuss how the ability of the model-based scheme to meet a client's probabilistic timing requirements varies as the client-induced load increases. When the queuing delays are small, as when the think time is 1000 milliseconds, the model-based scheme is able to meet the client's deadline requirements with a probability as high as 0.9 by choosing a replica subset from the set of seven available replicas (see Figure 3.6a and Figure 3.6b). The response time in that case is dominated by the service time, which is about 100 milliseconds on the average. The two-way gateway delay contributes about 1 to 2 milliseconds, while the queuing delay contributes about 68 microseconds. For smaller think times, however, the contribution of the queuing delay varies significantly, from values as low as a few tenths of a millisecond to values comparable to the mean service time. As the queuing delay increases, the response time also increases well beyond 100 milliseconds. Hence, for some of the deadline values, the model-based scheme is unable to find enough replicas to meet the deadline with the probability requested by the client. For instance, in Figure 3.6e, where the think time is 100 milliseconds, the model-based scheme is unable to find a replica subset to meet deadline values ≤ 120 milliseconds with a probability ≥ 0.9. Similarly, in Figure 3.6g, where the think time is 0 milliseconds, the model-based scheme is unable to find a replica subset to meet deadline values even as high as 200 milliseconds with a probability ≥ 0.9.

Thus, as the client-induced load increases, there are cases in which the available replica resources are insufficient for satisfying a client's QoS specification. In such cases, the model-based scheme sends the request to all seven available replicas, as shown in Figure 3.6f and Figure 3.6h. There are some alternative measures that the timing fault handler can take under such conditions, such as informing the client that there are insufficient resources to satisfy its QoS requirement, so that the client can choose either to renegotiate its QoS specification, or to be admitted at a later time when the replicas are less



(a) Probability of timing failure



(b) Number of replicas selected

Figure 3.7: Model-Based scheme vs. static scheme: variable number of clients

loaded. Alternatively, the middleware can provide support for adding more replicas when the service is in greater demand.

Varying the Number of Clients

A second method we used to vary the client-induced load was to vary the number of clients that accessed a service concurrently. All the clients used the same think time value of 250 milliseconds. One of the clients specified a different deadline in each run and requested that its deadline be met with a probability ≥ 0.5. All of the remaining clients specified a deadline of 200 milliseconds in each run and requested that this deadline be met with a probability ≥ 0.0. The plots in Figure 3.7 compare the performance of the model-based scheme with that of the static scheme using 2, 4, and 8 clients. Figure 3.7a shows the timing failure probability for each case, as measured at the client that specified that its probability of timely response should be at least 0.5, and Figure 3.7b shows the corresponding average number of replicas selected by the model-based scheme to meet the QoS specifications of this client.

When the client-induced load was low, as in the case when we used two clients, the performance of the static scheme was good. However, with more clients accessing the service, the `static_all` scheme resulted in more timing failures than the model-based scheme. Again, the model-based scheme

scales better than the `static_all` scheme for higher loads. As before, the poor scalability of the `static_all` scheme can be attributed to the more substantial role played by the queuing delay when there are more clients. In the case of the model-based scheme, the queue length was at most 1 when there were 4 or fewer clients accessing the service. The queuing delay varied from about 65 microseconds to about 100 milliseconds when the queue length was 1. When we used 8 clients in the model-based scheme, the queue length varied between 3 and 5, and the queuing delay correspondingly varied between 300 and 400 milliseconds.

In the model-based scheme, the two-way gateway delay was around 1 to 2 milliseconds when there were 2 clients, and 2 to 5 milliseconds when there were 4 clients. When there were 8 clients, the gateway delay varied between 5 and 10 milliseconds. This increase can be explained as follows. When the number of clients concurrently accessing the service increased from 2 to 8, the frequency of requests serviced by the replicas also increased. In the model-based scheme, since each replica broadcasts its performance update when it completes servicing a request, the number of performance updates broadcast by the replicas increases when more clients access the service. This adds to the network load and hence results in an increase in the gateway delay for the model-based scheme. The `static_all` scheme, on the other hand, does not use any performance broadcasts. However, since the broadcast messages are rather small (< 20 bytes), the increase in the gateway delay for the model-based scheme is less substantial than the increase in the queuing delay. Hence, the latter plays a more dominant role in the increase of the probability of timing failures for the model-based scheme with a larger number of clients. As we mentioned earlier in Section 3.5, if the gateway delay becomes as significant as the service time and the queuing delay, we can record the value of the gateway delay over a sliding window as we do for the service time and queuing delay, instead of considering only the most recently recorded value of the gateway delay as we currently do.

Model-Based Algorithm vs. Round-Robin Algorithm

We now compare the performance of the model-based scheme with that of the round-robin scheme. The round-robin scheme is a partially dynamic scheme in which each client's request is sent to two different

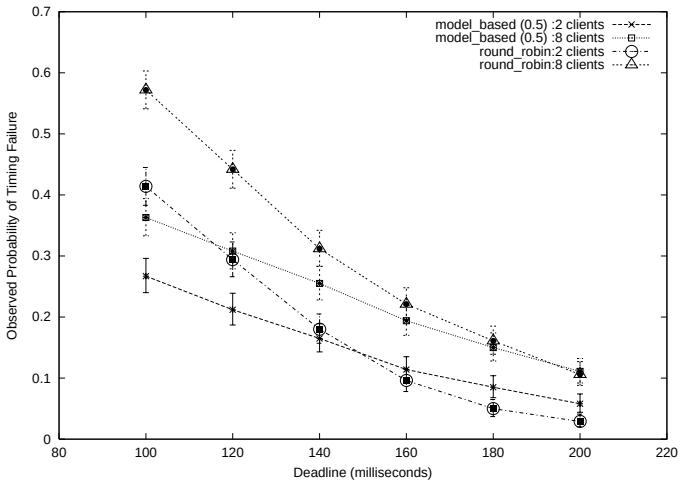


Figure 3.8: Model-Based scheme vs. round-robin scheme

replicas, which are selected from the replica list in a round-robin fashion. We chose a redundancy level of 2 for the round-robin scheme so that the round-robin algorithm, like the model-based algorithm, can tolerate single crash faults. The round-robin scheme is dynamic because it chooses different replicas for each request. However, unlike the model-based algorithm, the round-robin scheme uses a fixed redundancy level, and does not base its selection on the replica's performance history. Further, unlike the model-based algorithm, the round-robin algorithm does not incur overheads due to replica selection and performance broadcasts.

In order to compare the performance of the model-based selection scheme and the round-robin scheme for different workload intensities, we repeated the same experiments that we conducted earlier to compare the model-based and static schemes. However, this time we replaced the static scheme with the round-robin selection scheme to assign replicas to service all the clients. Figure 3.8 presents the results when we varied the number of clients from 2 to 8. One of the clients varied its deadline from 100 to 200 milliseconds and requested a probabilistic timeliness guarantee of at least 0.5. The figure shows the probability of timing failures observed by this client. The average number of replicas chosen by the model-based scheme for the client is the same as shown in Figure 3.7b. The round-robin scheme always chose two replicas for each client.

When the client-induced load increases, the queuing delays in the case of the round-robin scheme are not as high as for the `static_all` or model-based schemes. The reason is that the round-robin

scheme assigns pairs of replicas to service a client in a round-robin manner and is therefore able to balance the client-induced load across the available replicas better than the other two schemes. The model-based scheme assigns more replicas than the round-robin scheme for the same QoS specification, especially when the client-induced load increases, as can be seen in the graphs in Figures 3.6 and 3.7b. However, since the round-robin scheme chooses replicas without assessing their ability to meet a client's QoS specification, it is hard to say conclusively whether the replicas chosen by the round-robin scheme will be able to meet a client's requested probability of timely response, even when sufficient replicas are available. For example, suppose that the client had requested that its probability of timing failures be ≤ 0.1; from Figure 3.8 we can see that when there were 8 clients accessing the service, the replica assignment made by the round-robin scheme would have failed to meet the client's QoS specification.

In summary, a round-robin scheme is good when the primary goal is load balancing, rather than meeting specific QoS requirements of a client. It is also good when all the replicas have nearly identical performance characteristics (such as service time, queue length, and gateway delay) and exhibit little or no variability in their performance. However, in a system in which the replicas have different and variable performance characteristics, if the primary goal is to meet specific QoS requirements of a client, a more adaptive replica selection scheme, like the model-based scheme, is desirable.

3.7.4 Tuning the Frequency of Performance Broadcasts

As mentioned in Section 3.6.1, the replicas broadcast their performance updates each time they complete servicing a request. Our experimental results showed that this additional traffic could result in an increase in the gateway delay when there is a large number of clients. To reduce the overhead due to these performance broadcasts, especially when there are several clients accessing a service, we conducted experiments in which we implemented the following two optimizations:

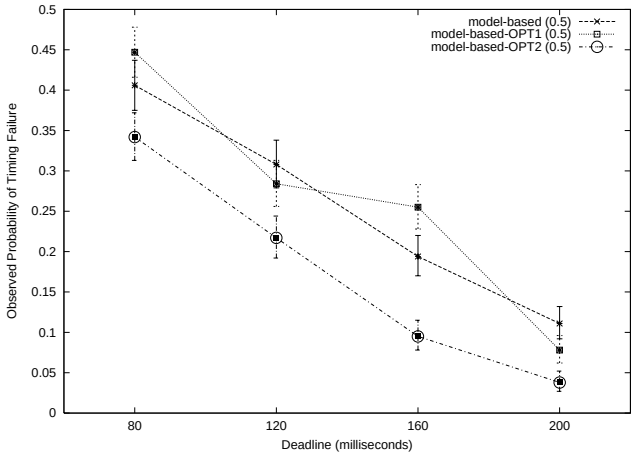
model-based-OPT1: We modified the model-based scheme so that each replica broadcasts its performance update only if the sum of the service time and queuing delay for a new request changed from the last recorded value by $\geq 20\%$.

model-based-OPT2: With this modification, we completely eliminated the performance broadcasts. Thus, we do not break up the response time variable, R, into its individual components (such as the service time, queuing delay, and gateway delay), as we did in Equation 3.2. Instead, each client gateway records the response time of a replica (t_r, defined in Section 3.6.2) upon receiving a response, and maintains a history of the response times. It then uses this history to compute the response time distribution for a replica and thereby predict the probability that the replica will meet the client's deadline. This prediction is then used by Algorithm 1 to select the replica subset, as before.

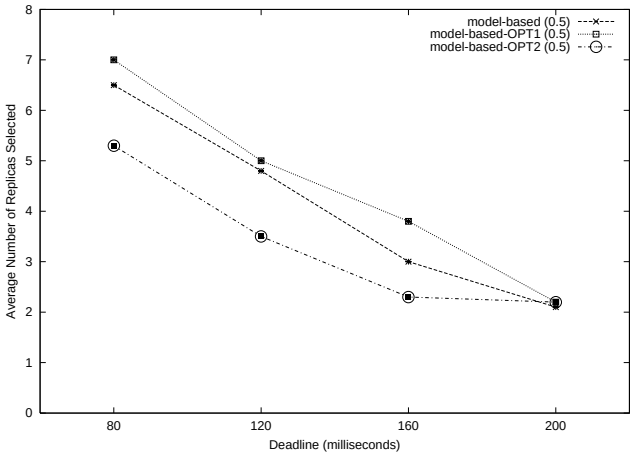
We conducted experiments to compare the performance of the unmodified model-based scheme with those of the model-based-OPT1 and the model-based-OPT2 schemes under different client-induced load conditions. The experiments were the same as those we described in Section 3.7.3 to compare the model-based scheme with the static and round-robin schemes. In Figure 3.9 we present the results obtained with eight clients accessing a service with a think time of 250 milliseconds; seven of the clients requested a deadline of 200 milliseconds in each run, with a value of 0.0 for the probability of timely response. The eighth client specified a different deadline in each run with a minimum probability of timely response of 0.5. Figure 3.9a compares the probability of timing failure observed by the eighth client for the three schemes, and Figure 3.9b compares the number of replicas selected for this client by the schemes.

We did not observe any significant difference among the performances of the schemes under low load conditions. For example, the performances of the three schemes were nearly identical when there were only two clients, each accessing the service with a think time of 1000 milliseconds. However, with eight clients accessing the service, as shown in Figure 3.9, we obtained a reduction of about 25% - 30% in the timing failure probability with the model-based-OPT2 scheme compared to the unmodified model-based scheme. We did not observe any significant improvement using the model-based-OPT1 scheme.

The appropriate frequency for publishing the performance updates depends on the system involved and the characteristics of the clients invoking the service. The publication frequency represents a



(a) Probability of timing failure



(b) Number of replicas selected

Figure 3.9: Varying frequency of performance broadcasts

tradeoff between reducing the load on the network and increasing the staleness of the performance data used for replica selection. We will now explain these tradeoffs using the model-based scheme and model-based-OPT2 scheme as examples. The two schemes lie at opposite ends of a spectrum. The advantage of a scheme, like the model-based scheme, is that it allows the decoupling of performance parameters that characterize a specific replica (such as the queuing delay, service time, host load, and queue length) from parameters that characterize the interaction of a specific replica-client pair (such as the gateway-gateway delay). We assume that the various clients accessing a service would like to be informed of any changes in a replica's performance characteristics. On the other hand, since the gateway-gateway delay depends on the path used for transmitting a message between a replica and a specific client, as well as on the size of the message transmitted, the gateway delay is useful only to the specific client involved. The decoupling of the replica-specific performance characteristics from the path-specific characteristics enables a replica to publish changes in its performance parameters to all the clients that may be affected by the change. This allows all the clients that access the same service to be aware of the most recent changes in a replica's performance without having to send a request to it. Thus, at the cost of incurring some additional network overheads, we reduce the possibility of using stale information during replica selection in the model-based scheme. This is especially useful in

a scheme, such as the model-based scheme, that is used in environments in which the replicas exhibit high variability in their performance characteristics over a period of time, thereby requiring quick adaptivity in the replica selection. Furthermore, broadcasting the performance updates to all the clients enables a client that accesses a service infrequently, or for the first time, to build a performance history for the replicas before it selects replicas for its next request.

On the other hand, in the model-based-OPT2 scheme, there is no decoupling of the individual components that contribute to the response time. Instead, a client gateway measures the end-to-end response time when it receives a response from a replica. The measured response time includes information, such as the gateway delay, that is specific to that client. In a typical distributed system, such client-specific information may be different for different clients. Hence, it may not be very useful to broadcast the response time that a client measures for its request to the other clients. As a result, using the model-based-OPT2 optimization, a client is aware of changes in a replica's performance characteristics only when it sends a request to the replica. While this results in lower network overheads compared to the model-based-OPT1 scheme, clients may have to use stale performance information, or send active probes to replicas to request their latest performance, at the time of replica selection. The model-based-OPT2 optimization is more useful in environments in which the replicas exhibit little or no variability in their performance characteristics over a period of time.

In summary, the experimental results we have presented show that at the expense of some computational overhead, the dynamic selection scheme we have developed makes effective use of the available replicas to meet the QoS specifications and thereby reduce the occurrence of timing failures, even when multiple clients access a service concurrently. While a scheme like the `static_all` scheme or the round-robin scheme would be sufficient when the clients do not have specific timing constraints, we believe that a dynamic scheme, like the model-based replica selection scheme we have developed, would be useful in an environment in which time-sensitive clients that have different QoS specifications access compute-bound service providers that display significant variability in their response times.

3.8 Related Work

In Section 3.1 we mentioned some of the related efforts that addressed the server selection problem with the goal of reducing the average response times, rather than meeting specific timeliness constraints. Our objectives are closely related to the efforts being made to build dependable, distributed object systems that also provide timeliness. Our research is also related to work that evaluates performance/reliability tradeoffs to address the problem of scheduling time-sensitive tasks to meet their deadlines in the presence of faults. We now summarize some of the related efforts in these areas.

3.8.1 Responsive Distributed Object Systems

The DREAM project [40] provides an integrated object-based framework for tolerating crash faults, value faults, and timing faults in a real-time distributed system. A real-time kernel (DREAM) provides real-time operating system support; a middleware layer that is layered above the kernel provides adaptive fault tolerance; and time-aware objects, which make up the top layer, implement all the object services. The middleware layer uses a primary-backup replication scheme to tolerate faults. The primary and secondary replicas receive a client's request and process it concurrently. In the absence of a failure, the primary replica responds to the client after notifying the secondary replica. If, however, the secondary replica does not receive the notification within a specific timeout period, it takes over as the primary replica and responds to the client. The DREAM approach differs from our approach in that it uses replication primarily to prevent timing failures that would occur as a result of replica crashes. Unlike our approach, which aims to reduce the occurrence of timing failures primarily by allocating the available replicas to service different clients according to the specific timing requirements of the clients, the DREAM approach simply allocates all the available replicas to service each client.

Like our project, the Timed Dynamic Method Invocation (TDMI) project [82] addresses the problem of meeting specific time constraints of clients that access a service implemented by CORBA objects. However, rather than use replication to meet the time constraints, TDMI follows a dynamic priority-based scheduling approach, in which it schedules the requests at the server according to their

time constraints. The TDMI approach allows CORBA clients to use an enhanced IDL interface to express their timing constraints and importance when invoking a CORBA method. In the TDMI approach the timing constraints are deterministic, whereas our approach uses a probabilistic QoS specification. Following a real-time scheduling policy, a global scheduling service uses the client inputs to assign a global CORBA priority to the request. Each server then maps this global priority to its native system priority to determine the order in which it has to service requests. Global priorities may change dynamically depending on resource availability as well as arrival and completion of other requests. If a request does not meet its deadline, the client is notified through an exception. Unlike our approach, TDMI does not attempt to tolerate the crash of a server when it is servicing a request.

3.8.2 Scheduling for Timeliness and Fault Tolerance in Real-Time Systems

Closely related to our research goals is the work described in [81], in which Wang, Ramamritham, and Stankovic have addressed the problem of determining the redundancy level with which a task has to be scheduled in a fault-tolerant, real-time system. However, their work addresses the problem in the context of a multiprocessor system subject to hardware faults, unlike our work, which focuses on a distributed system. Their approach associates a reward value with each task when it successfully completes within its deadline, and a penalty value if it is not admitted in the system due to lack of resources or if it misses its deadline after admission. They build a mathematical model that evaluates the performance index of a task based on its reward value, penalty value, and reliability. They then determine the appropriate redundancy level for the tasks in the system using analytical models to maximize the system's performance index, which is the sum of the performance indices of the individual tasks in the system at any point in time. Thus, unlike our work, which uses local schedulers to determine the redundancy level with which an individual request has to be serviced in order to meet the client's QoS specification, the work presented in [81] uses a global scheduler that is aware of the task schedules of all the tasks in the system. The global scheduler determines the redundancy level with which the tasks have to be scheduled in order to maximize the performance index of the system.

The imprecise computation technique [48, 32] is another scheduling approach that addresses the

problem of preventing timing failures under transient overload conditions in certain dependable, real-time application domains, such as radar tracking, collision detection systems, and voice and video applications. The technique is based on the premise that these applications can make a tradeoff between the precision of a result and its timely delivery. Hence, in a system based on that technique, each time-critical task is scheduled in such a way that it can produce an approximate, but usable result whenever a failure or overload prevents the system from delivering a precise result. A client that uses the imprecise computation technique divides its task into a *mandatory* and an *optional* part. Upon completing its execution, the mandatory part yields a result that has at least the minimum precision required by the client, while execution of the optional part only improves the precision of the result. The goal of the imprecise computation scheduler is to find a feasible schedule that prevents timing failures by allocating enough time to compute the mandatory portion of each task within the specified deadline so that it can deliver a response that has at least the minimum precision requested by the client. Any slack time that is available is divided among the tasks to execute their optional portions so that the accuracy of their results can be improved. In contrast, in our work, the clients require that every request produce the correct response at the expense of allowing the temporal requirements to be met probabilistically.

Summary

We have presented a new approach that tolerates timing faults in replicated services. The approach uses an algorithm that chooses replicas dynamically at request time, based on their ability to meet a client's time constraints in the presence of delays and replica crashes. An important contribution of this work is the definition of a probabilistic model that uses the performance updates that have been received from the replicas, to predict, at runtime, the probability that the response from a replica will arrive by a given time. The prediction is made by a scheduler, which is part of the timing fault handler, and is used to select a set of replicas that can meet a client's time constraint with at least the probability requested by the client. We have implemented the selection algorithm in AQuA, which is an infrastructure for building dependable distributed applications, and obtained experimental results that show its efficacy.

Our model and selection algorithm can be extended to other environments that host replicated services and provide a mechanism for tracking and recording a performance history of the replicas.

Chapter 4

A Survey of Weak Replica Consistency Models

Look to the past for guidance into the future.

- Robert Goodkin

The previous chapter addressed the problem of meeting the temporal constraints of applications when the replicated state is static. We assumed that either the replicated objects were stateless or the operations that were performed on them by the clients did not result in modification to the replicated state. Hence, we did not consider the state of the replicas when estimating their responsiveness. Our goal is to extend that work by considering the case in which the replicated state varies with time. An important problem that arises is that of maintaining replica consistency. Several consistency models have been proposed to address the needs of different applications. In our work, we primarily target time-sensitive applications that can tolerate a certain degree of relaxed consistency, in exchange for better response times. In this chapter, we describe some of the key challenges in maintaining replica consistency and survey some of the consistency models proposed by other researchers, primarily focusing on weaker consistency models. This overview will form the basis for the framework we propose in the next chapter for meeting temporal constraints in the presence of consistency requirements, when accessing replicated objects.

4.1 Introduction

Replication of distributed services enables us to service multiple clients concurrently and deliver good response times by selecting different replicas to service different clients. Concurrency, however, has the potential for introducing replica inconsistency. Hence, one of the challenges in replicating distributed services is the problem of supporting concurrent client operations while ensuring that the replicated state does not diverge in an unacceptable manner. Traditional replica consistency models provide a binary view of consistency: *strong* consistency models that guarantee *immediate* convergence, or *weak* consistency models that provide *eventual* convergence. In the strong consistency model based on one-copy equivalence, concurrent operations on replicated data are equivalent to a serial execution on non-replicated data. Pessimistic replication algorithms, such as active and passive replication (e.g., [65, 55, 67]), have traditionally been used to maintain strong consistency among replicated data. Although these algorithms, which provide single-copy semantics, ensure correctness for a wide class of applications (e.g., banking transactions), the performance overheads incurred in enforcing one-copy equivalence to maintain mutually consistent replicas may be unreasonably high for clients that do not require strong consistency. Further, strong consistency may not be a viable option in environments in which some of the replicas run on hosts and links that either are inherently slow, or tend to become slow due to transient overloads and failures.

On the other hand, in the weak consistency model, operations are performed on some subset of replicas, and the updates are propagated to the other replicas either lazily or on demand. Typically, in the weak consistency model, the only guarantee provided to the clients is that the replicated state will eventually converge, if update activity ceases. Several optimistic replication algorithms (e.g., [12, 59, 70, 20]) have been proposed for applications that can tolerate relaxed consistency. These algorithms allow a client to access any replica in order to provide better responsiveness, unlike the pessimistic algorithms, which allow access only to those servers that have the most up-to-date state. However, if the clients access different servers before their states converge, the resulting inconsistency may lead to conflicts. Optimistic algorithms mainly target applications for which the probability of occurrence of these conflicts is rather small.

Although optimistic replication algorithms have been studied for a long time, their use has become more popular with the growth of the Internet and mobile computing. Usenet [37], the Internet bulletin board system, is one of the well-known replicated services that is based on optimistic replication. Each server replicates all the news articles so that a user can read any article from the closest server. The postings are propagated among the servers by periodic flooding. While this periodic propagation causes a delay before an article posted on one server reaches another server, many users are willing to tolerate this delay in return for Usenet's availability. Optimistic replication is used in directory services (e.g., Clearinghouse [12], Active Directory [51], and DNS [52]), in file systems (e.g., Ficus [20] and Coda [71]), in Internet services (e.g., [70]), and in database systems for mobile and disconnected operations (e.g., Bayou [77] and Lotus Notes [53]). Optimistic replication techniques can also be used to support groupware and collaborative applications, such as shared whiteboards and shared calendars [13, 3].

When dealing with weak replica consistency, an important challenge is the problem of controlling the divergence in the replicated state so that we can deliver information that is meaningful to the client applications. In order to do so, we need to define appropriate measures that allow an application to express the degree to which it can tolerate relaxed consistency, and design mechanisms to propagate updates and resolve conflicts, if they arise. We now discuss how these issues have been addressed by some of the related work.

4.2 Consistency Measures

Consistency has several dimensions and it is hard to capture it using a single metric. Several measures have been proposed to control the degree of replica inconsistency along the ordering and temporal dimensions. Consistency models used in DNS [52], stock quote servers, quasi-copies [2], timed consistency [79], and epsilon-serializability [60] bound inconsistency along the temporal dimension. They guarantee that the updates are propagated to the replicas within a finite amount of time based on the assumption that the network links are reliable. Timed consistency models, defined in [79, 41], require that if a write is executed at time t, then the effect of the write should be visible to others by $t+x$, where

x is the maximum acceptable delay for propagating the effect of the write. The value of x represents a trade-off between consistency and scalability. By choosing appropriate values of x, it is possible to represent other well-known consistency models as a special case of the timed consistency model. For example, the timed consistency model reduces to *linearizability* when $x = 0$; it reduces to sequential consistency when x does not have a finite upper bound.

Beehive [75] introduces the notion of *delta consistency*. Similar to timed consistency, delta consistency allows an application to specify a delta value that bounds the inconsistency delivered to the application. The delta consistency model requires that a read of a memory location return the latest value that was written no more than delta time units before the read operation, where the delta is an application-specified temporal tolerance given in virtual time. It is a contract that bounds the staleness of values presented to a consumer to at most delta units before the time of the read operation. The notion of *epsilon-serializability* mentioned in [60] is another way of bounding the level of replica divergence. Similar to the delta parameter in delta consistency, epsilon is a tunable parameter whose value can be used to control the inconsistency perceived by a user for different operation semantics.

Some researchers have proposed measures to provide probabilistic consistency guarantees (e.g., [84, 61]). For instance, when there is a large number of replicas, we can bound the delay to propagate an update to a certain fraction of replicas, rather than the entire replica set. The delay may be expressed in real-time or as the number of rounds for which a local server should hold back an update. The client that is interested in urgent updates can set a minimal delay, while a client with more tolerance for relaxed consistency can opt for a larger update delay. Another probabilistic measure is the model proposed in [84], which guarantees that with a certain probability P, the value of an object returned to a client will be temporally consistent with the newest copy of the object in the system. The probability depends only on the update patterns of the servers rather than on the underlying latency.

Another way to quantify inconsistency is to express it in terms of the number of concurrent operations that can be allowed, in terms of the number of update operations with which a read operation can overlap [60], or in terms of semantic metrics [83]. For example, an airline reservation system is an example of a system in which bounded inconsistency guarantees in terms of semantic metrics may be

useful, so that the degree of overbooking can be limited to within x number of seats to avoid causing inconvenience to the passengers. Some of the file systems that support optimistic replication use conflict count or conflict rate as a way to limit inconsistency (e.g. [42]). The drawback of that approach, however, is that conflicts are detected only upon reconciliation, and unresolved conflicts may multiply the number of conflicts. So the frequency and pattern of reconciliation have a direct effect on the number of conflicts observed. Furthermore, the definition of a conflict is very application-specific.

Some measures have also been proposed to express consistency along the ordering dimension. For example, Bayou supports four types of ordering guarantees on a per-session basis [13, 76]: read your writes, monotonic reads, monotonic writes, and writes follow reads. The first two provide causal read guarantees. A banking transaction is an example of an application in which "read your writes" guarantees would be useful. When a client checks his latest balance, he expects that it will reflect the result of all his previous deposits and transfers. That information can be correctly provided by a causal read guarantee, which preserves an ordering between the reads and writes, such that the reads incorporate the previous writes by the same user. Monotonic reads guarantee that two successive reads by the same user return increasingly up-to-date contents. Providing causal read guarantees is trivial when a user's requests are directed to the same replica, but becomes challenging if the user's accesses are forwarded to a different replica on each request.

Finally, the TACT middleware [83] uses a combination of all the above measures to provide tunable consistency and availability. TACT allows applications to specify their desired consistency level on a *conit*, which is a physical or logical unit of consistency. The consistency measures used by TACT to bound the level of inconsistency include the *order error*, which limits the number of tentative writes that can be outstanding at any replica; the *numerical error*, which bounds the difference between the value delivered to the client and the most consistent value; and *staleness*, which places a real-time bound on the delay of write propagation among the replicas. Applications that require strong consistency specify a value of zero for all three metrics, while applications that can tolerate weak consistency do not specify any bounds.

4.3 Protocols for Update Propagation

Having looked at some of the measures for expressing consistency, we now describe some of the factors that distinguish protocols designed to achieve convergence of replicated state. Protocols used to achieve replica consistency adopt different update protocols, depending on the consistency guarantees they provide. The update protocol determines where an update can be made, and when and how the update is propagated.

4.3.1 Source of the Update

In a *single-master* system (e.g. [67]), one of the replicas is considered to be the master, and its copy is designated the primary copy. The other replicas are considered to be backups, and their copies are regarded as secondary. A client is allowed to update only the primary copy directly. The updates are then propagated to the backup replicas by the master. The potential for conflicts in single-master systems is low, because of the restriction that the clients can update only the primary copy directly. However, the disadvantage of these systems is that the master can become a single point of failure. Single-master schemes are typically used by pessimistic replication algorithms.

Multi-master systems (e.g. [50]) allow updates to be issued at any replica. The replicas periodically exchange their updates, resolve conflicts if there are any, and merge their updates to reach a consistent state. While multi-master systems provide better availability, they may suffer from the problem that some of the updates may have to be rolled back when there are conflicts.

4.3.2 Unit of State Transfer

There are several ways to propagate an update. An update may be propagated by transferring the entire state of the object. Another way is to incrementally transfer only the portion of the state that has been updated. A third alternative is to propagate the update operation rather than the content. Each of the alternatives has different pros and cons. Transferring the entire state incurs communication overhead, especially if the object state is large and only a portion of the state is updated. However, by batching

the effects of multiple update operations, not only can we reduce these communication overheads, but we can avoid the need for each replica to perform each update operation individually. Timestamped vectors allow the modified state to be transferred incrementally [50, 22, 43]. In that scheme, each replica timestamps each update it has committed, usually using a logical clock [44]. During a state transfer, the timestamp vectors of the sender and recipient are compared, and only the missing updates are exchanged. Instead of transferring the state, one can also achieve replica consistency by propagating the update operations and allowing each replica to perform the update locally. While the latter option reduces the usage of network bandwidth, it consumes more CPU resources.

4.3.3 Direction of State Transfer

Different update protocols differ in who is responsible for initiating the state transfer. In pull-based algorithms [50], each replica is responsible for polling other replicas and pulling their updates. This is well-suited for mobile environments where the replicas may be not be connected all the time. In push-based algorithms, the replica whose state was updated is responsible for initiating the update propagation. This is more attractive in fully connected environments. In *blind pushing*, a replica with an update blindly publishes a new update to its group members or to all the other replicas. Single-master systems, Usenet [37], and routing protocols are examples of such services. In multi-master systems in which the same update may be disseminated multiple times, it is the responsibility of the receiver to discard duplicates [12].

In a gossip or anti-entropy protocol [22, 78], pairs of replicas exchange updates with each other. The gossip partners may be chosen at random or based on network topology. The bimodal multicast [6] is a gossip-based protocol that ensures that the probability that almost all or almost none of the processes will receive a message is very high. In this protocol, a message is sent by an initial unreliable multicast, and is followed by gossip that propagates the message to members that did not receive the initial multicast. Gossip proceeds in rounds, which are not synchronized between processes. In each round, a process randomly selects another process in the group and sends it a digest of its most recently received messages. The recipient may then request retransmission of a message that it has not

received. After holding each message for a specific number of rounds, the replica may garbage collect the message.

4.3.4 Frequency of Updates

Another important factor is deciding when the updates should be propagated. The frequency of update propagation represents a tradeoff between the overhead involved and the rate at which the states of the replicas converge. Applications that require stronger consistency guarantees need the updates to be propagated more frequently than those applications that can tolerate relaxed consistency. In pull-based algorithms, the replicas may poll other replicas periodically based on the frequency of writes. In push-based algorithms, the source replica may push the state immediately after an update, periodically, or by batching a certain number of update operations together. TACT uses a combination of pull-based and push-based state transfer to ensure that the staleness of the replicas is within the application-specified staleness threshold.

4.3.5 Ordering Updates

An algorithm that merely ensures that all the replicas receive the same set of updates is not sufficient to guarantee replica consistency. It should also ensure that the replicas apply the updates in an order that results in consistent state. Updates may be totally ordered or partially ordered. Total ordering can be achieved by ensuring that all the replicas apply the updates in the same order. There are several ways to ensure that they do. For example, Golding's approach [22] uses *ack vectors* to estimate the state of the replicas. An ack vector AV is an N-element array of timestamp vectors (TV), where N is the number of replicas. Replicas exchange ack vectors and update them like timestamp vectors. The element $AV[j]$ on a replica j is the $min_{0 \leq i < N} TV[i]$, which implies that replica j has not received any update newer than $AV[j]$. $AV[i]$ on j represents j's conservative estimate of the most recent update received by replica i. All updates older than the $min_{0 \leq i < N} AV[i]$ are guaranteed to have been received by all the replicas. Therefore, they can be sorted and committed. This approach ensures total ordering in a decentralized manner and never aborts updates. However, it may result in livelocking, because

a single slow replica can prevent the progress of the ack vectors of all the other replicas by keeping $min_{0 \leq i < N} TV[i]$ from increasing.

Another approach, which is used in Bayou [59], designates one of the replicas as the primary. The primary replica is only responsible for deciding the order in which the updates will be committed by all the replicas. It does this by assigning a commit sequence number (CSN) to each update when it commits it. The mapping between an update and its CSN is propagated to the other replicas. The remaining replicas then commit the updates in increasing order of the CSN. The problem with this approach is that using a single primary to decide the order may not scale well when there is a large number of replicas.

A third approach, used by Deno [39], is based on an optimistic version of the quorum consensus protocol [19, 30, 34, 49]. Deno assigns each replica a weight in such a way that the weight of the replicas adds up to 1. Each update is associated with a vote that increases by an amount that is equal to a replica's weight, when the update is received by the replica. When multiple updates circulate simultaneously, the earliest update that gets the majority wins and the other updates are discarded. All the replicas are informed about the update that won the vote. Thus, this approach applies the updates in the order in which they acquire majority votes. Deno's approach is adaptive; when the entire weight of 1.0 is assigned to a single replica, it reduces to the case in which a single replica is responsible for deciding the update order, as in Bayou. On the other hand, assigning the same weight to all the replicas reduces Deno's approach to a majority voting scheme. The drawback of the Deno approach is that it discards updates based on the assumption that all updates propagate equally fast among all the replicas. Hence, it may not be suitable in heterogeneous environments in which some of the replicas may be on slower links than the others.

Total ordering has some shortcomings. For example, it cannot preserve dependencies among updates applied at different replicas. Further, total ordering may not be suitable when semantic constraints have to be considered. Several systems address the shortcomings of total ordering by taking advantage of semantic and causal relationships. For example, some systems treat commutative operations, such as additions on numeric data [83, 60], distinctly. Some allow causal relationships to be explicitly spec-

ified by the user [43], while others define sessions and implicitly assume that a user's update depends causally on the previous updates recorded in the user's session [77, 20].

4.4 Conflict Resolution

An important aspect of optimistic consistency protocols is the detection and resolution of conflicts. Two updates conflict when a replica performs an update to a piece of data before incorporating all the previous updates made to the same piece of data. Conflict resolution is application dependent and sometimes has to be manually resolved. Conflicts can be detected by associating each update with the issuer's timestamp vector at the time the update was performed. If one update's timestamp vector dominates another update's vector, then the two do not conflict. Otherwise the updates result in a conflict. Microsoft's Active Directory [50] uses a conflict resolution approach in which the site that has more updates wins the conflict. If the number of updates is the same for the sites, the timestamp is used to resolve the conflict in favor of the more recent modification.

Summary

In this chapter we briefly reviewed some of the traditional notions of replica consistency. The traditional models use a binary approach and support either strong or weak replica consistency. We looked at how the binary model can be redefined to support tunable consistency. The two main aspects of such tunable consistency models are 1) definition of appropriate measures to quantify consistency and 2) design of protocols that control the inconsistency perceived by the users. The survey presented in this chapter forms the basis for the work we describe in the next chapter.

Chapter 5

Tunable Consistency and Timeliness

The more unpredictable the world is, the more we rely on predictions.

- Steve Rivkin

In Chapter 4, we discussed some ways of measuring consistency and outlined some of the challenges involved in maintaining consistency among the replicas, when the replicated state is dynamic. In this chapter, we discuss the framework we have designed to support tunable consistency and timeliness. The framework enhances the work we described in Chapter 3, by allowing a client to access replicated information based on its consistency requirements, in addition to its timeliness constraints. We begin by explaining the need for a framework that supports tunable consistency and timeliness, by describing a few motivating applications in Section 5.1. Our work targets client applications that have specific temporal and consistency requirements. These applications can tolerate a certain degree of relaxed consistency, in exchange for better response times. In Section 5.2, we propose a QoS model that allows a broad spectrum of applications to express their timeliness and consistency requirements. In Section 5.3 we describe a two-level hierarchical organization of replicas, which forms the basis of our framework for providing tunable consistency. One of the features of the framework is that it allows us to build protocols for providing different consistency guarantees and use them on demand. These protocols use a combination of immediate and lazy update propagation to ensure that the states of the replicas in the two-level hierarchy do not diverge in an unacceptable manner. As a proof-of-concept, in Section 5.4, we describe how we use this adaptive framework to maintain replica consistency by implementing sequential and FIFO ordering guarantees. One of our main goals is to meet the QoS

specification of the clients by selecting the appropriate replicas to service their requests. In Section 5.5 we describe a probabilistic model that we use to guide the selection process. Similar to the work we presented in Chapter 3, this model uses the performance history of replicas obtained by online performance monitoring to predict a replica's ability to meet a client's QoS specification. However, while our previous work assumed that the replicas had static state, our current model addresses the selection problem in the context of replicas with dynamic state.

5.1 Motivation

With the increasing shift towards dynamic content in emerging distributed applications, it is hard to guarantee that the content delivered to the clients of the applications will always be up-to-date. Providing this guarantee is especially hard if the clients require timely delivery. It is well-known that there is a fundamental tradeoff between consistency and responsiveness in a replicated service, and that providing stricter consistency guarantees is likely to result in higher response times. However, we need a framework that allows us to analyze and quantify the tradeoffs between consistency and timeliness.

In this work, our goal is to support client applications that are time-sensitive. These applications can relax their consistency requirements in exchange for improving the probability that their response time constraints can be met. However, in order for the response to be meaningful, they need some bounds on the inconsistency in the response they receive. We argue that it would not be possible to effectively support such applications that have specific temporal and consistency requirements by viewing consistency as a binary property, as traditional consistency models did. Instead, we allow the applications to express their timeliness and consistency requirements as QoS specifications, and propose an adaptive middleware framework that allows us to explore the use of *tunable consistency* models to study the tradeoffs between timeliness and consistency. The key challenges involved in designing such a framework are to control the inconsistency in the replicated state, and to select replicas to service a request, based on the specified QoS.

Several applications motivate the need for a framework that supports tunable consistency and time-

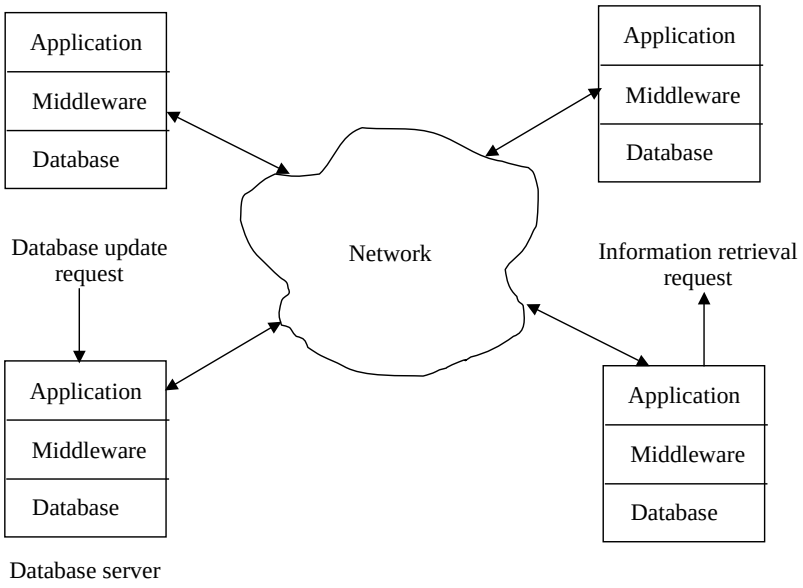


Figure 5.1: Motivating application scenario

liness. Such a framework would be especially useful for a real-time database application that stores time-varying information, such as a distributed directory service. We now describe a few representative applications.

Ticket Reservation System: Consider a ticket reservation system that stores the status of the seats in a replicated database, as shown in Figure 5.1. Such a system may be used for reserving tickets on an airline or train, or for an event, such as a popular concert. Each copy of the database is associated with a server and the state of a server is simply the information in its copy of the database. Users may contact the replicated servers either to make a reservation or to query the schedule and seat availability. When a reservation is made, it results in an update to the state of the replicated servers. If the volume of requests is high, it may not be possible to propagate the updates to all the replicas immediately. Therefore, some of the replicas may have out-of-date information, but may be able to respond quickly to inquiries about the seat availability. Users that are interested in a quick response may be satisfied with approximate information, provided the information is not significantly out-of-date. A framework that supports tunable consistency and timeliness would allow users to access information based on their timeliness and consistency constraints.

Electronic Patient Recording System: Consider a hospital information system in which all of a

patient's medical records are maintained in an electronic database. The goal of such a system is to make a patient's medical records, lab results, and images available to clinicians as well as patients remotely. The database may be replicated across the hospital intranet, as shown in Figure 5.1, in order to improve availability. The replicated database servers may receive requests from doctors, nurses, or other hospital staff to update a patient's record. It may take a while before the updates are propagated to all the replicas, resulting in some of the replicas being out-of-date. The replicated servers may also receive requests to retrieve a patient's record. For example, a patient may wish to lookup his record in order to have a prescription filled. Alternatively, in case of an emergency, such as an accident, medical professionals may want to lookup a victim's medical history before treating him. Under such time-critical circumstances, it may be more important for the medical professionals to access information in a timely manner, even if it is a little out-of-date, rather than delay treatment while waiting for the most recently updated history. Again, a framework that supports tunable consistency and timeliness would allow server replicas to be selected based on their ability to supply information that has the appropriate level of consistency in a timely manner.

Traffic Monitoring Application: In this application, sensors track the traffic at different street intersections and periodically update the servers that store the traffic status. At the same time, servers receive inquiries from drivers about the traffic status. The servers may be replicated to provide better response times and fault tolerance. The data recorded by the replicated servers is highly time-varying, and some of the replicas may not have the most recent updates. Rather than wait for the most recent updates, drivers may be satisfied with approximate information about the traffic status, provided the information is not significantly obsolete and the response time is small. Instead of placing the responsibility of choosing the servers on the drivers, we can make use of a framework that supports tunable consistency and timeliness to mediate access to the replicated servers. Such a framework would allow drivers to access information based on their timeliness and consistency requirements.

Online Stock-Quote Application: An online stock-trading application is another example of a system that would benefit from a framework that supports tunable timeliness and consistency. In such an application, users frequently buy and sell stocks, which affects the stock quote information stored

on the servers. Users also access the servers to retrieve the stock quotes. It may not be possible to deliver the most recent quote, especially if the user needs a quick response. This is another example of a service for which differentiated timeliness/consistency QoS is important, because different clients have different requirements for the freshness of the data.

5.2 QoS Model for Timed Consistency

The QoS model we propose for providing timed consistency extends the model described in Section 3.3 by allowing a client to specify its consistency requirement, in addition to its temporal requirement. This extended consistency specification includes two new attributes: the *request* attribute and the *consistency* attribute.

5.2.1 Request Attribute

In order to allow concurrent requests while at the same time ensuring bounded inconsistency, we propose a request model that allows our middleware to distinguish invocations that modify the state of the object they invoke from those that merely retrieve the state. If we do not allow this distinction to be made, the middleware will be forced to assume that all invocations result in modifications to the states of the objects they invoke. Such a pessimistic approach would, however, needlessly penalize the performance of the truly concurrent operations. Since the middleware is not aware of the application semantics, the application must be allowed to specify whether or not an operation modifies the state of an object. However, the CORBA invocation model that we use in our work does not have a provision that allows applications to make this distinction.

To address that shortcoming, our QoS model allows a client application to identify all the *read-only* methods it invokes on an object by their names at the beginning of a session. If an operation is not specified as read-only, then our middleware considers it to be an *update* operation. A read-only operation retrieves the value of one or more data fields of the object and responds to the client, but does not cause any modification to the state of the object. An update operation is any invocation that

modifies the state of the object on which the operation is performed, and may be either a *write-only* operation or a *read-write* operation. For example, in a sensor-tracking application, the sensors that record new values perform update operations, which modify the state of the object. On the other hand, display clients that retrieve new values and display them on the external monitors perform read-only operations. An alternative approach for specifying the request attribute would be to allow the servers to advertise their read-only interfaces to the clients.

5.2.2 Consistency Attribute

In Section 4.2, we looked at some of the ways of measuring consistency. We saw that different applications have different views of consistency and that it is therefore it is hard to capture consistency requirements using a single metric. It is important to choose metrics that capture the requirements of a broad range of applications. Our QoS model regards consistency as a two-dimensional attribute: <*ordering guarantee, staleness threshold*>. The *ordering guarantee* is a service-specific attribute that denotes the guarantee that a service provides to all its clients about the order in which their requests will be processed by the servers, so as to prevent conflicts between operations. Some well-known ordering guarantees that a service can offer are sequential (or total), causal, and FIFO [5]. In this thesis, we target services that provide sequential and FIFO ordering guarantees.

The *staleness threshold*, which is specified by the client, is a measure of the maximum degree of staleness a client is willing to tolerate in the response it receives. In other words, the threshold indicates how "fresh" the client wants the information it accesses to be. In our framework, the staleness of a response denotes the staleness of the state of the replica that sent the response. In this paper, we will use the terms "staleness of a replica" and "staleness of a replica's state" synonymously. We compute the staleness of a replica by associating a timestamp with each update operation. We use timestamps based on "logical clocks" [44] because this obviates the need for synchronized clocks across the distributed replicas. These logical timestamps make it possible to specify the staleness in terms of "versions." For example, in a document-sharing application, a replica whose staleness is x is one whose state has not yet been updated to reflect the modifications ensuing from the most recent x updates. The replica's state,

however, reflects the modifications of all updates committed prior to that. Although we measure the staleness using logical clocks, it should be possible to design an equivalent model that uses timestamps based on real clocks. That would allow a user of, say, an online stock-trading application to retrieve a quote that is no more than *x* seconds old. In order to meet a client's QoS specification, a response delivered to the client should be no more stale than the staleness threshold specified by the client.

5.2.3 Timeliness Attribute

The timeliness specification is also specified using a pair of attributes: <*response time, probability of timely response*>. This pair specifies the time by which a client expects a response after it has transmitted its request, and the probability with which it expects its temporal constraint to be met. Failure to meet a client's deadline results in a *timing failure* for the client. These timeliness attributes were explained in Section 3.3. In our QoS model, the timeliness attribute is applicable only for read-only requests and not for update operations.

As an example of the use of the above QoS model, consider a document-sharing application in which multiple readers and writers concurrently access a document that is updated in sequential mode. Using the above model, a client of such an application can specify that he wishes to obtain a copy of the document that is not more than 5 versions old within 2.0 seconds with a probability of at least 0.7. The timeliness and age specification can be either specified at start-up time in a configuration file (see Appendix A) or renegotiated at runtime as often as the client wants. The ordering guarantee, which is used to select the protocol to communicate with a service, is decided at start-up time, and is assumed to remain constant for the entire duration of a session.

5.3 Adaptive Middleware Framework for Tunable Consistency

Given the above QoS model, our goal is to build a framework that can be easily tuned to support the different application-specific requirements at the middleware layer. In order to design this framework, we address three main issues: 1) organization of the replicas, 2) development of protocols that implement

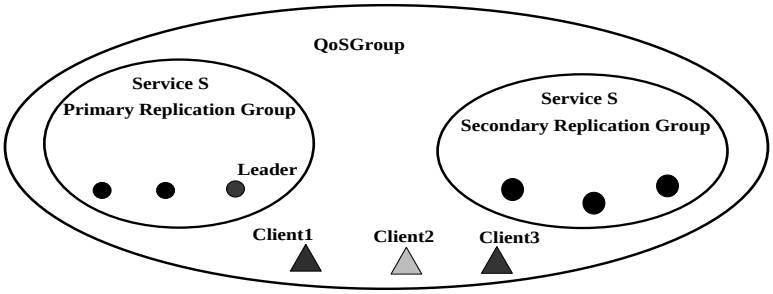


Figure 5.2: Replica organization for adaptive consistency

different consistency semantics and design of an infrastructure that would allow the protocols to be used on demand, and 3) development of a mechanism to select replicas to service a client dynamically based on the client's QoS requirements. We will now describe the approach we have used to address these issues in the context of the AQuA middleware.

5.3.1 Hierarchical Replica Organization

We organize the replicas into different groups, as shown in Figure 5.2. All the replicas offering the same service are organized into two groups: a *primary replication group* and a *secondary replication group*. We also have a *QoS group*, which encompasses all of the replicas of a service and their clients. In our implementation, all of these groups are derived from Maestro groups [80], and members of a group communicate with each other by making use of the Maestro-Ensemble group communication protocol [28]. For each group, Ensemble elects one of the members of the group as the *leader*. However, only the leader of the primary group is relevant to this work. We depend on Maestro-Ensemble to provide reliable, virtual synchrony, and FIFO messaging guarantees, and build upon these guarantees to provide the different end-to-end consistency guarantees. We also depend on Maestro-Ensemble to inform the group members when changes in the group membership occur.

The primary and secondary replication groups may be used to organize the replicas of an object adaptively to implement multiple consistency semantics. The primary replication group is used to implement strong consistency semantics, whereas the secondary group implements weaker consistency semantics. The size of these groups can be tuned to implement a range of consistency protocols. For example, when the secondary group is empty and all the replicas are placed in the primary group, the

replica organization supports an active replication approach (e.g., [65, 55]), in which all the replicas implement strong consistency semantics. On the other hand, by placing all the replicas in the secondary group, we can support concurrent operations in the absence of consistency requirements, as we described in Chapter 3 for the case in which the replicas were stateless. A third way to adapt would be to implement a primary/backup protocol with multiple backup replicas (e.g., [67]), by placing one of the replicas in the primary group, and all the remaining replicas in the secondary group.

In many applications (e.g., file servers and databases), read-only operations are more frequent than the operations that modify the state of the object [23, 71]. The above two-level replica organization was motivated by the need to favor the read operations that can tolerate relaxed consistency to a certain degree in exchange for a timely response. While a write-all scheme that writes to all the replicas concurrently always provides access to the latest updates, it may result in higher response times for the read operations. We therefore reduce the overheads incurred by a write-all scheme by performing the updates on the smaller primary group, while allowing the secondary replicas, which are greater in number, to handle the read-only operations of different clients. The primary replicas subsequently bring the state of the secondary replicas up to date using lazy update propagation. The degree of divergence between the states of primary and secondary replicas can be bounded by choosing an appropriate frequency for the lazy update propagation. Thus, while clients that need the most up-to-date state to be reflected in their response may have to depend more on the response from a primary replica, clients that are willing to tolerate a certain degree of staleness in their response can achieve better response times, due to the higher availability of the secondary replicas. Although in our work we restrict ourselves to a two-tier organization of replicas in order to study the tradeoffs between timeliness and consistency, it should be easy to extend our architecture to multiple tiers representing intermediate degrees of staleness in the replica states.

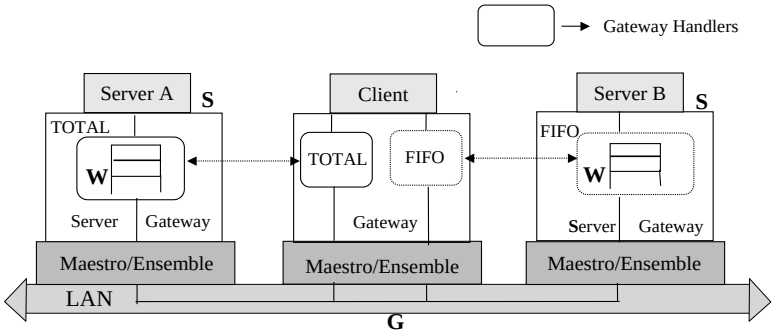


Figure 5.3: Timed consistency handlers in the AQuA gateway

5.4 Tunable Consistency Protocols

We now describe how we maintain consistency across the replicas belonging to the two-tier organization described in the previous section. As mentioned in Section 5.2, in order to maintain replica consistency, we need to ensure that the replicas service their clients by respecting the ordering guarantee associated with the service. Our framework allows different ordering guarantees to be implemented as *timed consistency handlers* within the AQuA gateway, as shown in Figure 5.3. We have implemented gateway handlers that provide sequential and FIFO ordering. A client can communicate with a replicated service by using the gateway handler appropriate for the service. For example, Figure 5.3 shows a client communicating with two different services. Service A represents an application, such as a document-editing application, that guarantees sequential consistency using total ordering. Service B represents an application, such as a banking transaction, that guarantees FIFO ordering. The client uses the timed sequential consistency handler to communicate with service A, while it uses the timed FIFO handler to communicate with service B. The protocol processing in the handler is divided into a client-side protocol and a server-side protocol. In this section, we will describe the processing involved on the server side in order to maintain the appropriate ordering guarantee among the hierarchically organized replicas, and in the next section, we will describe how the client-side protocol uses these replicas to meet the client's QoS specification.

5.4.1 Sequential Ordering Protocol

In our sequential consistency model, the update requests of the clients are executed by all the primary replicas in the same order. The secondary replicas do not directly service a client's update request. Instead, the secondary replicas update their state when one of the members of the primary group lazily propagates its updated state to the secondary group. We call that member the *lazy publisher*. Thus, although the replicas may update their state at different points in time, they all see the effects of the updates in the same sequential order. The order in which an update is committed by the replicas is determined by its *Global Sequence Number (GSN)*, which is assigned by the leader of the primary group. Thus, the leader serves as the *sequencer*. Although we initially allowed the sequencer to service the requests from the clients, we found that it resulted in high variability in the response times. Hence, we decided to use a dedicated sequencer that does not actually service the client's request, but merely passes out the global sequencer number for each request. Henceforth, we will use the terms "leader" and "sequencer" synonymously. The value of the GSN at any instant of time may be considered to be the value of the leader's logical clock, and it also serves as the basis for determining the staleness of a replica. In a strict sequential consistency model, all requests, regardless of whether they retrieve or update the state, have to be executed in the same order by all the replicas. In our sequential consistency model, however, the read-only invocations of the clients need not be executed by all the replicas in the same order. In fact, different replicas may service different sets of read-only requests. The only constraint imposed on executing a read-only invocation on an object is that the state of the replicated object should be no more stale than the staleness threshold specified by the client. This is done to ensure that a read operation never returns data that is more than x versions older than the most recently updated version of the object, where x is the staleness threshold specified by the client in its QoS specification.

We will now describe how the updates and read-only requests are processed by the replicas. The processing depends on whether the replica is in the primary or secondary group. All of the processing is done at the middleware layer, within the gateway handler of the replicas. Each gateway handler maintains the following state in order to provide sequential ordering guarantees.

`my_GSN:` a replica's local view of the current Global Sequence Number (GSN). The GSN dictates the

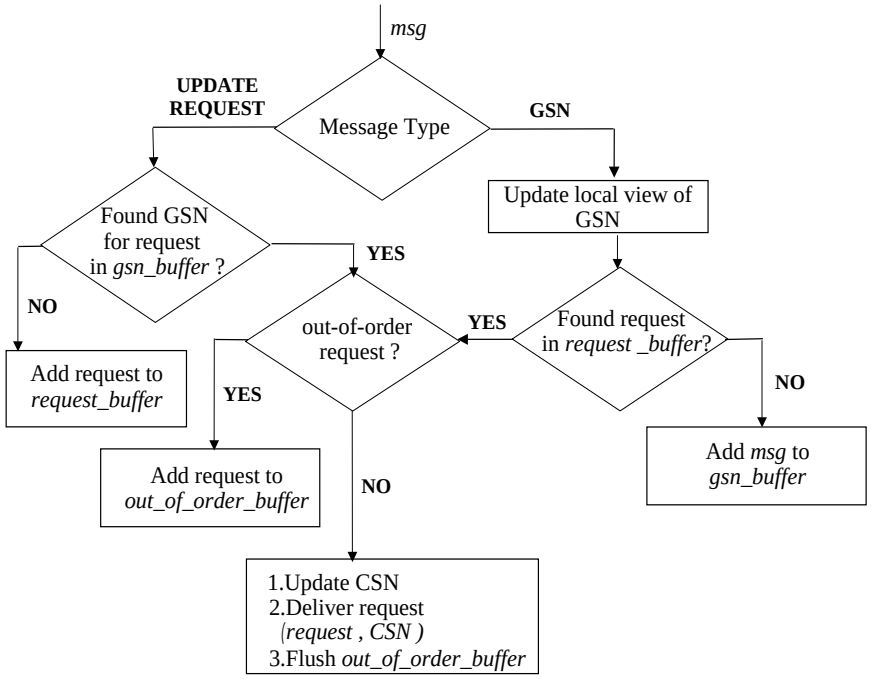


Figure 5.4: Processing of updates in sequential order by primary group members

order in which the requests are processed by the replicas in order to provide timed sequential consistency guarantees.

`my_CSN:` highest commit sequence number (CSN) of a replica, which indicates the GSN of the most recent update committed by the replica. The commit sequence number increases strictly in monotonic order, and the replica is assumed to have committed all those updates whose global sequence number is less than or equal to `my_CSN`. For any replica, `my_CSN` $\leq$ `my_GSN`, and the staleness of the replica's state is given by (`my_GSN` - `my_CSN`).

`request_buffer:` stores requests received from the clients that are awaiting GSN assignments.

`gsn_buffer:` stores the mapping between the requests and their corresponding GSNs.

`out_of_order_buffer:` stores the requests that have arrived out of the global order.

`deferred_read_buffer:` stores those read-only requests whose responses have to be deferred because the replica's state is not sufficiently current.

Update Operations

The update operations are sent to all members of the primary group, including the sequencer. When the sequencer receives an update request from a client, it advances the GSN and broadcasts the GSN assignment for the request to all the other members of the primary group. The primary replicas commit the updates in the order of the GSN.

Figure 5.4 shows how a non-leader primary replica services an update request from a client. A non-leader primary replica can service an update request immediately, provided it has already received the GSN broadcast for that request from the sequencer. That can happen if the transmission delays along different paths are significantly different. If, however, the replica has not yet received the GSN for the request, the replica stores the request in its `request_buffer` and processes it upon receiving the GSN assignment from the sequencer. Before processing the update request, the replica ensures that the update is committed in sequential order by first verifying that the current value of its CSN is one less than the GSN of the update request. If it is, the replica advances its CSN, and then delivers the update request to the server application. If, however, the request is out of the global order, the replica buffers the request in the `out_of_order_buffer` and commits it at a later time, after the intermediate requests have been committed.

Read-Only Operations

A read-only request is sent to the sequencer and a subset of the primary and secondary replicas. When the sequencer receives a read-only request, the leader broadcasts the current value of the GSN to the primary and secondary replicas, without advancing the GSN. Figure 5.5 shows how a non-leader primary replica or a replica in the secondary group services a read-only request from a client. When a non-leader primary or a secondary replica receives a read-only request from a client, it buffers the request until it receives the GSN assignment for the request from the sequencer. The replicas use the GSN to measure the staleness of their state. To determine its staleness, the replica first sets its local view of the global sequence number (which is given by the variable `my_GSN`) to the value of the GSN broadcast by the sequencer. The replica then computes the value of (`my_GSN - my_CSN`), where `my_CSN` is the

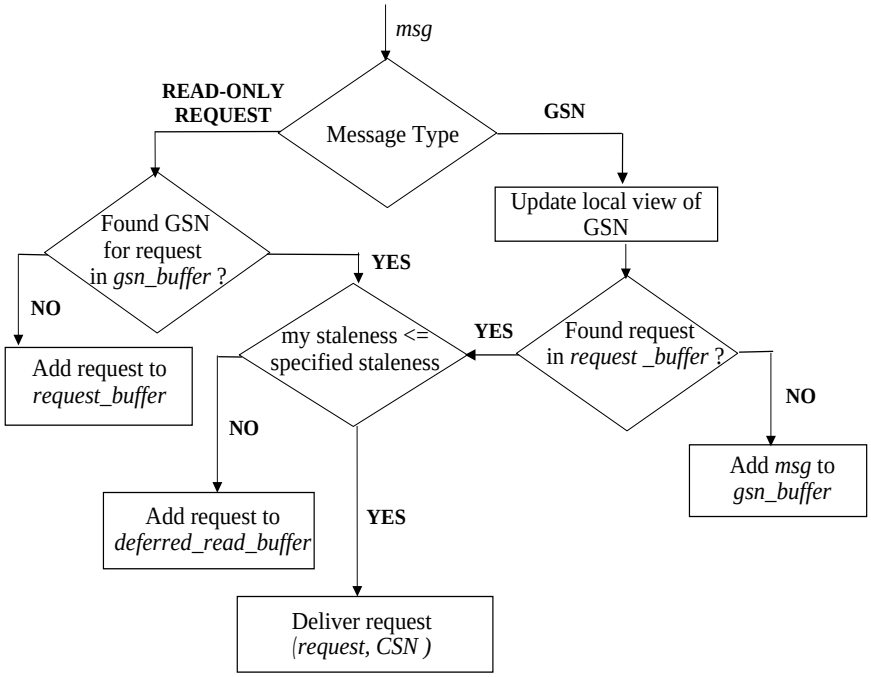


Figure 5.5: Processing of a read-only request

commit sequence number of the most recent update committed by the replica. The computed value is a measure of how stale the state of the replica is. If the replica's state is within the staleness threshold specified by the client in its QoS specification, the replica can service the client's request immediately. However, a secondary replica may have state that is more stale than the specified staleness threshold, because the secondary replicas update their state only when they receive the state update from the lazy publisher. In that case, the replica buffers the read request in the `deferred_read_buffer` and responds to the client immediately after receiving the next state update from the lazy publisher.

Replica Failures

One of the goals of AQuA is to provide fault tolerance. Hence, we need to ensure that the consistency guarantees are preserved even when replica failures occur. In our work, we assume that replicas fail by crashing. Since both the leader of the primary group and the lazy publisher play a crucial role in providing sequential consistency semantics, our algorithm handles their failures to ensure that the consistency guarantees are not violated.

The Maestro-Ensemble group communication protocol greatly simplifies our handling of replica

failures. First, if any of the group members fails, Maestro-Ensemble notifies the remaining group members about the failure. Further, if the leader of a group fails, Ensemble elects a new leader and notifies the other group members about the election. By virtue of this election protocol, we can guarantee that when the current sequencer, which in our case is the leader of the primary group, fails, then another member of the primary group will be elected the sequencer. The new sequencer first checks if the lazy publisher is still alive. If the lazy publisher has crashed, the sequencer designates one of the surviving members of the primary group as the new lazy publisher. The sequencer then notifies all the clients that it is the new sequencer. That is necessary because the clients need to send their requests to the sequencer. The sequencer also has the responsibility of preserving the sequential ordering guarantees during the transition. Since we depend on Maestro-Ensemble to provide virtual synchrony and reliability, we assume that all the replicas would have received the messages sent in the old view. Hence, to ensure sequential ordering, the new sequencer begins by assigning the GSN to the pending requests, if any, before assigning the GSN for the newly incoming requests.

5.4.2 FIFO Ordering Protocol

In order to compare the tradeoffs between timeliness and consistency, we also implemented FIFO ordering, which provides weaker consistency semantics than sequential ordering. While the sequential handler was motivated by applications, such as document-sharing applications, in which all of the clients access a shared replicated state, the FIFO handler was motivated by applications, such as banking transactions, in which the servers maintain states that are specific to each client. Like the sequential handler, the FIFO handler also uses the secondary group of replicas only for servicing the read requests of the clients. The primary replicas process the update requests of the clients, and the lazy publisher propagates its state lazily to the secondary group. However, the FIFO handler differs from the sequential handler with respect to the order in which the replicas commit their updates and the manner in which a replica determines if its state meets the staleness threshold specified by a client. Unlike the sequential handler, the FIFO handler does not use a dedicated sequencer to determine the order in which the replicas commit their updates. Instead, the clients send a sequence number with each of their in-

vocations. The primary replicas commit the updates of each client in increasing order of this sequence number. In the case of a read request, the replicas use this client-specific sequence number to determine if their state, with respect to a client's updates, is within the client-specified staleness threshold. They then perform an immediate or deferred read accordingly.

The two handlers also differ in the overheads incurred during the lazy update propagation. In the sequential handler, each replica's commit sequence number (defined as `my_CSN` in Section 5.4.1) summarizes the state of the replica. During the lazy update propagation, the lazy publisher propagates its commit sequence number, along with its updated state, to the secondary group. In the case of FIFO handler, since each replica maintains state specific to each client, the replicas have to associate a CSN with each client. Hence, in the FIFO handler, each replica maintains a state vector, which is a vector of $< client_i, CSN_i >$ pairs. This state vector summarizes the state of the replica. During the lazy update propagation, the lazy publisher propagates its state vector, along with its updated state, to the secondary group. Since the size of the state vector increases with the number of clients, we can expect the overhead of the lazy update propagation to be higher for FIFO ordering than for its sequential counterpart. However, it may be possible to reduce these overheads by propagating the modified vector incrementally, rather than propagating the entire vector during each lazy update.

Failure handling is another point in which the FIFO handler differs from the sequential handler. Since the FIFO handler does not use a dedicated sequencer, it only has to handle the failure of the lazy publisher. In the FIFO handler, the leader of the primary group is designated as the lazy publisher. As stated earlier, this leader is always elected by Ensemble. When the new leader is elected upon the failure of the old leader, it propagates its state to the secondary replicas, schedules the subsequent lazy updates with the appropriate frequency, and then continues to service the requests from the clients.

5.5 Selection of Replicas with Dynamic State

Having described the processing involved in the gateway handler on the server side, we now describe the processing done on the client side in order to meet the QoS specification of the client. One of

the important goals of this work is to make use of replication to meet the QoS requirements of the client applications. As mentioned in Section 5.2, our work targets clients that have specific consistency and timeliness constraints. Each client expresses its constraints in the form of a QoS specification that includes the response time constraint, d; the minimum probability of meeting this constraint, $P_c(d)$; and the maximum staleness, a, that it can tolerate in its response. If a response fails to meet the deadline constraint of the client, then it results in a timing failure for the client. Hence, we need to find a way to select replicas that can deliver a timely and consistent response to the clients, thereby reducing the occurrence of timing failures. One of the important responsibilities of the client handler is to select an appropriate subset of replicas from the two-tier replica hierarchy, to service its client's request.

A simple approach would be to allocate all the available replicas to service a single client, but that approach would not be scalable, as it would increase the load on all the replicas and result in higher response times for the remaining clients. On the other hand, assigning a single replica to service each client would allow us to service multiple clients concurrently, but if a replica should fail while servicing a request, the failure could result in an unacceptable delay for the client being serviced. Hence, neither approach is suitable when a client has specific timing constraints and when failure to meet the constraints results in a penalty for the client. In order to build a dependable service, we need a method that attempts to prevent the occurrence of such timing failures for a client by selecting replicas from the available replica pool, based on an understanding of the client's QoS requirements and the responsiveness and state of the replicas.

As mentioned earlier, in our model, the constraints specified by a client apply only for the read transactions invoked by the client. For an update transaction, the only constraint that applies is that it has to be committed by the replicas in a manner that respects the ordering guarantee associated with the service. Hence, our selection algorithm handles an update request of a client by simply multicasting the request to all the primary replicas. The handler on the server side takes care of committing these updates in the appropriate order, as described in Section 5.4. For the read-only requests, the selection algorithm has to choose from among the primary and secondary replicas based on their ability to meet the client's temporal requirements, as well as on whether the state of the replica is within the staleness

threshold specified by the client. However, the uncertainty in the environment and in the availability of the replicas due to transient overload and failures makes it impossible for a client to know with certainty if a set of replicas can meet its deadline. Further, while a client can be certain that the state of the primary replicas is always up-to-date, because all of the clients propagate their updates directly to them, the client cannot be certain about the state of the secondary replicas. The reason is that the secondary replicas update their state only when they receive the lazy updates propagated by the lazy publisher.

Hence, our selection approach makes use of probabilistic models to estimate a replica's staleness and to predict the probability that the replica will be able to meet the client's deadline. These models make their prediction based on information gathered by monitoring the replicas at runtime. A selection algorithm then uses this online prediction to choose a subset of replicas that can together meet the client's timing constraints with at least the probability requested by the client. While the algorithm ensures that the response delivered to the client will meet the staleness constraint, it can only provide probabilistic guarantees about meeting the temporal constraint. We will now describe our probabilistic models and replica selection algorithm. They enhance the selection approach we presented in Chapter 3, which made the assumption that the replicas were stateless. We first define the notation required to understand our model.

- t : the time at which a request is transmitted. Since replicas are selected at the time a request is transmitted, we also use t to denote the time at which the replica selection is done.
- R_i : the random variable denoting the response time of a replica i.
- $A_i(t)$: the staleness of the state of replica i at time t.
- $P(A_i(t) \leq a)$: the probability that the state of replica i at time t is within the staleness threshold, a, specified by the client. We call this the *staleness factor* for replica i.
- $P(R_i \leq d)$: the probability that a response from replica i will be received by the client within the client's deadline, d.

- $P_K(d)$: the probability that at least one response from the set K, consisting of $k > 0$ replicas, will arrive by the client's deadline, d.

The probability that a replica can meet the client's time constraint, d, and thereby prevent a timing failure depends on whether the replica is functioning and has a state that can satisfy the client-specified staleness threshold. We can make use of the probabilities of the individual replicas to choose a subset K of replicas such that $P_K(d) \geq P_c(d)$. The replicas in the set K will then form the final set selected to service the request.

5.5.1 Modeling the Response Time Distribution

We will now derive the expression for $P_K(d)$, which is the probability that at least one response from the replicas in set K arrives by the client's deadline, d, and thereby avoids the occurrence of a timing failure. The set K is made up of a subset K_p of primary replicas and a subset K_s of secondary replicas (i.e., $K = K_p \cup K_s$). While each replica in K processes the client's request and returns its response, only the first response received for a request is delivered to the client. Hence, a timing failure occurs only if no response is received from any of the replicas in the selected set K within d time units after the request was transmitted. Therefore, we have

$$P_K(d) = 1 - P(\text{no replica } i \in K \ni R_i \leq d)$$

In our work, we assume that the response times of the replicas are independent because they process their requests independently. While this assumption may not be strictly true in some cases (e.g., if the network delays are correlated), it does result in a model that is fast enough to solve online, which is especially helpful for the time-sensitive applications we target in our work. Furthermore, the results we present in Chapter 6 show that the resulting model makes reasonably good predictions most of the time. Thus, using the independence assumption, we obtain

$$P_K(d) = 1 - P(\text{no } i \in K_p \ni R_i \leq d) \cdot P(\text{no } j \in K_s \ni R_j \leq d) \tag{5.1}$$

Case 1: Primary Replicas

In Section 5.4, we mentioned that the update requests of the clients are propagated to the primary group immediately. Hence, for a primary replica i, the staleness factor $P(A_i(t) \leq a) = 1$, and the replica always has a state that can satisfy the staleness threshold of the client. Therefore, in the case of the primary replicas, we have

$$P(\text{no } i \in K_p \ni R_i \leq d) = \prod_{i \in K_p} P(R_i > d) = \prod_{i \in K_p} (1 - F^I_{R_i}(d)) \tag{5.2}$$

where $F^I_{R_i}$ denotes the response time distribution function for replica i, given that it can respond immediately to a read request without waiting for a state update.

Case 2: Secondary Replicas

The response time of a secondary replica depends on whether it has a state that can satisfy the client specified staleness threshold, a. If the replica's staleness is within the specified staleness threshold, then the replica can perform an immediate read. Otherwise, as mentioned in Section 5.4.1, the replica has to buffer the request until it receives the next lazy update, and perform a deferred read. At the time of replica selection, the client gateway that selects the replicas does not know for certain how stale the secondary replicas are. Hence, it has to estimate the staleness of the secondary replicas. The client gateway uses a probabilistic approach for this estimation. The probabilistic approach allows us to express the responsiveness of a replica $j \in K_s$ as a conditional probability using the following equation:

$$P(R_j > d) = P(R_j > d | A_j(t) \leq a) \cdot P(A_j(t) \leq a) + P(R_j > d | A_j(t) > a) \cdot P(A_j(t) > a)$$

where $P(A_j(t) \leq a)$ is the staleness factor of replica j, as defined earlier. Since the lazy update is propagated to all the secondary replicas at the same time, it is reasonable to assume that their degrees of staleness at the time of request transmission, t, are identical. Hence, rather than associate staleness

with an individual replica j as above, we associate staleness with the entire secondary group of replicas. We use $A_s(t)$ to denote the staleness of the secondary group at the time of request transmission t, and express the probability that no secondary replica can respond within the deadline d as follows.

$$P(\text{no } j \in K_s \ni R_j \leq d) = \left[\prod_{j \in K_s} P(R_j > d | A_s(t) \leq a)\right] \cdot P(A_s(t) \leq a) + \left[\prod_{j \in K_s} P(R_j > d | A_s(t) > a)\right] \cdot P(A_s(t) > a)$$

$$P(\text{no } j \in K_s \ni R_j \leq d) = \left[\prod_{j \in K_s} (1 - F^I_{R_j}(d))\right] \cdot P(A_s(t) \leq a) + \left[\prod_{j \in K_s} (1 - F^D_{R_j}(d))\right] \cdot (1 - P(A_s(t) \leq a)) \tag{5.3}$$

where $F^I_{R_j}$, as before, denotes the response time distribution function for the replica j, given that j can respond immediately to a request without waiting for a state update, and $F^D_{R_j}$ is the response time distribution function, given that the replica defers the read until it has received the next lazy state update.

In a real system, a replica that is alive at time t may crash before responding to the request. In deriving the above expressions to estimate the responsiveness of the primary and secondary replicas, we have implicitly assumed that the probability of such a crash is small. In other words, we have assumed that $P(alive_i(t+d)|alive_i(t)) \approx 1$. For systems in which that is not true, we can extend the above model to derive the response time distribution conditional to the availability of a replica. A replica is available to service a client's request if the replica remains functional until it has serviced the request and has state that meets the client's staleness threshold. Using that definition, we can compute the availability of a replica i using Equation 5.4. We can then use this equation to compute the conditional response time distribution function of a replica i, as we did above.

$$P(Avail_i(t)) = P(alive_i(t+d)|alive_i(t)) \cdot P(A_i(t) \leq a) \tag{5.4}$$

We now describe how we compute the staleness factor, $P(A_s(t) \leq a)$, for the secondary replicas, and then follow that with a description of how we compute the values of the response time distribution functions $F^I_{R_i}$ and $F^D_{R_i}$ for a replica i.

Staleness Factor

The staleness of a secondary replica, at the instant t, is the number of update requests that have been received by the primary group since the time of the last lazy update. Let t_l denote the duration elapsed between the time of request transmission, t, and the time of the last lazy update. Let $N_u(t_l)$ be the total number of update requests received by the primary group from all the clients in the duration t_l. Since $A_s(t) = N_u(t_l)$, we have $P(A_s(t) \leq a) = P(N_u(t_l) \leq a)$. Our approach estimates the staleness of the secondary replicas based on a probabilistic model, rather than using the prohibitively costlier method of probing the primary group at the time of request transmission in order to obtain the value of $N_u(t_l)$. Using the assumption that the arrival of update requests from the clients follows a Poisson distribution with rate λ_u, we obtain

$$P(A_s(t) \leq a) = P(N_u(t_l) \leq a) = \sum_{n=0}^{a} \frac{(\lambda_u t_l)^n e^{-\lambda_u t_l}}{n!} \tag{5.5}$$

Therefore, the staleness of a secondary replica can be determined probabilistically if we know the arrival rate of the update requests and the time elapsed since the last lazy update. In Section 5.5.4, we will explain how we measure those two parameters at runtime. Although we have assumed Poisson arrivals in our work, it should be possible to evaluate the staleness factor when the arrival of update requests follows a distribution that is not Poisson. Finally, we can use the expressions in Equations 5.2, 5.3, and 5.5 in Equation 5.1 to evaluate the probability $P_K(d)$ that at least one of the replicas in the selected set K can deliver a timely and consistent response.

5.5.2 Evaluating the Response Time Distribution Function

We now explain how we determine the values of the conditional response time distributions, $F^I_{R_i}(d)$ and $F^D_{R_i}(d)$, for a replica i. To do this, we extend the method we described in Section 3.5 for the stateless case, which made use of the performance history recorded by online performance monitoring to compute the value of the distribution function for a replica i.

Immediate Reads

When a replica can respond to a request without waiting for a state update, as in the case of a primary replica or a secondary replica whose state is within the specified staleness threshold, the response time random variable for a replica i is given by Equation 5.6:

$$R_i = S_i + W_i + G_i \tag{5.6}$$

where S_i is the random variable denoting the service time for a read request serviced by replica i; W_i is the random variable denoting the queuing delay experienced by a request waiting to be serviced by i; and G_i is the random variable denoting the two-way gateway-to-gateway delay between the client and replica i. In the case of sequential ordering, the queuing delay includes the time the replica spends waiting for the sequencer to send the GSN for the request. The service time and queuing delay are specific to the individual replicas, while the gateway delay is specific to a client-replica pair.

Deferred Reads

If the replica has to buffer the read request until it has received the next state update in order to respond to the request, the response time random variable is given by Equation 5.7, where S_i, W_i, and G_i are as defined above, and U_i is the duration of time the replica spends waiting for the next lazy update.

$$R_i = S_i + W_i + G_i + U_i \tag{5.7}$$

For each request, we experimentally measure the values of the above performance parameters as described in Section 5.5.4. The client handlers record the most recent l measurements of these parameters in separate sliding windows in their local information repositories. The size of the sliding window, l, is chosen so as to include a reasonable number of recent requests, while eliminating obsolete measurements. To evaluate $F^I_{R_i}(d)$, we first compute the probability mass function (pmf) of S_i and W_i based on the relative frequency of their values recorded in the sliding window. We then use the pmf of S_i, the pmf of W_i, and the most recently recorded value of G_i to compute the pmf of the response time R_i as a discrete convolution of S_i, W_i, and G_i. For G_i, unlike the other parameters, we use its most recently recorded value instead of its history recorded over a period of time, because the gateway delay does not fluctuate as much as the other parameters do. The pmf of R_i can then be used to compute the value of the distribution function $F^I_{R_i}(d)$. We follow a similar procedure to compute $F^D_{R_i}(d)$, although in this case we record a performance history of U_i and include the pmf of U_i in the convolution.

5.5.3 State-Based Replica Selection Algorithm

Algorithm 2 outlines the selection algorithm that enables a client gateway to select a set of replicas that can together meet the client's QoS specification, based on the prediction made by the probabilistic models described in the previous section. The algorithm uses the model's prediction to select no more than the number of replicas necessary to meet the client's response time constraint with the probability the client has requested. This algorithm is executed in a distributed manner by a client gateway when the client associated with it performs a read-only request on a server object. If the client makes an update request, the gateway simply sends the request to all the primary replicas.

In order to maximize the concurrent accesses to a service, it would be ideal if we could use the model's prediction to choose the smallest subset of replicas to service a client. In Chapter 3, we presented a greedy algorithm to do this for stateless replicas. Using that strategy, each client first sorted the replicas in decreasing order of the probability that the replica can individually meet the client's deadline. The clients then chose the replica subset by traversing the replica list in sorted order. The problem with this greedy approach is that it may result in "hot-spots," for the following reason. As

Algorithm 2 Replica selection algorithm: dynamic state

Require: $V =< i, F^I_{R_i}(d), F^D_{R_i}(d), etr_i >$, staleFactor
Require: Client Inputs: a : staleness threshold, d : deadline, $P_c(d)$: minimum probability of meeting the deadline
1: primCDF $\Leftarrow$ 1; secCDF $\Leftarrow$ 1; secImmedCDF $\Leftarrow$ 1; secDelayedCDF $\Leftarrow$ 1
2: sortedList $\Leftarrow$ sort V in decreasing order of etr_i.
3: K $\Leftarrow$ [first(sortedList)] ; maxCDFReplica $\Leftarrow$ [first(sortedList)] ; advance(sortedList)
4: **for all** i in sortedList **do** {visit the remaining replicas in sorted order}
5: $K \Leftarrow K \cup i$
6: **if** $F^I_{R_i}(d) >$ maxCDFReplica.immedCDF() **then**
7: found $\Leftarrow$ includeCDF(maxCDFReplica, maxCDFReplica.immedCDF(), maxCDFReplica.delayedCDF())
8: maxCDFReplica $\Leftarrow i$
9: **else**
10: found $\Leftarrow$ includeCDF($i, F^I_{R_i}(d), F^D_{R_i}(d)$)
11: **end if**
12: **if** found eq true **then** {found an acceptable set}
13: return K
14: **end if**
15: **end for**
16: return K {return the set comprising all the replicas if no acceptable subset is found}
17: **includeCDF(replica, immedCDF, delayedCDF)**
18: **begin**
19: **if** replica $\in$ PrimaryGroup **then**
20: primCDF $\Leftarrow$ primCDF * (1 - immedCDF)
21: **else**
22: secImmedCDF $\Leftarrow$ secImmedCDF * (1 - immedCDF); secDelayedCDF $\Leftarrow$ secDelayedCDF * (1 - delayedCDF)
23: secCDF $\Leftarrow$ secImmedCDF * staleFactor + secDelayedCDF * (1 - staleFactor)
24: **end if**
25: **if** 1 - (primCDF * secCDF) $\geq P_c(d)$ **then**
26: return true {found an acceptable replica set}
27: **else**
28: return false {need more replicas}
29: **end if**
30: **end**

mentioned in the previous section, the model used by the algorithm makes use of the performance information broadcast by a replica to estimate the replica's ability to meet a client's QoS specification. Since the information repositories of the different clients may contain almost identical performance histories for the replicas, that may cause the clients to select the same or common replicas. Hence, we have designed Algorithm 2 to select the replica subset in such a way that it alleviates the occurrence of such hot-spots, to achieve a more balanced utilization of all the available replicas. It does so by using information that is specific to a client-replica pair, in addition to the replica-specific performance information. While the latter information is nearly identical in the different client repositories, the former information is different. Unlike the algorithm in Chapter 3, however, Algorithm 2 does not always guarantee that the smallest replica subset will be chosen to service a request.

The algorithm receives as input the QoS specification of the client, and the list of secondary and primary replicas along with relevant information about them. For each replica i, the algorithm receives the values of its immediate and delayed response time distribution functions, which are denoted by $F^I_{R_i}(d)$ and $F^D_{R_i}(d)$. For a primary replica i, $F^D_{R_i}(d)$ is not used. The algorithm also receives the elapsed time of response, etr_i, which is the duration that has elapsed since a reply was last received by the client from replica i. The response time distribution, which is computed from the performance history as explained in Section 5.5.2, is specific to the individual replica and is nearly identical in all the client information repositories. However, the etr information is not the same in all the repositories, as it is specific to each client-replica pair. In addition, the algorithm also receives the staleness factor for the secondary replicas, which is computed using Equation 5.5. The algorithm first sorts the replicas in decreasing order of their elapsed time of response, etr. That allows the clients to favor the selection of replicas that they used least recently and thereby obviate the hot-spot problem mentioned above. Replicas that have the same value of etr are sorted in decreasing order of the values of their distribution functions.

The algorithm traverses the replica list in sorted order, including each visited replica in the candidate set K, until it includes enough replicas in K to satisfy the terminating condition $P_K(d) \geq P_c(d)$. Each time it includes a new replica, the algorithm invokes the function `includeCDF()` to obtain the value

of $P_K(d)$ for the replicas that are currently included in the set K. This function receives as arguments the values of $F^I_{R_i}(d)$ and $F^D_{R_i}(d)$ for the newly included replica, and uses them to compute the value of $P_K(d)$ according to Equation 5.1. The function then tests the terminating condition in Line 25 and returns true if the condition is satisfied, indicating that an appropriate replica subset has been found. In the case of sequential ordering, the selected set K is extended to include the sequencer, which is responsible for broadcasting the global sequence number. If a service is new and no performance history has been recorded for any of its replicas, the handler initially chooses all of the available replicas offering the service.

Notice that when evaluating $P_K(d)$, we exclude the response time distribution of the member (denoted by `maxCDFReplica` in Algorithm 2) that has the highest probability of responding by the requested deadline among the selected members. We do this in order to choose a subset that can meet a client's time constraint despite a single replica failure. We propose that if we can choose a set of replicas that can satisfy the timing constraint with the specified probability despite the failure of the member that has the highest probability of meeting the client's deadline, then such a set should be able to handle the failure of any other member in the set. In Section 3.5.2, we provided a formal justification for this proposal. The above exclusion in effect simulates the failure of the replica with the highest probability of meeting the client's deadline among the selected replicas, and therefore allows us to tolerate single replica failures. Although Algorithm 2 addresses only single replica crashes, it should be possible to extend it to handle multiple replica crashes. Finally, we account for the overhead of the algorithm during each selection, as explained in Section 3.5.3 for the static state.

5.5.4 Online Performance Monitoring

We now explain some of the key implementation details of how the client and server gateway protocols interact to measure and record the different performance parameters, which are used to compute the response time distribution function and staleness factor. We monitor the performance at runtime by instrumenting the gateway handlers, through a procedure similar to that described in Section 3.6.1 for the static state. However, there are some additional parameters that need to be measured in the case of

dynamic state, as we now explain. When a client makes a request to a service, the client-side handler transparently intercepts the request and records the interception time, t_0. The handler makes use of the performance history recorded in its local information repository to select a set of replicas based on the client's QoS specification, as explained in Section 5.5.3. The handler then multicasts the request to the selected set of replicas through the Maestro-Ensemble group communication layer.

Upon receiving the request from the client, the server-side gateway handler delivers the read or update request to the server application, according to the appropriate ordering protocol, as described in Section 5.4. We instrumented the gateway handler so that it can record the service time, t_s, and queuing time, t_q, for the request. t_q also includes the time spent waiting to receive the GSN for the request from the sequencer. In addition, if a replica performs a deferred read, the gateway handler records the duration of time, t_b, for which the request was buffered while waiting for the next lazy update. When the server sends its response to the client, the server handler intercepts the response and piggybacks $t_1 = t_s + t_q + t_b$ in the response message. Each server handler also publishes the newly measured values of t_s, t_q, and t_b to all of its clients whenever it completes servicing a read request. The performance updates published by the server replicas is used by the client to update its gateway information repository.

When the client handler receives a reply from a replica, it records the time of reception, t_p, in its information repository. The time of reception is used by the client to compute the elapsed time of response for the replica, when it executes Algorithm 2 to sort the replicas for its next read request. The client uses the piggybacked information, t_1, to record the new value of the two-way gateway-to-gateway delay, t_g, between the client and the replica. This delay, t_g, is given by $t_g = t_p - t_m - t_1$, where t_m is the time at which the client handler transmitted the request to the selected set of replicas using Maestro-Ensemble.

If the reply is the first one it has received for a request, the client handler delivers the reply to the client. The timing failure detector in the client handler computes the response time, $t_r = t_p - t_0$, to check whether a timing failure has occurred. A timing failure occurs if $t_r > d$, where d is the response time requested by the client. The timing failure detector maintains a counter that keeps track of the

number of times the client has failed to receive a timely response from a service. If the frequency of timely response from the service is lower than the minimum probability of timely response the client has requested in its QoS specification, the client handler notifies the client by issuing a callback.

Staleness Measurement

We now explain how we monitor the information required to measure the staleness of the secondary replicas at runtime. From Equation 5.5, we infer that if a client gateway knows the arrival rate of the update requests (λ_u) and the duration elapsed since the last lazy update (t_l), then it can determine whether a secondary replica has a state that can meet the staleness threshold specified by the client. The sequential and FIFO handler differ in the way they measure the arrival rate of updates (λ_u). We first explain how λ_u and t_l are measured in the case of the sequential handler and then follow by describing how the measurement differs for the FIFO handler. To measure the values of the two parameters in the sequential handler, the server that is designated as the lazy publisher broadcasts the following additional information when it publishes its performance measurements to the clients. The information includes 1) $< n_u, t_u >$, where n_u is the number of update requests the lazy publisher has received from the clients in the duration t_u, which is the time elapsed since the publisher's last performance broadcast; and 2) $< n_L, t_L >$, where n_L is the number of update requests the lazy publisher has received from the clients in the duration t_L, which is the time elapsed since the lazy publisher propagated its last lazy update. The client handlers record in their information repositories the most recently published value of $< n_L, t_L >$ and a history of $< n_u, t_u >$ over a sliding window. The arrival rate is computed as $\lambda_u = \sum n_u^i / t_u^i$, where the sum is taken over the sliding window. At the time of request transmission t, the duration elapsed since the last lazy update is computed as $t_l = (t_L + t_z)$ modulo T_L, where T_L is the periodicity with which the lazy updates are propagated. t_z is the duration of time that has elapsed since the client received the most recent performance broadcast from the lazy publisher, relative to t.

While the sequential handler measures λ_u as the rate at which the primary replicas receive the updates from all the clients, the FIFO handler measures λ_u individually for each client, as the rate at which the client that is making the replica selection updates the replicated object. In other words, for

FIFO ordering, rather than use the pair of values $< n_u, t_u >$, published by the server side, each client independently keeps track of its own rate of updates and uses it in Equation 5.5 to measure the staleness, as explained above. The reason for the difference is that in sequential ordering, the clients modify a globally shared server state, whereas in FIFO ordering, each client's update requests modify only the portion of the server state that is specific to that client.

Summary

In this chapter we described an adaptive framework that we have developed for providing tunable consistency and timeliness in the context of a replicated service. The framework can be used to access information stored in the servers, based on the timeliness and consistency requirements of a client. The framework allows different application-specific consistency guarantees to be implemented as protocols that can be dynamically loaded within the middleware. The replicas in the framework are organized into a hierarchy, with each hierarchical layer implementing a different degree of consistency. Although we described a 2-layer hierarchy, the framework can be extended to a multi-layer hierarchical organization when the number of replicas is large. The framework uses probabilistic models to choose replicas dynamically, based on the degree of consistency and timeliness desired by the clients.

Chapter 6

Performance Evaluation of Tunable Consistency and Timeliness

No amount of experimentation can ever prove me right, a single experiment can prove me wrong.

- Albert Einstein

We conducted experiments to evaluate the performance of the framework we described in Chapter 5 that provides tunable consistency and timeliness. We also experimentally analyzed the tradeoffs between timeliness and consistency, using the sequential and FIFO ordering handlers we implemented in AQuA. In this chapter, we discuss the results we obtained. In Section 6.1, we present results that show the overhead of Algorithm 2, which uses a probabilistic approach to select replicas that have a time-varying replicated state. In Section 6.2, we evaluate the adaptability and effectiveness of the model-based selection in a crash-free scenario and during a single replica crash. Then, in Section 6.3, we experimentally analyze the cost/performance tradeoffs of lazy update propagation. Section 6.4 discusses the performance of the framework under higher workload intensities. In Section 6.5, we discuss the pros and cons of a hierarchical replica organization, and study the impact of the size of the primary group of replicas. In Section 6.6, we compare the performances of the probabilistic selection scheme and a round-robin scheme, in the case of dynamic replicated state. Finally, in Section 6.7, we discuss the performance of the state-based selection scheme using a time-varying workload. All of the experiments were conducted using an experimental setup composed of a set of uniprocessor Linux machines with processor speeds ranging from 300 MHz to 1 GHz. The machines were distributed over

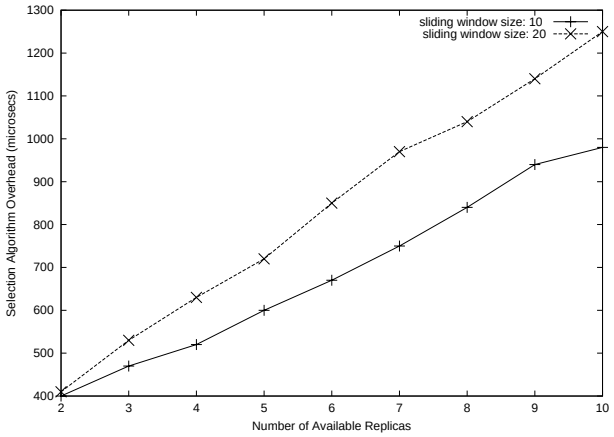


Figure 6.1: Overhead of selecting replicas with dynamic state

a 100 Mbps LAN. All confidence intervals for the results presented are at a 95% level, and have been computed under the assumption that the number of timing failures follows a binomial distribution [36].

6.1 Overhead of State-Based Selection Algorithm

The overhead of the selection algorithm in the case of dynamic state includes the time to compute the response time distribution function for the replicas and the time to select the replica subset. In addition, it includes the time to compute the staleness factor for the secondary group. Like the static state selection algorithm (Algorithm 1), the overhead of the dynamic state selection also depends on the size of the replica pool and the size of the sliding windows used to record the performance histories. Based on our experiments, we found that computation of the response time distribution function and the staleness factor contributes to 80% of the overheads, while selection of the replica subset using Algorithm 2 contributes to the remaining 20%.

Figure 6.1 shows how the overhead of the probabilistic selection algorithm in the case of dynamic state varies with the number of available replicas for sliding windows of sizes 10 and 20. These overheads were measured on a 1 GHz machine. If we compare Figure 6.1 with the overhead of the static selection scheme presented in Figure 3.3, we find that the overhead of selecting replicas with static state is lower. This is to be expected, because the computational overhead involved in evaluating Equation 3.1, which is the basis for the static selection algorithm, is lower than the overhead involved in

computing Equation 5.1, which is the model that guides the selection of replicas with dynamic state.

From Figure 6.1, we find that the overhead increases linearly with the size of the replica pool. However, sorting overhead can be improved by using a better algorithm to sort the replicas according to their elapsed response time. Currently, we use insertion sort. Further, the larger the sliding window, the greater the number of data samples used to compute the response time distribution and staleness factor, and therefore the higher the selection overhead. We used a sliding window of size 20 for the experiments we describe in the following sections.

6.2 Effectiveness of Probabilistic Model

One of the main objectives of Algorithm 2 is to assign the replicated servers to service a request adaptively, based on the current responsiveness of the replicas and the QoS requested by the clients. We now present the results of the experiments we conducted to evaluate the adaptivity and effectiveness of the probabilistic selection algorithm for sequential and FIFO ordering. Our experimental setup is summarized in Table 6.1. We used 10 server replicas, of which 4 were in the primary group, and 6 were in the secondary group. In addition, we had a sequencer when we were using sequential ordering. The service time for all the read and update requests was normally distributed with a mean of 100 milliseconds and a variance of 50 milliseconds. We used two clients that ran on two different machines and independently issued requests to the replicated service with a 1000 millisecond think time. In every run, each of the two clients issued 1000 alternating write and read requests to the service. This resulted in an update rate of about 1 update/second in the case of sequential ordering and 1 update every 2 seconds in the case of FIFO ordering. Recall that the client update rate in the case of sequential ordering includes the update requests of both the clients.

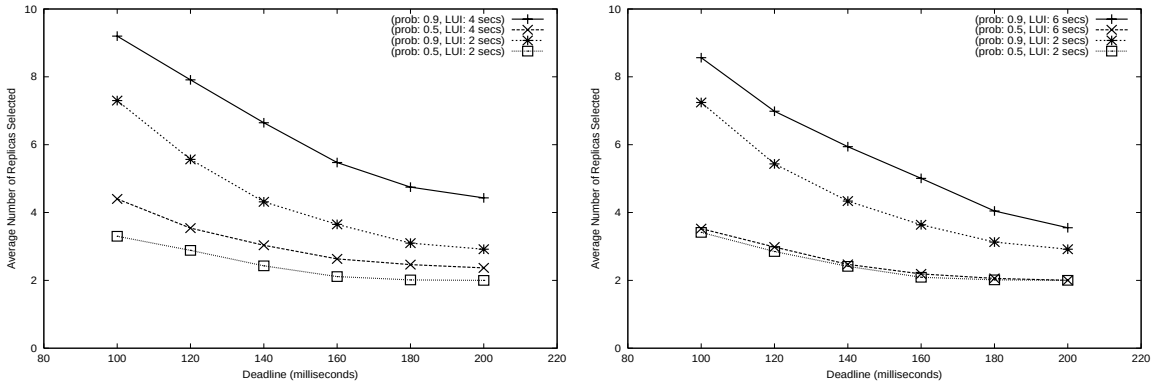
One of the clients (which we refer to as Client1) requested the same QoS for all of the runs; it specified a staleness threshold of value 4, a deadline of 200 milliseconds, and a minimum probability of timely response of 0.1. The second client (which we refer to as Client2) specified a staleness threshold of value 2 in all the runs, but requested a different deadline in each run. To study the behavior of the

Table 6.1: Experimental setup: effectiveness of probabilistic model

Total number of servers	10
Primary/secondary group size	4/6
Number of read requests	1000
Number of update requests	1000
Think time between requests	1000 ms
Service time for update requests	Normal(100 ms, 50 ms)
Service time for read requests	Normal(100 ms, 50 ms)
Number of clients	2
QoS of Client1	(a = 4, d = 200 ms, P(d) = 0.1)
QoS of Client2	(a = 2, d = var, P(d) = var)
Ordering guarantee	sequential, FIFO

selection algorithm for different values of the probability of timely response specified by a client, we repeated the experiments for two different probability values specified by Client2 in its QoS specification: 0.9 and 0.5. In order to study the effect of the staleness of the replicas on the timeliness of their response, we conducted experiments using different values for the lazy update interval (LUI). The LUI is the periodicity with which the lazy publisher propagates its updates to the secondary replicas, and was denoted as T_L in Section 5.5.4. For sequential ordering, we carried out experiments using lazy update intervals of 2 seconds and 4 seconds. In the case of FIFO ordering, we observed nearly identical performances for our experiments using LUI of 2 seconds and 4 seconds. Therefore, in the case of FIFO ordering, we have shown the results for LUI values of 2 seconds and 6 seconds.

We evaluated the adaptivity of our algorithm by determining how the degree of redundancy chosen to service a request varies with the requested QoS. In our experiments we did this by measuring how the average number of replicas chosen by the selection algorithm to service Client2 varied, as the client varied its QoS specification. Figure 6.2a shows the results for sequential ordering and Figure 6.2b shows the results obtained for FIFO ordering. From these graphs we observe that, regardless of the ordering, the number of selected replicas decreases as the requested deadline varies from 100 milliseconds to 200 milliseconds. Similarly, the number of selected replicas decreases as the requested probability of timely response reduces from 0.9 to 0.5. Another observation from these two figures is that for the same QoS specification and the same LUI values, the number of replicas selected in the case



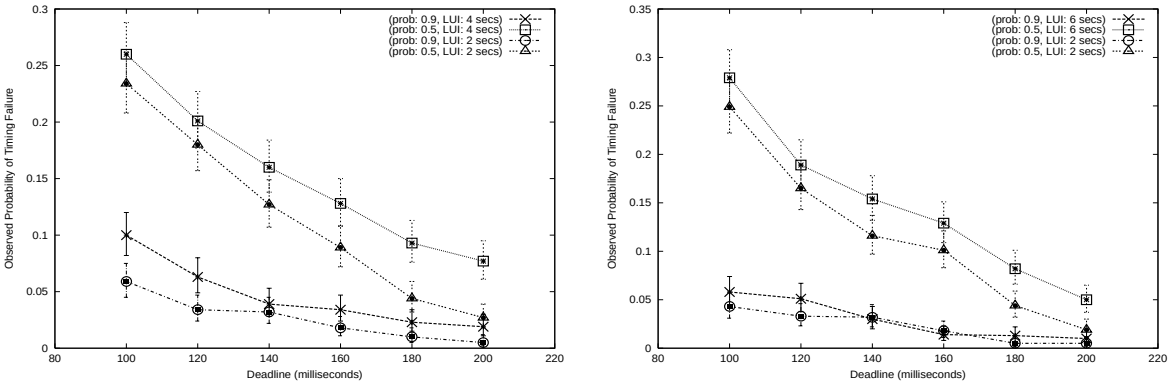
(a) Number of replicas selected: sequential

(b) Number of replicas selected: FIFO

Figure 6.2: Sequential vs. FIFO: adaptivity of probabilistic model

of sequential ordering is higher than in the case of the weaker, FIFO ordering. All of these observations lead us to conclude that as the timeliness and consistency requirements become less stringent, Algorithm 2 is able to adapt by choosing fewer replicas to meet the client's QoS requirement. As mentioned in Section 5.5.3, the algorithm uses the model's prediction to select just enough replicas that can meet a client's QoS request, despite a single replica failure. The less stringent a client's QoS specification is, the higher the probability that a chosen replica will meet the client's specification. Hence, as the QoS requirement becomes less stringent, the algorithm can satisfy the request with fewer replicas.

We also experimentally evaluated the effectiveness of our probabilistic model by determining if the selected replicas were indeed able to meet the QoS specification of the clients. Our selection algorithm guarantees that a client will always receive a response that is within the specified staleness threshold, while it provides only probabilistic guarantees about meeting the temporal requirement. In our experiments, we used the observed probability of timing failures as the metric to assess the effectiveness of our model. For each of the QoS specifications of Client2, we experimentally measured the probability of timing failures in a run by monitoring the number of requests in the run for which the client failed to receive a response within the requested deadline. This monitoring was done by the *timing failure detector*, which is a component of the client's gateway handler. Figure 6.3 shows how successful the selected replicas (shown in Figure 6.2) were in meeting the QoS specifications of



(a) Probability of timing failure: sequential

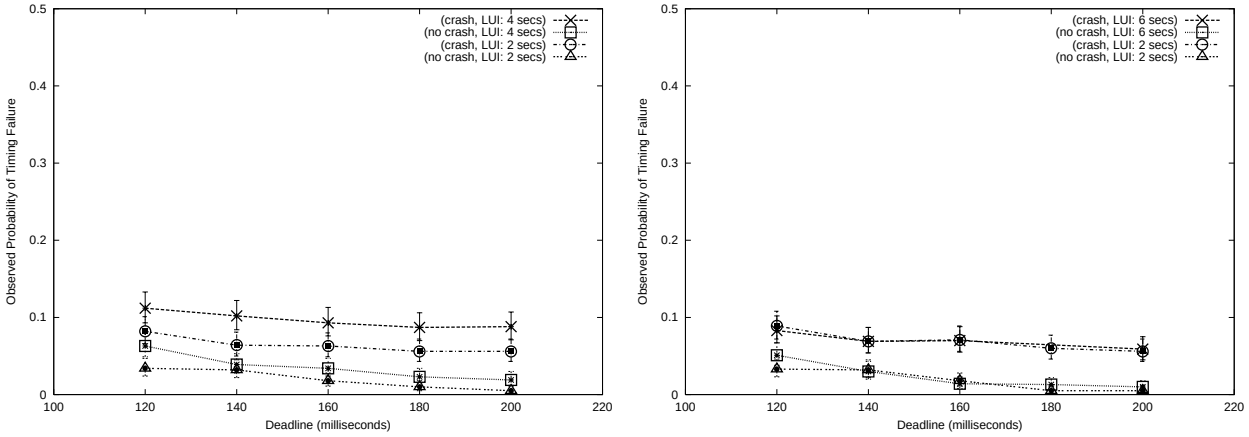
(b) Probability of timing failure: FIFO

Figure 6.3: Sequential vs. FIFO: effectiveness of probabilistic model

Client2, for sequential and FIFO ordering.

The first observation from Figures 6.3a and 6.3b is that in each case, the set of replicas selected by Algorithm 2 was able to meet the client's QoS requirements successfully by maintaining the timing failure probability within the failure probability that was acceptable to the client. For example, in Figure 6.3a, consider the case in which the LUI is 4 seconds and the client requests a probability of timely response of at least 0.9. The probability of timing failures we observe experimentally is 0.1 for a deadline value of 100 milliseconds, and reduces as the deadlines become more flexible. We observe similar behavior for the other cases as well. Thus, for the experimental runs we conducted, the model we used was effective in predicting the set of replicas that can return the appropriate response by the client's deadline, with at least the probability requested by the client.

The second observation from Figures 6.3a and 6.3b is that as the interval between lazy updates increases, the observed probability of timing failures also increases. The reason is that the replica's state becomes increasingly stale as the lazy updates become less frequent. That, in turn, increases the probability that a chosen replica may have to defer its response until it has received the next lazy update, in order to meet the client's staleness threshold. Thus, when the client specifies a staleness threshold that is much smaller than the lazy update interval, fewer replicas are available to respond immediately to the client's request. The delayed response results in a higher probability of timing



(a) Probability of timing failure: sequential

(b) Probability of timing failure: FIFO

Figure 6.4: Sequential vs. FIFO: performance during replica crash

failures. A final observation from Figure 6.3 is that for the same QoS specification and lazy update frequency, more timing failures are observed in sequential ordering than for FIFO. The reason is that in sequential ordering, both clients update a globally shared replicated state, whereas with FIFO ordering, each client updates only the portion of the server state that is specific to the client. The shared updates in the case of sequential ordering cause the replicated state to become obsolete faster, thereby increasing the probability of a delayed response. This suggests that in order to achieve timing failure probabilities that are closer to those obtained using the FIFO handler, we need to propagate the lazy updates more frequently when sequential ordering is used.

6.2.1 Performance During a Replica Crash

Figures 6.2 and 6.3 show the behavior of the model-based algorithm in the absence of a replica crash. Since we have designed our algorithm with the goal of tolerating a single replica crash, we were interested in evaluating the algorithm when one of the selected replicas was unavailable. We conducted experiments to verify the claim we made in Section 5.5.3, which stated that if we choose a set of replicas in such a way that it can satisfy the timing constraint of a client with the specified probability, even if the member of the selected set that has the highest probability of meeting the client's deadline crashes, then such a set should be able to tolerate the failure of any one of the other selected replicas. We now

discuss the results of the experiments we performed to verify this claim.

We used the same experimental setup with two clients that we described in the previous section for the crash-free scenario. We simulated the crash of a replica by having the client handler forward its client's request to all the selected replicas except the one with the highest probability of meeting the client's deadline. Since the client never receives a response from the omitted member, this procedure effectively simulates the scenario in which the member has crashed before completing its service, or is simply unresponsive. Figure 6.4 compares the results obtained for the single crash scenario with the results obtained in the crash-free case, for sequential and FIFO ordering. The plots show how the timing failure probability observed by Client2 varies with the lazy update interval, when Client2 requested a probability of timely response of 0.9. From Figure 6.4, we see that in a scenario in which a replica crashes, there are slightly more timing failures than there are in the corresponding crash-free scenario. However, the graphs show that despite the failure of a replica, the model-based algorithm is able to adapt the selection of replicas to meet the client's QoS specification, for the workload we have considered.

6.3 Cost/Performance Tradeoffs of Lazy Updates

The results presented in Figure 6.3 showed that increasing the frequency of lazy updates resulted in smaller buffering delays for deferred reads, and thereby reduced the occurrence of timing failures. However, there is a cost associated with lazy update propagation. It arises from timer interrupts, network load, and processing of the lazy updates, and the cost increases as the frequency of lazy update propagation increases. Hence, we were interested in analyzing the cost/performance tradeoffs associated with lazy update propagation. We conducted experiments using a more intense workload than the one we used in Section 6.2. Table 6.2 shows the values we used for the experimental parameters. The think time between successive requests is 250 milliseconds, which is one-fourth of the value we used in Section 6.2; the number of clients is twice what we used for the experiments in Section 6.2. The above workload resulted in a client update rate of 5 updates/second, which is nearly five times the update rate

Table 6.2: Experimental setup: impact of the lazy update frequency

Total number of servers	10
Primary/secondary group size	4/6
Number of read requests	1000
Number of update requests	1000
Think time between requests	250 ms
Service time for update requests	Normal(100 ms, 50 ms)
Service time for read requests	Normal(100 ms, 50 ms)
Number of clients	4
QoS of Client1	(a = 2, d = 120 ms, P(d) = 0.5)
QoS of Client2, Client3, Client4	(a = 4, d = 200 ms, P(d) = 0.1)
Ordering guarantee	sequential

of the experiments in Section 6.2 for the sequential consistency case.

Figure 6.5 shows the observed timing failure probability as the lazy update interval increases from 1 second to 6 seconds. We see that as the LUI increases, the failure probability *reduces*, which is in contrast to the behavior shown in Figure 6.3a under the less intense workload. From these results, we conclude that increasing the lazy update frequency has the potential to reduce the buffering delay for deferred reads and thereby improve the responsiveness of the replicas. However, increasing the frequency beyond a certain threshold value causes the overheads associated with the lazy update propagation to become more dominant, nullifying any performance gains. The threshold value is specific to each workload. Thus, our experiments show that the lazy update interval has to be chosen so as to balance the cost/performance tradeoffs, depending on the update rate of the clients, think time, QoS specification of the clients, and the primary/secondary group size.

6.4 Performance Under Load

We now discuss the experiments we carried out to determine how well our replica selection approach adapts to meet the client's QoS specification under different client-induced loads. We varied the client-induced load by varying 1) the think time between the requests, and 2) the number of clients accessing a service. The induced load on the servers is higher for smaller think times. In Figure 6.6, we present

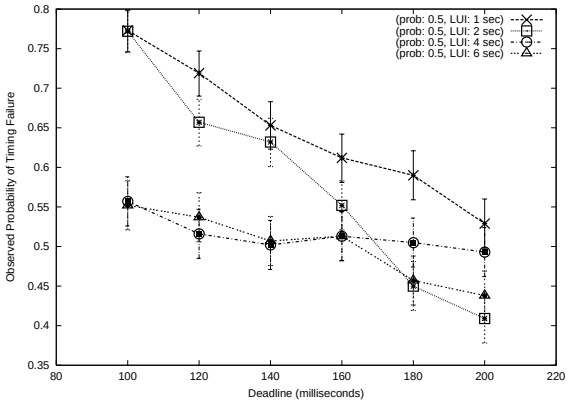
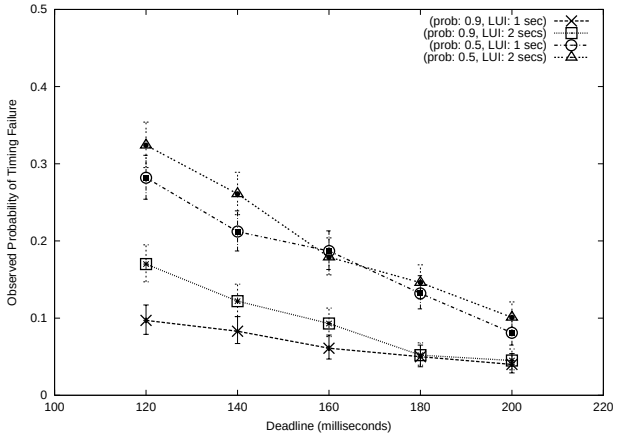


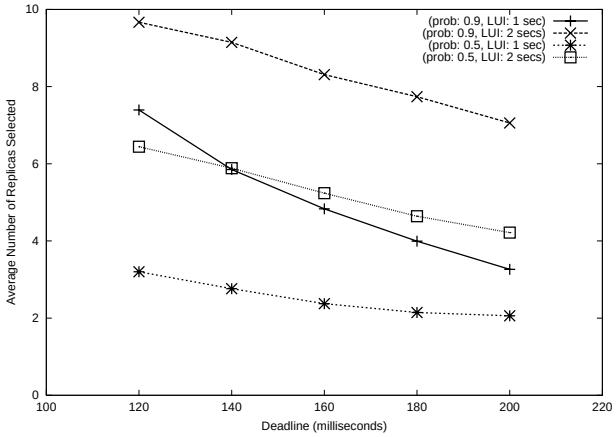
Figure 6.5: Impact of lazy update propagation

the results for sequential and FIFO ordering when we used a think time of 250 milliseconds and lazy update interval values of 1 second and 2 seconds. The graphs in the left-hand column compare the observed timing failure probabilities, while the graphs on the right compare the average number of replicas selected. We compare these results with those presented in Figures 6.2 and 6.3 for which the think time was 1000 milliseconds.

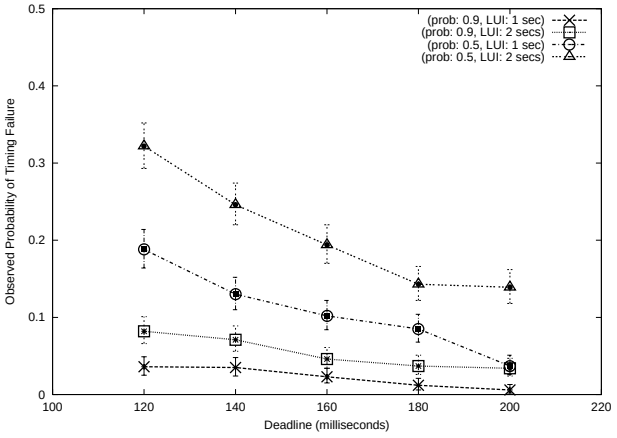
In Section 6.2, when we used a think time of 1000 milliseconds, we used larger values for the LUI (for e.g., 2 and 4 seconds for sequential ordering, and 6 seconds for FIFO ordering). For the experiments in Figure 6.6, we used smaller values of LUI (1 and 2 second intervals). The reason for using a smaller value of LUI is that when the think time is smaller, the rate at which the clients send their update requests is higher. Hence, the state of the secondary replicas becomes out-of-date faster. Therefore, if the lazy updates to the secondary replicas are not sufficiently frequent, there will not be enough replicas whose state is within the specified staleness threshold value of 2 that can deliver a timely response to Client2 with a probability ≥ 0.9. However, when Client2 requested a timely delivery with a smaller probability value of 0.5, we found that the chosen replicas were able to deliver a timely response with a staleness threshold ≤ 2, even when we used larger values of LUI. The reason is that the number of replicas required to meet the timeliness requirements with a smaller probability value is small, as we explained in Section 6.2. This suggests that one way to increase the probability of timely response when the update rate is high is to adaptively increase the frequency of the lazy updates. Hence, we have used smaller lazy update intervals of 1 second and 2 seconds in Figure 6.6 to show the results



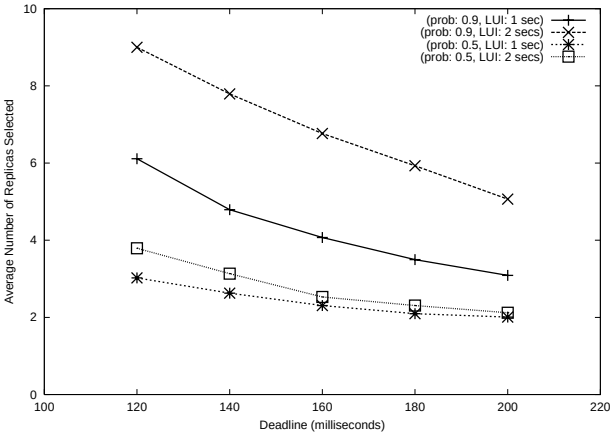
(a) Observed probability of timing failures: sequential



(b) Number of replicas selected: sequential



(c) Observed probability of timing failures: FIFO



(d) Number of replicas selected: FIFO

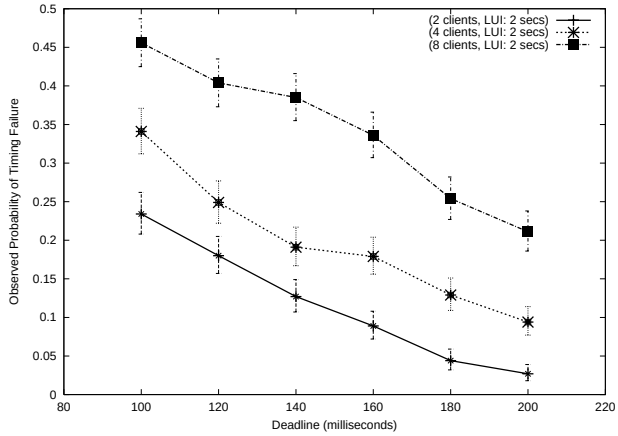
Figure 6.6: Performance under load: think time = 250 ms

under high loads.

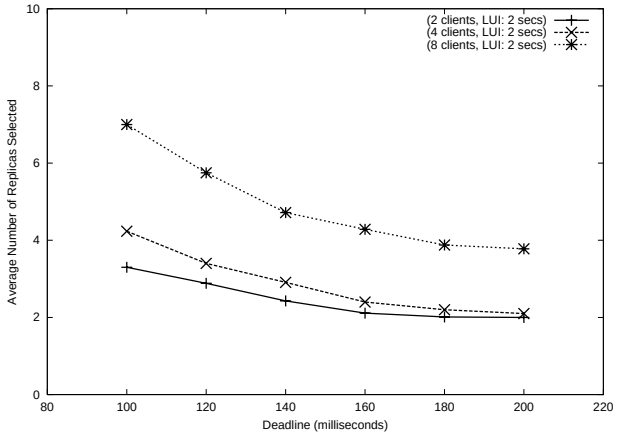
Another observation from comparing the results in Figures 6.3 and 6.6 is that the observed failure probability increases as the think time reduces from 1000 milliseconds to 250 milliseconds. The reason is that as the think time reduces from 1000 milliseconds and approaches values closer to the mean service time of 100 milliseconds, the number of requests that experience queuing delays at the servers increases. Although our selection algorithm usually adapts to choose an appropriate set of replicas to meet the QoS requirements under higher workload intensities, we observe from Figure 6.6a that in certain cases, the probabilistic selection algorithm is unable to find enough replicas to meet the QoS requested by the client. For instance, in the case of sequential ordering, when the think time is 250 milliseconds, the replica subset chosen by the algorithm is unable to deliver a timely response with a probability ≥ 0.9 for deadline values < 140 milliseconds, although we see from Figure 6.6b that the algorithm chooses all 10 available replicas to service the request. In such cases, the selection handler can inform the client that there are insufficient resources to satisfy its QoS requirement, so that the client can choose to renegotiate its QoS specification or be admitted at a later time when the system is less loaded.

6.4.1 Variable Number of Clients

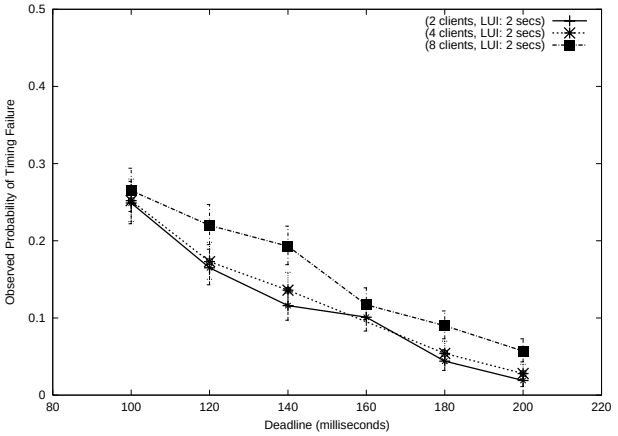
We now consider the case in which we varied the client-induced load by varying the number of clients accessing a service. The induced load on the servers increased with the number of clients. We made the following changes to the experimental setup shown in Table 6.1. We used 2 to 8 clients. All the clients used a value of 1000 milliseconds for the think time. One of the clients specified a different deadline in each run and requested that its deadline be met with a probability ≥ 0.5. All the remaining clients specified a deadline of 200 milliseconds in each run and requested that the deadline be met with a probability ≥ 0.1. The plots in Figure 6.7 evaluate the performance of the sequential and FIFO handlers using 2, 4, and 8 clients when the lazy update interval is 2 seconds. The graphs in the left-hand column of Figure 6.7 show the timing failure probability observed in each case, as measured by the client that specified that its probability of timely response should be at least 0.5; the graphs on the



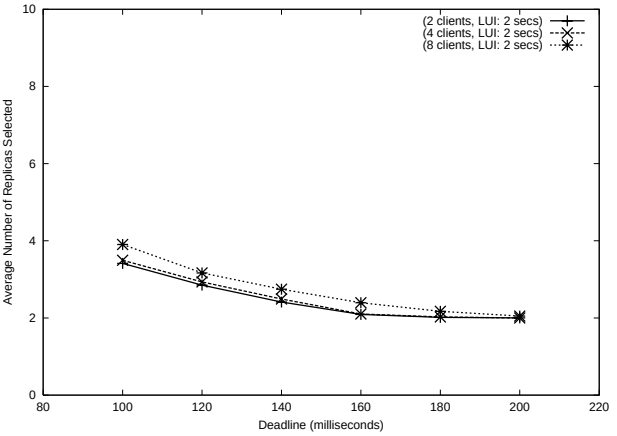
(a) Probability of timing failures: sequential



(b) Number of replicas selected: sequential



(c) Probability of timing failures: FIFO



(d) Number of replicas selected: FIFO

Figure 6.7: Performance under load: variable number of clients

right show the corresponding average number of replicas selected by the probabilistic scheme to meet the QoS specifications of that client.

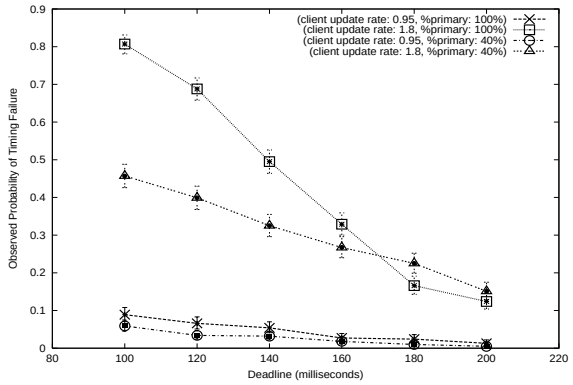
We see that the observed timing failure probability increases as the number of clients requesting service increases, and the increase is more significant in the case of sequential ordering. The increase in the failure probability in the cases of sequential and FIFO ordering is caused by higher queuing delays. In the case of sequential ordering, another factor contributing to the increase is the higher update rate resulting from more clients writing to the shared state. That increases the rate at which the state of the secondary replicas becomes obsolete, and therefore increases the probability of delayed responses. On

the other hand, in the case of FIFO, since each client updates only the client-specific state, the increase in the number of clients accessing a service does not impact the staleness of a replica's state with respect to an individual client. From Figure 6.7, we find that as long as enough replicas are available in the selection pool, the algorithm is able to use the model's prediction to select a subset of replicas that can maintain the failure probability within the threshold acceptable to the client.

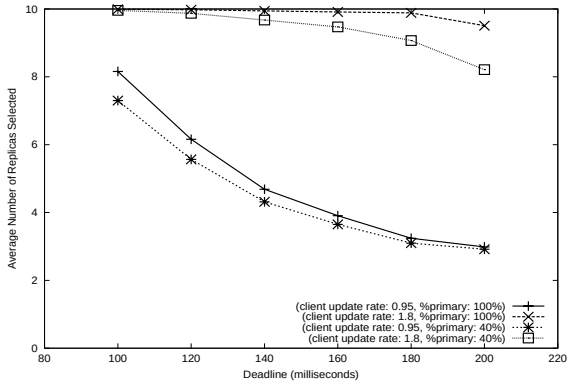
6.5 Single-Tier vs. Hierarchical Replica Organization

In Section 5.3.1, we mentioned that our motivation for using a two-tier replica organization was to favor read operations that can tolerate relaxed consistency to a certain degree, in exchange for better responsiveness. Our thesis was that by limiting the writes to a small subset of replicas, we can improve the availability of the replicas for the read operations that can tolerate relaxed consistency, and thereby reduce the occurrence of timing failures. However, one may argue that a write-all scheme that writes to all the replicas concurrently may result in fewer timing failures. The reason being that unlike the replicas in a two-tier scheme, all the replicas in a write-all scheme can respond without waiting for a state update. In other words, there is no potential for deferred reads in a write-all scheme. Hence, we were interested in experimentally verifying if we were justified in using a hierarchical replica organization with lazy updates for providing timely delivery for applications that can tolerate relaxed consistency.

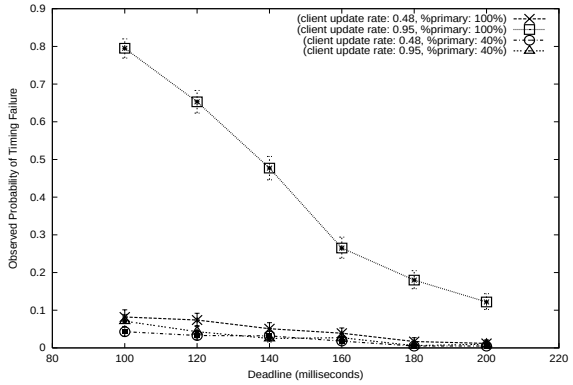
We carried out experiments using the same setup, with 10 server replicas and 2 clients, that was described in Section 6.2. In the two-tier organization, 40% of the replicas were in the primary group, and the lazy update interval for updating the secondary replicas was 2 seconds. For the write-all scheme, we used a single-tier organization in which all the replicas were designated as primaries. Client1 used a QoS specification of deadline = 200 milliseconds, probability of timely response = 0.1, staleness threshold = 4, whereas Client2 specified a QoS specification of probability of timely response = 0.9, staleness threshold = 2, and varied its deadline from 100 to 200 milliseconds. Figure 6.8 compares the performance of the two-tier organization with that of the write-all scheme, for both sequential and FIFO ordering. The graphs depict how the failure probabilities and number of replicas selected for



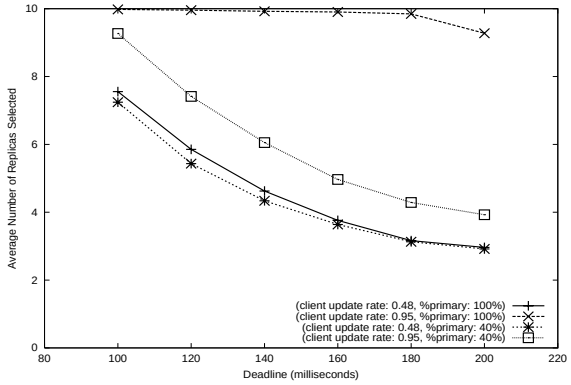
(a) Probability of timing failure: sequential



(b) Number of replicas selected: sequential



(c) Probability of timing failure: FIFO



(d) Number of replicas selected: FIFO

Figure 6.8: Single-tier vs. hierarchical replica groups

Client2 varies with the requested deadline, for different update arrival rates of the clients (denoted by λ_u in Section 5.5.1).

The arrival rate of the updates from the clients for sequential and FIFO ordering is measured as described in Section 5.5.4, and may be varied in several ways. For instance, one could vary it by varying the think time between requests, by sending more write requests in the same interval, and by using more clients to send the updates. We chose to vary it by varying the think time between the update and read requests of the clients, as we now describe. We mentioned in Section 6.2 that each client in our experiments executed 1000 iterations, and in every iteration it invoked an update request followed by a read request. For all of the results shown in Figure 6.8, each client used a think time of 1000

milliseconds between a read and the following update. However, each client used two different values for the think time between an update and the following read request. When they used a value of 1000 milliseconds, it resulted in an update arrival rate of 0.9 updates/second in the sequential ordering case, and 0.48 updates/second in the FIFO ordering case. When the think time was reduced to 0 milliseconds (i.e., each client issued the read immediately after the update request), the update arrival rate almost doubled for both sequential and FIFO ordering. The update rate in the case of sequential ordering is higher than in FIFO ordering, because the sequential update rate includes the updates of both clients, whereas the FIFO update rate is computed independently for each client and does not include the updates of the other client.

From the results in Figure 6.8 we see that when λ_u is smaller, the two-tier and write-all schemes perform comparably, for both sequential and FIFO ordering. However, when the value of λ_u is almost doubled, the write-all scheme results in a significantly higher number of timing failures, even though it chooses more replicas than the two-tier scheme. The reason is that as the update rate increases, the client-induced load on all the replicas in the write-all scheme increases. Although the replicas in the write-all scheme do not have to wait for a state update in order to respond to a request, they have to wait for the previous write to complete before they can respond. Since all the replicas are involved in writes, the queuing delay experienced by read operations is higher and results in higher response times. On the other hand, in the case of the two-tier scheme, only 40% of the replicas are involved in write operations. Although some of the remaining 60% may have to defer their responses until they receive the next state update, the value of the lazy update interval we have chosen ensures that the probability of such deferred reads is small.

We also repeated our experiments with the two-tier scheme using a higher value for the lazy update interval (LUI = 4 seconds in the case of sequential ordering and LUI = 6 seconds in the case of FIFO ordering). This time, when the arrival rate of the updates from the clients was doubled, the write-all scheme resulted in more timing failures than the two-tier scheme did for smaller deadline values ($<$ 140 milliseconds); as the deadlines became more flexible, it performed better than the two-tier scheme. These results show that although a hierarchical scheme allows some of the replicas to have inconsistent

Table 6.3: Experimental setup: impact of the primary group size

Total number of servers	10
Number of read requests	1000
Number of update requests	1000
Think time between requests	250 ms
Service time for update requests	Normal (100 ms, 50 ms)
Service time for read requests	Normal (100 ms, 50 ms)
Number of clients	4
QoS of Client1	(a = 2, d = 120 ms, P(d) = 0.5)
QoS of Client2, Client3, Client4	(a = 4, d = 200 ms, P(d) = 0.1)
Ordering guarantee	sequential order

state, we can bound the replica inconsistency by adaptively choosing a value for the lazy update interval that increases the probability of timely delivery. It must be pointed out that if the client applications always retrieve the most recent update (i.e, staleness threshold = 0), then the write-all scheme results in a higher probability of timely delivery. We have verified this experimentally.

Another observation from the graphs in Figure 6.8a and Figure 6.8b is that for the sequential handler, the selection algorithm chooses almost all the available replicas to service Client2 when the client update rate increases to 1.8. Despite this, neither the write-all scheme nor the two-tier scheme is able to maintain a timing failure probability within 0.1, as requested by Client2. The reason is that the available number of replicas is insufficient to meet the high probability of timely response. The above results justify the choice of a hierarchical replica organization to support relaxed consistency requirements. Although in this work we restrict ourselves to a two-tier organization of replicas to study the tradeoffs between timeliness and consistency, it should be easy to extend our architecture to multiple tiers representing intermediate degrees of staleness in the replica states.

6.5.1 Impact of the Primary Group Size

Having demonstrated the advantages of a hierarchical replica organization over a single-tier organization, we now present results that show how the size of the primary group impacts the performance, for a given number of server replicas. Table 6.3 shows the values we used for the different experimental parameters in order to study the impact of the primary group size. Figure 6.9 shows the probability of

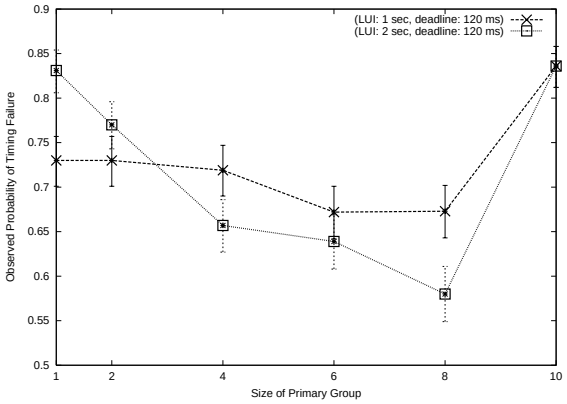


Figure 6.9: Impact of the primary group size

timing failures observed by Client1 as the percentage of replicas in the primary group is varied from 10% to 100% (i.e., all 10 replicas are in the primary group), for two different values of LUI: 1 second and 2 seconds. From Figure 6.9, we see that the observed probability of timing failures reduces as the size of the primary group increases up to the point at which 80% of the replicas are in the primary group. However, increasing the size of the primary group beyond that results in an increase in the number of timing failures. These observations can be explained as follows.

The size of the primary group represents a tradeoff between two different delay factors: the *buffering delay* introduced by the deferred reads and the *queuing delay* caused by the update operations. Increasing the size of the primary group reduces the buffering delay, because more replicas have consistent state. On the other hand, when the arrival rate of updates from the clients is high, increasing the primary group size causes more replicas to be involved in update operations. That results in higher queuing delays, and thereby reduces the availability of the replicas for the read operations. Applying this theory to the results in Figure 6.9, we see that the queuing delay begins to play a more dominant role when more than 80% of the replicas are in the primary group. Although a larger percentage of the replicas have the appropriate state to meet the client's staleness threshold in that region, there are not enough replicas available that can respond within the client's deadline. That is the reason for the increase in timing failures. The above results show that there is a certain optimal ratio between the sizes of the primary and secondary groups that can deliver the best balance between the buffering delay and queuing delay. That ratio is specific to each workload and can be used to configure the size of the two

Table 6.4: Experimental setup: probabilistic vs. round-robin

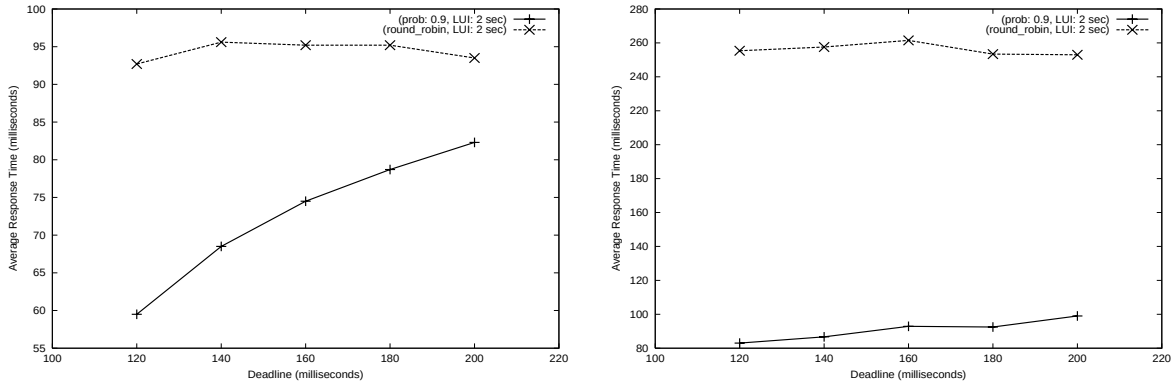
Total number of servers	10
Primary/secondary group size	4/6
Lazy update interval (LUI)	2 seconds
Number of read requests	1000
Number of update requests	1000
Think time between requests	1000 ms, 250 ms
Service time for update requests	Normal (100 ms, 50 ms)
Service time for read requests	Normal (100 ms, 50 ms)
Number of clients	2
QoS of Client1	(a = 4, d = 200 ms, P(d) = 0.1)
QoS of Client2	(a = 2, d = var, P(d) = 0.9)
Ordering guarantee	sequential

groups according to the workload.

Another observation from Figure 6.9 is that the observed failure probability is lower when the frequency of lazy updates has the smaller of the two values (i.e., LUI = 2 seconds). The reason is that beyond a certain threshold frequency, the overhead of the lazy update propagation becomes dominant. We explained this in Section 6.3, using the experimental setup shown in Table 6.3, for the case in which 40% of replicas were in the primary group. Figure 6.9 also shows that as we increase the size of the primary group, we can reduce the frequency of lazy updates, because a larger fraction of the replicas are consistent. Notice from Figure 6.9 that the probability of timing failures exceeds the value tolerated by Client1, when LUI = 1 second as well as when LUI = 2 seconds. This shows that we need to reduce the frequency of lazy updates even further so that we can trade the overheads for the performance gains. We have verified (see Figure 6.5) that an acceptable timing failure probability is achieved when we use a value of around 4 seconds for the LUI.

6.6 Probabilistic vs. Round-Robin Replica Selection

We also conducted experiments to compare the performance of the probabilistic selection scheme with that of a round-robin scheme. Unlike the probabilistic scheme, which makes use of the performance history as well as the elapsed response times of the replicas to select a set of replicas to service a client's



(a) Think time: 1000 ms

(b) Think time: 250 ms

Figure 6.10: Comparison of response times: probabilistic vs. round-robin selection

request, the round-robin scheme uses only the elapsed response time information and selects two of the replicas that the client used least recently to service a request. The round-robin scheme selects two replicas so that it can tolerate single replica crashes, like its probabilistic counterpart. To compare the two schemes, we used the experimental setup shown in Table 6.4.

Figure 6.10a compares how the average response time measured by Client2 varied for different deadlines, using the round-robin and probabilistic selection schemes, when both clients sent their requests with a think time of 1000 milliseconds, while Figure 6.10b presents the results when both clients used a think time of 250 milliseconds; the latter results in higher workload intensity than the former. From the graphs in Figure 6.10 we conclude that the round-robin scheme results in higher response times than the probabilistic selection scheme, on average, under low as well as high workload intensities. The reason is that the round-robin scheme chooses replicas independent of their performance history and client's QoS requirements. In contrast, the probabilistic scheme chooses a set of replicas based on their ability to deliver a response within the requested deadline with a probability of at least 0.9.

The second observation is that while the response times delivered by the round-robin scheme are fairly independent of the client's deadline, the response times delivered by the probabilistic scheme increase as the deadline requested by Client2 increases. The reason is that the round-robin scheme cir-

culates the 1000 read requests of each client among the replicas by choosing different pairs of replicas to service each request. Hence, all the replicas service each client's requests with nearly equal probability, regardless of the client's deadline. On the other hand, the probabilistic scheme selects different sets of replicas for each requested deadline. In Figure 6.2 we showed that the size of the set selected by the probabilistic scheme decreases as the requested deadline becomes less strict. For a given staleness threshold and probability of timely response, the stricter the requested guarantees are, the more responsive the set of the replicas selected by the probabilistic scheme is. Hence, the average response time using probabilistic selection is smaller when the requested deadline is smaller.

Finally, we also compared the throughput of the two schemes experimentally, by measuring the time it took Client2 to execute 1000 alternating read and write requests using each scheme. Although we expected the round-robin scheme to perform better because of its superior load-balancing ability, our experimental results showed that the throughput was 37% higher in the case of the probabilistic selection scheme than in the case of the round-robin scheme. All of the above observations suggest that a dynamic selection scheme based on feedback is more efficient than a round-robin selection scheme that simply balances the load among the available replicas.

6.7 Time-Varying Workload

In the experiments we have described so far, the service time was normally distributed with a mean of 100 milliseconds and variance of 50 milliseconds. The mean workload of those experiments did not vary with time. The results we presented showed that the probabilistic scheme was able to use the performance history of the replicas effectively and adapt the selection of replicas to meet the QoS requested by the clients, when the mean workload was non-time-varying in nature. We now discuss the experimental evaluation of our probabilistic framework using a time-varying workload. The experiments we present are also motivated by the finding that the workloads in many well-known distributed services exhibit *heavy-tailed* distributions. For example, evidence suggests that the retrieval times of files from Web servers as well as from file servers in general, show heavy-tailed properties [26].

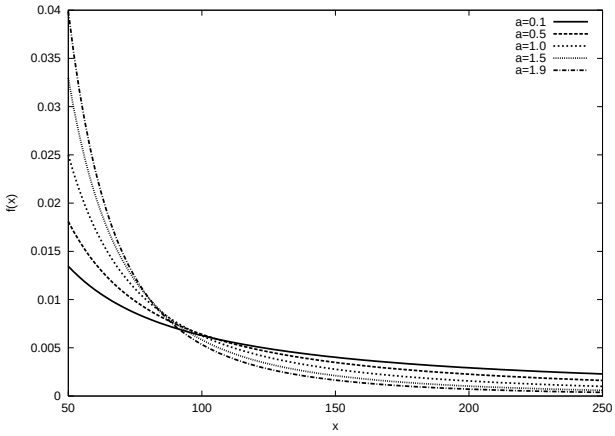


Figure 6.11: Density function of the Bounded Pareto distribution

We first define a heavy-tailed distribution and describe how we use it to model the service time of remote method invocations, in order to generate a time-varying workload.

Heavy-Tailed Distribution: A random variable X follows a heavy-tailed distribution with a tail index α if $P[X > x] \backsim x^{-\alpha}, 0 < \alpha < 2$. Heavy-tailed distributions are characterized by high variability; the variance increases as α decreases. Such a distribution has infinite variance, and has infinite mean when $\alpha \leq 1$. A simple example of a heavy-tailed distribution is the $Pareto$ distribution.

In a typical client/server application, the service time has some upper bound. Hence, we model the service time using a *Bounded Pareto* distribution [26]. The Bounded Pareto distribution is characterized by three parameters: α, which controls the variance and mean of the distribution; k, which is the lower bound for the samples in the distribution; and p, which is the upper bound for the samples in the distribution. The probability density function of the Bounded Pareto distribution is given by

$$f(x) = \begin{cases} \frac{\alpha \cdot k^{\alpha}}{1-(k/p)^{\alpha}} x^{-\alpha-1} & \text{if } k \leq x \leq p, \\ 0 & \text{otherwise.} \end{cases}$$

The Bounded Pareto distribution has finite moments, and therefore does not strictly conform to the above definition of a heavy-tailed distribution. However, it does display high variability when k is

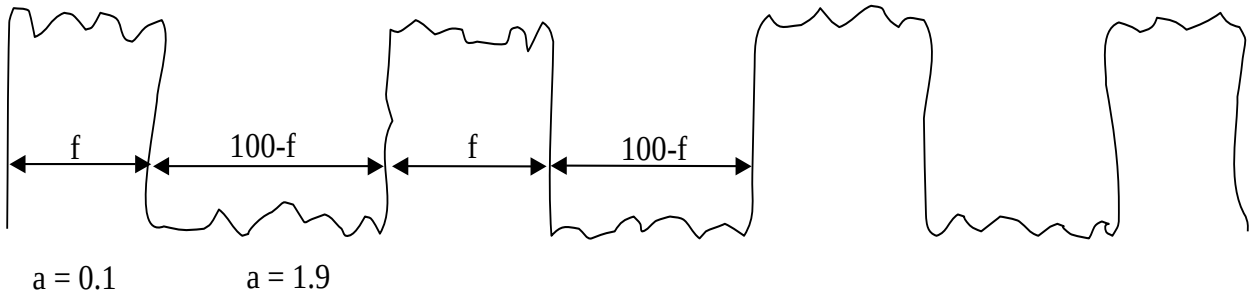
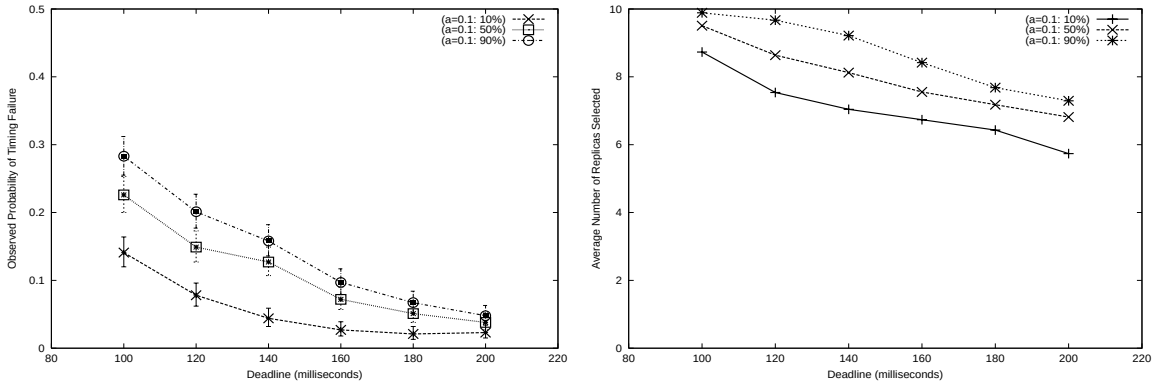


Figure 6.12: Time-varying workload

significantly less than p. We have verified this by generating samples using different values of $\alpha, k,$ and p. In our experiments we set k to 50 milliseconds and p to 250 milliseconds. Thus, the service time varied between 50 and 250 milliseconds. To generate a time-varying workload, we conducted experiments using different values of α. Figure 6.11 shows how the density function varies with the values of α (denoted by 'a' in the figure), $k,$ and p that we used. In general, for a Bounded Pareto distribution, the smaller the value of α, the higher the mean. We now discuss the results when we varied α between two values: 0.1 and 1.9. When $\alpha = 0.1$, the mean of the samples was 121, and when $\alpha = 1.9$, the mean was 84.

We used two clients, Client1 and Client2, each of which sent alternating read and update requests with a think time of 250 milliseconds. Both clients sent 1000 requests of each type. Client1 specified a staleness threshold of 4, a deadline of 200 milliseconds, and a probability of timely response of 0.1, while Client2 specified a staleness threshold of 2, varied its deadline from 100 to 200 milliseconds, and requested a probability of timely response of 0.9. 40% of the replicas were in the primary group, and the lazy updates were propagated to the secondary group at intervals of 2 seconds.

We generated the time-varying workload by varying the service time of a replica between two states, NORMAL and HIGH, as shown in Figure 6.12. For the workload parameters we used, the HIGH state corresponds to $\alpha = 0.1$, where the mean service time was 121 milliseconds, and the NORMAL state corresponds to $\alpha = 1.9$, in which the mean service time was 84 milliseconds. For every 100 requests it received, each replica serviced the first f requests it received in the HIGH state and then made a transition to the NORMAL state, in which it serviced the remaining $100 - f$ requests. The replica then made a transition back to the HIGH state to service the next f requests, and repeated the cycle. To



(a) Probability of timing failures

(b) Number of replicas selected

Figure 6.13: Performance using a time-varying workload

illustrate this procedure with a specific example, let us consider the case when $f = 90$. Each replica services in the HIGH state the first 90 requests it receives, and then services in the NORMAL state the following 10 requests. It then services the next 90 requests in the HIGH state and so on. Thus, the length of the cycle is 100 requests, and the number of requests a replica processes in a state can be considered the *burst length* of that state. Although we use a discrete measure, such as the number of requests that have been processed in a state, to control the transition between the states, one may alternatively use the duration of time spent in each state to control the transition.

Figure 6.13 presents the observed probability of timing failures and the average number of replicas selected for Client2 for different values of f, using sequential consistency guarantees. The first observation from Figure 6.13a is that the probability of timing failures increases as the percentage of requests processed in the HIGH state increases from 10% to 90%. This is to be expected as an increasing percentage of requests experience higher service times that have a mean close to 121 milliseconds. The second observation is that when the value of f increases, the observed failure probability exceeds the client's expectation for deadline values that are close to the mean service time. We considered two possible explanations for this. Our first hypothesis was that the model is unable to adapt to a time-varying workload. To verify the hypothesis, we conducted experiments with an equivalent, non-time-varying workload that used a Bounded Pareto distribution with the same parameter values as described above

for the time-varying case. In the non-time-varying workload, the transition between the HIGH state and low state was controlled using a probabilistic measure, as follows. Before servicing a request, each replica used a uniform random number generator to generate a value p between 0 and 1. Like the time-varying case, we studied the performance using a non-time-varying workload for the following three cases:

1. when $p > 0.9$, the replica serviced the request in the HIGH state; when $p \leq 0.9$ it serviced the request in the NORMAL state. This is equivalent to the time-varying case in which 10% of requests were serviced in the HIGH state ($f = 10$).

2. when $p > 0.5$, the replica serviced the request in the HIGH state; when $p \leq 0.5$ it serviced the request in the NORMAL state. This is equivalent to the time-varying case in which 50% of requests were serviced in the HIGH state ($f = 50$).

3. when $p > 0.1$, the replica serviced the request in the HIGH state; when $p \leq 0.1$ it serviced the request in the NORMAL state. This is equivalent to the time-varying case in which 90% of requests were serviced in the HIGH state ($f = 90$).

Our hypothesis was that if the replicas selected by the model are able to maintain a failure probability within the acceptable threshold of 0.1, in the case of non-time-varying workload, then it indicates that our model is unable to cope with a time-varying workload. However, we observed that the behavior in the non-time-varying case was nearly identical to that presented in Figure 6.13, for the time-varying workload.

The second possible explanation is that there are not enough replicas that can deliver a timely response with a probability of at least 0.9 for smaller deadline values, under a higher workload. From Figure 6.13b we see that the model tries to meet strict requirements under higher workload by choosing more replicas. However, we see that in certain cases there are not enough replicas available to deliver a timely and consistent response with the requested probability. In such cases, our model saturates the entire pool of replicas. The above results show that the model can adapt to a time-varying workload. However, if there are stringent demands, there may not be enough replicas available to meet the

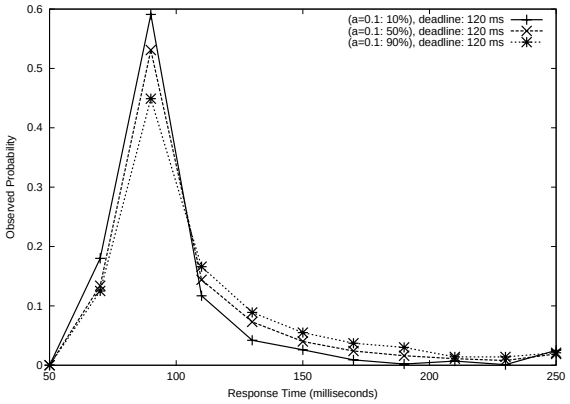


Figure 6.14: Predictability of end-to-end response time

demands. The above discussion suggests that in such cases, we can adapt by resorting to the same measures we suggested in Section 6.4 for a non-time-varying workload. Those measures include increasing the size of the available replica pool and propagating the lazy updates more frequently, so that we have more replicas with up-to-date state.

The above experiments evaluate the effectiveness of the model's prediction under a time-varying workload, but we also wanted to learn how the variability in the service times of the replicas under such a workload affects the predictability of the end-to-end response times perceived by clients. Since the service time, which constitutes a significant portion of the end-to-end response time, oscillates between two states, one may expect the end response time to also exhibit a bimodal distribution. Furthermore, since the algorithm selects different replicas to process different requests, the predictability of the response times may be of interest to client applications that have limited tolerance for jitter, such as multimedia applications. Figure 6.14 shows how the density of the response times perceived by Client2 varies as the percentage of read requests processed in the HIGH state varies from 10% to 90% (i.e, the burst length of the HIGH state varies from 10 to 90). The results are presented for the case in which Client2 specified a deadline of 120 milliseconds. The distribution is computed based on a history of response time measurements for a window of 1000 read requests. Figure 6.14 shows that the response times perceived by the client are fairly predictable, even though the probabilistic algorithm chooses different sets of replicas to service the requests in the window and the service times of the individual replicas follows a bimodal distribution. The reason for the unimodal response time distribution

shown in Figure 6.14 is that the selection algorithm chooses replicas to service a request based on their ability to meet the deadline (in this case 120 milliseconds). The above result shows that the selection is effective, despite the time-varying nature of the server workload.

Summary

The experimental results we have presented show that the model we developed to assign the replicated servers to service the clients allows a QoS-aware middleware to adapt the assignment based on changes in the load and on the timeliness and consistency requirements of the clients. The model also helped us understand the tradeoffs between timeliness and consistency for different consistency semantics. We have shown the effectiveness of the replica allocation scheme under different scenarios, including varying workload intensities, single replica crashes, and different QoS specifications. The results we presented show that the frequency of lazy updates is an important parameter that allows us to tune the tradeoffs between the desired levels of consistency and timeliness. Some of the factors that one must consider when choosing the frequency include the arrival rate of the updates from the clients, the ratio between the sizes of the primary and secondary replica groups, and the timeliness, as well as the consistency specification of the clients.

Chapter 7

Dynamic Replica Creation

I want to know how God created this world. I am not interested in this or that phenomenon, in the spectrum of this or that element. I want to know His thoughts; the rest are details.

- Albert Einstein

Any system that supports server replication must deal with two fundamental issues. One is that of distributing the requests of the clients among the replicas, which we have addressed in the previous chapters. The other is that of deciding the size of the pool of available replicas and the placement of the replicas. In a system that supports dynamic replication, both of these issues influence the availability of the system. We have so far assumed that the size of the replica pool is decided at initialization time and remains constant, unless a replica crashes. In this chapter, we address the problem of creating replicas on demand. In order to support dynamic replica creation, we need to determine when, where, and how many replicas to create. In the following sections, we discuss how we have addressed these issues within the AQuA framework.

7.1 Introduction

Availability of a replicated service is determined both by the number of replicas and the placement of the replicas. Merely increasing the degree of replication does not necessarily result in higher availability. For example, consider a scheme in which every new replica is created on the same node. That would increase the number of replicas, but if the node hosting all the replicas should fail, it would make

the service unavailable. Similarly, if each new replica is placed on nodes that are on bottleneck links (links that are either slow or susceptible to congestion), then adding new replicas does not improve the responsiveness of the system. Hence, in order to improve the availability of the system, placement decisions have to be considered when replicas are created dynamically.

In our work, we use the observed probability of timing failures as the metric for measuring availability. In the work that we described in the previous chapters, we created a pool of replicas at initialization time. If that pool was insufficient to provide a timely response for a client with the specified probability, then the middleware notified the client through a callback. The client was then allowed to adapt in a manner that best suited its requirements. In other words, the middleware used an application-aware model of adaptation when there were not enough resources to meet a client's demands. An alternative approach would be to have the middleware add more resources to meet the demands. We now address the issue of creating replicas on demand, if more replicas are required to satisfy a client's requirements.

We use the dependability manager component of AQuA, described in Section 2.3.2, to create a new replica. The following decisions need to be made when creating a new replica:

- when should a replica be created?
- how many replicas should be created?
- where should the new replica be placed?
- what state should the new replica be initialized with?

In the following sections, we describe the approach we have taken to address the above issues.

7.2 Determining the Instant of Replica Creation

The factors that trigger the creation of a replica depend on what the replicas are being used for. For example, in the case of active and passive replication handlers in AQuA [65], replicas are created to tolerate a certain number of faults, as specified by the clients. Those handlers use the number of faults to decide the degree of replication. In an attempt to maintain that degree of replication, the

dependability manager creates a new replica whenever an existing replica fails. On the other hand, in a system that uses replication to deliver good response times, a new replica may be created whenever the load on the existing replicas exceeds a certain threshold. The load on a replica may be measured by the number of requests or the rate at which the requests are sent to a replica. In our approach, the replica selector in a client's gateway requests the dependability manager to create a replica when the selector is unable to find enough replicas to provide a timely response with the probability specified by the client. As mentioned in the earlier chapters, the selector uses the response time distribution functions of the replicas to determine if they can meet the QoS requirements of the clients.

7.3 Computing the Number of Replicas

There are several ways to determine the number of replicas that need to be created on demand. We can either create a constant number of replicas or choose the number adaptively based on feedback from the system. In the case of active and passive replication handlers in AQuA, the number of replicas created at runtime is the difference between the degree of replication required to tolerate the number of faults specified by the clients and the current degree of replication. The work described in [62] makes use of the probability of node failures as input to an analytical model to determine the number of replicas that need to be created to provide a certain degree of availability. We use an incremental approach in which a client gateway requests the dependability manager to create a single replica whenever more replicas are needed to meet the timeliness requirements, as mentioned in the previous section. Our approach can easily be extended to create multiple replicas per request.

7.4 Replica Placement

As mentioned earlier, it is important to locate the replicas appropriately, in order to make the best use of replication. There are several ways to select the node that will host a new replica. The simplest way is to choose the node at random. However, if the goal of creating a replica is to improve the responsiveness for the clients, then choosing the placement becomes a more interesting and challenging problem. One

possible approach is to monitor the access patterns of the clients and locate the replica in close proximity to the clients that access the service more frequently. Alternatively, the new replica may be located near the clients that have most stringent time constraints. In a wide-area network in which some of the links can potentially become slow or congested, it would be more appropriate to use a dynamic placement scheme that uses feedback from the system to guide the location, so that we can avoid placing the replicas on bottleneck links. In our approach, the dependability manager places the new replica on the least loaded host. As mentioned in Chapter 2, the object factory on each host periodically monitors the load on that host and conveys the load information to the dependability manager. This enables the manager to pick the least loaded host. We currently do not use the geographical proximity or the propagation delays in deciding the placement, because AQuA is LAN-based. However, if we extend our work to large-scale networks, it may be more appropriate to monitor the network delays and the host load and use a combination of the two to decide the placement.

7.5 State Initialization

In the case of stateless replicas, a newly created replica may begin to service the clients immediately upon creation. However, in the case of replicas with dynamic state, the new replica must inherit the appropriate state before beginning to service a client. Since our framework organizes the replicas with state into two groups, the state initialization depends on which group the new replica joins. If the replica joins the primary group, it has to be initialized with the most up-to-date state, because the primary group implements strong consistency semantics. The new replica can request a state transfer from the lazy publisher. The state transfer would include the current application state; the gateway state of the lazy publisher, such as the commit sequence number; and the update requests that are in the request queue waiting to be committed by the lazy publisher. The new replica would first initialize its state with the current state of the lazy publisher. It could then commit the pending updates by respecting the ordering guarantee, and then process the requests in the new view. An alternative approach would be to defer the state transfer until the lazy publisher has committed all the updates in the old view.

In our approach, the new replica joins the secondary group, instead of the primary group. When the lazy publisher disseminates its state to the secondary group, the new replica inherits the state. While this would prevent the new replica from responding to the client until the next lazy update, it avoids the overheads of an additional state transfer, which would be required if the replica had joined the primary group. When the states of the primary and secondary replicas become coherent at a later time, the middleware can promote one or more of the secondary replicas to the primary group, if the middleware decides that such a transition would help the system meet the timeliness requirements of the clients.

7.6 Performance Measurement

In this section, we analyze the time it takes to create replicas dynamically, using our implementation. We first explain the protocol steps and then present the performance measurements characterizing the replica creation. We use a reasonably lightweight protocol for replica creation; the following steps are executed from the time a client gateway decides that it needs more replicas to meet the QoS requirements of the client, until the time the replicas are created.

1. The client gateway requests the dependability manager to create additional replicas by communicating with the manager using a *dynamic send handler* [63]. The dynamic send handler allows a client to temporarily join the group comprising the dependability manager and its replicas, multicast the replica creation request to the group using Ensemble, and leave the group when the message has been transmitted. The message sent by the client includes the name of the client application, the name of the server application whose replica pool has to be increased, the number of additional replicas required, and a flag indicating whether the new replicas are in the primary or secondary group.

2. Upon receiving the message, the dependability manager first extracts the arguments. It then traverses through the list of hosts in increasing order of the most recently reported host load and sends a replica creation request to the object factories running on the hosts. The dependability

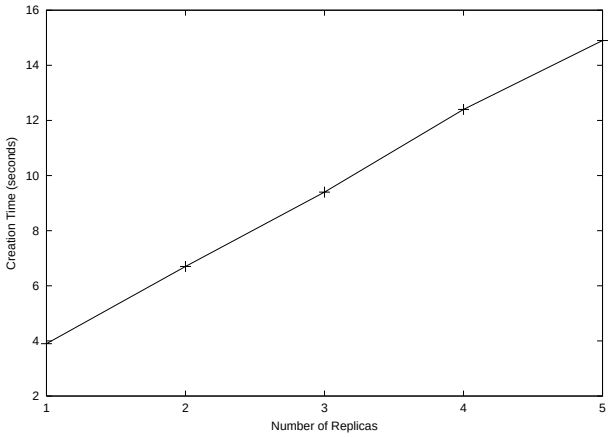


Figure 7.1: Replica creation time

manager contacts only as many object factories as the number of additional replicas requested by the client.

3. Upon receiving a replica creation request, each object factory starts a new replica on the respective host, using the configuration specified by the service configuration file. An example of a service configuration file that we have used is provided in Appendix A.

4. The new replica joins the appropriate server replication group, according to whether it is a primary or secondary replica. Ensemble notifies the group members of a membership change when the join is successful. The new replica also joins the QoS Group (see Section 5.3.1), which allows the clients to subscribe to receive the performance updates of the new member.

We experimentally measured the time it takes to create different numbers of replicas at runtime using the above protocol. These experiments were conducted on single processor machines with a processor speed of 1 GHz. Figure 7.1 shows the results. From the figure, we observe that it takes about 4 seconds to create the first replica, and about 3 seconds to create each additional replica. The extra 1 second for the first replica denotes the time to execute steps 1 and 2 above. Since we use a single request to create multiple replicas, the overheads involved in steps 1 and 2 are incurred per-request rather than per-replica. However, steps 3 and 4 are executed for creating each new replica. It takes about 0.06 seconds to execute step 3, while the remainder of the 3 seconds is contributed by step 4. Thus, the overheads involved in joining a group dominates the replica creation time. Once the new

replica has joined the group, it has to further wait to inherit the appropriate state before it can service a client's request. If the new replica is in the secondary group, the waiting time depends on the lazy update interval (LUI) and the time to propagate the state across the network.

Summary

Responsiveness of a replicated service depends on the number of available replicas as well as their placement. In times of high demand, it may be necessary to create replicas dynamically, so that the response time constraints of the clients can be met. In this chapter we described some of the issues involved in creating replicas on demand in order to improve the responsiveness, and presented how we deal with these issues within the AQuA framework.

Chapter 8

Concluding Remarks

I hope that posterity will judge me kindly, not only as to the things which I have explained, but also to those which I have intentionally omitted so as to leave to others the pleasure of discovery.

- Rene Descartes, La Geometrie.

We conclude this dissertation by suggesting some key avenues for future exploration and then presenting our conclusions.

8.1 Future Directions

The work in this dissertation suggests some interesting and potentially useful directions for future work. Some of them are direct extensions of the work we have described in this dissertation. Other ideas relate to generalizing our framework more broadly for QoS-based resource allocation.

8.1.1 Alternative Specifications

In our current QoS model, the clients express their timeliness requirements by specifying their deadlines and probability of timely response. While it is easy for the clients to specify the deadline values for their requests, the way they should choose appropriate probabilities of timely response may not be very obvious. Our experimental results showed that for a given deadline, the higher the requested probability of timely response, the greater the number of replicas our algorithm assigns to a client. Hence, the

probabilistic input plays an important role in deciding the number of replica resources to allocate to service a client's request. We will now discuss some alternative interpretations of the probabilistic input.

The probability input may be interpreted as a means for the clients to express the "importance" level at which they want their requests to be serviced. If we allow importance levels to be expressed as integral values between 0 and 10, then the middleware can internally map the importance level to a probabilistic value by simply dividing the importance level by 10. Alternatively, instead of using a numerical value for the importance, the clients can specify the importance class using qualitative measures, such as HIGH, MEDIUM, or LOW. The middleware can internally map these importance classes onto their probabilistic equivalent before running the replica selection algorithm. A simple mapping, for example, would be to associate the HIGH importance class with a minimum probability of timely response of 0.9, the MEDIUM importance class with a minimum probability of timely response of 0.5, and the LOW importance class with a minimum probability of timely response of 0.0. If the middleware is unable to allocate enough replicas to meet a client's requirement, it can issue a callback to the client. The client can then choose to either lower its importance level or access the service at a later time when enough replicas are available.

Another interesting application of our probabilistic model would be in environments that enforce accounting of resources. In such environments, clients are often charged based on the amount of resources they consume. Thus, a client can specify a deadline for its requests and the price it is willing to pay if its timing constraints are met. The middleware can then map the client's budget to an appropriate probability of timely response and allocate the appropriate number of replicas to service the client's request, using the probabilistic models we have developed. Clients that are willing to pay a higher price will receive a higher probability of timely response. This form of accounting also has the additional advantage of preventing the clients from indiscriminately choosing a high level of service for their requests and thereby penalizing the other clients.

8.1.2 Admission Control

The experimental results we presented demonstrate that when the confidence that the individual replicas can meet a client's QoS requirements is low, as in the case of overloads or high failure rates, our algorithm adapts by choosing a higher degree of redundancy to process a request. In other words, the greater the uncertainty in the system, the higher is the chosen redundancy. As we have seen, our replica selectors use a partially greedy approach in which the primary goal is to satisfy the requirements of the local clients. While the replica selectors choose no more than the number of replicas predicted by the probabilistic model to service a request, in the worst case they may forward the request to the entire replica set. That may further aggravate the load in an already loaded condition.

There are different ways to deal with that situation. In Chapter 7, we presented one approach, which was to create replicas dynamically when demand arises. Another alternative that our framework uses is to inform the clients about insufficient resources through a callback. This allows the clients to renegotiate their QoS requirements. While this approach may result in degraded service, it may be acceptable to those clients who do not want to be denied service. A third alternative is to use admission control. Currently, we admit all the clients and assign replicas to competing clients in a decentralized manner using local utility measures. There are no global measures that guide the local decision-making. In times of heavy demand, we can impose some restriction on the clients that can be admitted. For example, we can interpret the probability of timely response as indicative of the importance level of a client and choose to admit only those clients that have importance levels higher than a certain value. Alternatively, we can choose to admit only those clients whose QoS requirements can be met by assigning no more than h replicas, where h is a tunable parameter whose value can be configured by the system administrator or by an entity that has a global view of the system, such as the dependability manager in AQuA. One way to choose the value of the upper bound h is by formulating appropriate global, system-specific criteria that can be used in conjunction with the local, client-specific requirements to assist the decentralized resource allocation. The local criterion we currently use is that the QoS specification of a client should be met. An example of a global criterion is the minimization of the total number of timing failures in the system in a period of time, which can be defined as the sum

of the timing failures experienced by all the clients that are currently admitted in the system.

8.1.3 Modeling the Lazy Update Interval

Our experimental results have shown that the lazy update interval is an important parameter that allows us to tune the tradeoff between timeliness and consistency. We have seen in Section 6.3 that increasing the frequency of the lazy updates up to a certain threshold value has the potential to improve the responsiveness. However, beyond that threshold, the overheads associated with the lazy updates start to play a significant role. Based on our experiments, we have determined that the threshold value depends on the number of clients, the rate at which updates arrive from the clients, the ratio of the primary group size to the secondary group size, and the timeliness as well as consistency requirements of the clients. Currently, we statically configure the lazy update interval upon initialization. However, it would be useful to adapt the interval at runtime by monitoring the number of timing failures globally across all the clients. For instance, when the middleware is unable to deliver timely responses with the requested probability for more than a certain threshold number of clients, the middleware can reduce the lazy update interval. It would, however, be more challenging to formulate an analytical model that relates the lazy update interval to the various factors mentioned above, so that the model can guide the lazy publisher in adaptively choosing the appropriate interval at runtime.

8.1.4 Scaling to Large-Scale Networks

We have currently used our framework to experimentally validate our models over a LAN. We now discuss some of the challenges involved in extending our work to large-scale networks. First, our assumption that the gateway delays are fairly constant and small compared to the service time may no longer be valid in a wide-area network, because propagation delays play a significant role in the wide area. Hence, rather than consider only the most recently recorded value of the gateway delay, as we have done, we need to record a history of the gateway delays and use the history when computing the response time distribution function as a discrete convolution of the service time, queuing delay, and gateway delay.

Second, we need a way to track the performance characteristics of the replicas in a scalable manner. In our current implementation, each replica disseminates its performance updates to all of the clients. In a larger scale system, in which the replicas and clients are more numerous and more widespread, it may not be feasible to propagate the performance updates to all of the clients in a timely manner, on account of larger latencies. This could result in a greater degree of inaccuracy in the performance histories. Hence, in order to improve the accuracy of the prediction made by the probabilistic models, we need a scalable approach to track the global performance characteristics in a larger system. In general, it is a challenge to balance communication costs with information accuracy as a function of system size and global characteristics.

One way to do so involves a clustering approach similar to the one proposed in Opus [7]. Using that approach, we can organize the replicas into a hierarchy of clusters by placing the replicas that are in close proximity in the same cluster. For example, all the replicas on the same LAN might form one of the clusters in the hierarchy. Using this scheme, the framework we described in the previous chapters can be thought of as a hierarchy consisting of a single cluster. The frequency with which a client receives performance updates from the replicas depends on its proximity to the different replica clusters. The replicas in a cluster directly disseminate their performance updates to the clients that are in close proximity every time they service a request, just as is done in our current framework. In order to propagate the updates to remote clients, every cluster elects a leader, and the replicas in a cluster forward their performance updates to the leaders of the remote clusters. Each leader then forwards a summary of the performance updates from the remote clusters to the clients that are in close proximity. In other words, the closer a client is to a replica cluster, the more accurately it can track the performance information of the replicas in that cluster. At the time of replica selection, we can factor in the inaccuracy by associating the response time distribution functions of the replicas with a weight that is proportional to their accuracy.

8.2 Conclusions

The framework we have developed enables a middleware to accommodate diverse application requirements by implementing them as protocols tailored to different application-specific requirements. The framework allows a dependable middleware to assign replicated servers to clients adaptively based on the QoS requirements of the clients and the current responsiveness and state of the replicas. The framework actively monitors the replicas at runtime and uses the feedback to guide the adaptation. The framework assigns replicas based on the prediction made by probabilistic models that are easy to compute at runtime. The experimental results we obtained demonstrate the role of feedback and the efficacy of analytical models for adaptively sharing the available resources among the users in a range of different scenarios. While a static scheme or round-robin scheme would be sufficient when the primary goal is load balancing and when the clients do not have specific timing constraints, we believe that a dynamic scheme, like the model-based replica selection scheme we have developed, would be useful in an environment in which time-sensitive clients that have different QoS specifications access compute-bound service providers that display significant variability in their response times.

Although our probabilistic approach was developed mainly for adaptively sharing replicated servers in uncertain environments, similar techniques can be applied to a range of problems, including scheduling and other resource allocation problems. Given the diversity of the requirements of client applications when accessing distributed services, such adaptive frameworks which rely on feedback-based control are likely to play an increasing role in solving a range of problems related to building dependable systems.

References

[1] T. Abdelzaher. *QoS Adaptation in Real-Time Systems*. PhD thesis, University of Michigan, Ann Arbor, August 1999.

[2] R. Alonso, D. Barbara, and H. Garcia-Molina. Data Caching Issues in an Information Retrieval System. *ACM Transactions on Database Systems*, 15(3):359–384, 1990.

[3] S. Bhola, G. Banavar, and M. Ahamad. Responsiveness and Consistency Tradeoffs in Interactive Groupware. In *Proc. of the 7th ACM Conference on Computer Supported Cooperative Work*, pages 79–88, November 1998.

[4] K. Birman. Replication and Fault Tolerance in the ISIS System. In *Proc. of the 10th ACM Symp. Operating Systems Principles*, pages 79–86, December 1985.

[5] K. Birman. *Building Secure and Reliable Network Applications*. Manning, 1996.

[6] K. Birman, M. Hayden, et al. Bimodal Multicast. *ACM Transactions on Computer Systems*, 17(2), May 1999.

[7] R. Braynard, D. Kostic, A. Rodriguez, J. Chase, and A. Vahdat. Opus: an Overlay Peer Utility Service. In *Proc. of Open Architectures and Network Programming*, June 2002.

[8] A. Campbell. Mobiware: QoS-aware Middleware for Mobile Multimedia Communications. In *Proc. of IFIP 7th Intl. Conf. on High Performance Networking*, April 1997.

[9] R. Carter and M. Crovella. Dynamic Server Selection using Bandwidth Probing in Wide Area Networks. Technical report, Boston University, BU-CS-96-007, 1996.

[10] OMG Technical Committee. *Common Object Request Broker: Architecture and Specification*. Object Management Group, 2.3 edition, June 1999. `www.omg.org/docs/orbos`.

[11] M. Cukier et al. AQuA: An Adaptive Architecture that Provides Dependable Distributed Objects. In *Proc. of the IEEE Symp. on Reliable Distributed Systems*, pages 245–253, October 1998.

[12] A. Demers, D. Greene, C. Hauser, W. Irish, and J. Larson. Epidemic Algorithms for Replicated Database Maintenance. In *ACM Symp. on Principles of Distributed Computing*, pages 1–12, 1987.

[13] W. Edwards, E. Mynatt, K. Petersen, M. Spreitzer, D. Terry, and M. Theimer. Designing and Implementing Asynchronous Collaborative Applications with Bayou. In *Proc. of the 10th ACM Symp. on User Interface Software and Technology (UIST)*, pages 119–128, October 1997.

[14] Z. Fei, S. Bhattacharjee, E. Zegura, and M. Ammar. A Novel Server Selection Technique for Improving the Response Time of a Replicated Service. In *Proc. of the IEEE INFOCOM'98*, March 1998.

[15] P. Felber, X. Défago, P. Eugster, and A. Schiper. Replicating CORBA Objects: A Marriage between Active and Passive Replication. In *Proc. of the Second IFIP International Working Conference on Distributed Applications and Interoperable Systems (DAIS'99)*, pages 375–387, June 1999.

[16] I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt, and A. Roy. A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation. In *Proc. of the International Workshop on Quality of Service*, 1999.

[17] I. Foster, A. Roy, and V. Sander. A Quality of Service Architecture that Combines Resource Reservation and Application Adaptation. In *Proc. of the 8th International Workshop on Quality of Service*, 2000.

[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[19] D.K. Gifford. Weighted Voting for Replicated Data. In *Proc. of the 7th Symposium on Operating Systems and Principles*, pages 150–162, 1979.

[20] A. Goel, G. Popek, and C. Pu. View Consistency for Optimistic Replication. In *Proc. of the IEEE Symp. on Reliable Distributed Systems*, October 1998.

[21] R. Golding. A Weak-Consistency Architecture for Distributed Information Services. *Computing Systems*, 5(4):379–405, 1992.

[22] R. Golding. *Weak-Consistency Group Communication and Membership*. PhD thesis, University of California at Santa Cruz, December 1992.

[23] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kauffman, 1993.

[24] Object Management Group. *Fault Tolerant CORBA (adopted specification)*. OMG Technical Committee, March 2000.

[25] R. Guerraoui and A. Schiper. Software-Based Replication for Fault Tolerance. *IEEE Computer*, pages 68–74, April 1997.

[26] M. Harchol-Balter, M. Crovella, and C. Murta. On Choosing a Task Assignment Policy for a Distributed Server System. In *Proc. of Performance Tools*, pages 231–242, September 1998.

[27] D. Hardin and E. Jensen. Development of the Real-Time Specification for Java. `www.real-time.org`, 1999.

[28] M. Hayden. *The Ensemble System*. PhD thesis, Cornell University, January 1998.

[29] J. Heidemann and V. Visweswaraiah. Automatic Selection of Nearby Web Servers. Technical report, University of Southern California, USC TR 98-688, 1998.

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

[31] M. Horstmann and M. Kirtland. DCOM Architecture. `msdn.microsoft.com/library`.

[32] D. Hull, W. Feng, and J. Liu. Enhancing the Performance and Dependability of Hard Real Time Systems. In *Proc. of the IEEE Computer Performance and Dependability Symposium*, pages 174–182, April 1995.

[33] Object Computing Inc. *TAO Developer's Guide, Version 1.0*. Object Computing Inc., 1999.

[34] S. Jajodia and D. Mutchler. Dynamic Voting Algorithms for Maintaining the Consistency of a Replicated Database. *ACM TODS*, 15(2):230–280, 1990.

[35] Java Specification. `java`.sun.com.

[36] N. Johnson, S. Kotz, and A. Kemp. *Univariate Discrete Distributions*, chapter 3, pages 129–130. Addison-Wesley, second edition, 1992.

[37] B. Kantor and P. Rapsey. Network News Transfer Protocol. RFC977, Feb 1986. `http://www.cis.ohio-state.edu/htbin/rfc/rfc977.html`.

[38] E. Katz, M. Butler, and R. McGrath. A Scalable HTTP Server: The NCSA Prototype. *Computer Networks and ISDN Systems*, 27:155–164, 1994.

[39] P. Keleher. Decentralized Replicated-Object Protocols. In *Proc. of the 18th ACM Symp. on Principles of Distributed Computing*, April 1999.

[40] K. Kim and C. Subburaman. ROAFTS: A Middleware Architecture for Real-Time Object-Oriented Adaptive Fault Tolerance Support. In *Proc. of the IEEE High Assurance Systems Engineering*, pages 50–57, Nov 1998.

[41] V. Krishnaswamy, M. Raynal, D. Bakken, and M. Ahamad. Shared State Consistency for Time-Sensitive Distributed Applications. In *Proc. of the Intl. Conference on Distributed Computing Systems*, pages 606–614, April 2001.

[42] G. Kuenning, R. Bagrodia, R. Guy, G. Popek, P. Reiher, and A. Wang. Measuring the Quality of Service for Optimistic Replication. In *Proc. of the ECOOP Workshop on Mobility and Replication*, July 1998.

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

[44] L. Lamport. Time, Clocks, and the Ordering of Events in Distributed Systems. *Communications of the ACM*, 21(7):558–565, July 1978.

[45] T. Lawrence. Quality of Service: A Model for Information. In *Proc. of the IEEE Workshop on Object-Oriented Real-Time Dependable Systems*, pages 180–183, Jan 1999.

[46] B. Li. *Agilos: A Middleware Control Architecture for Application-Aware Quality of Service Adaptations*. PhD thesis, University of Illinois, 2000.

[47] M. Little. *Object Replication in a Distributed System*. PhD thesis, University of Newcastle upon Tyne, September 1991.

[48] J. Liu, K. Lin, R. Bettati, D. Hull, and A. Yu. Use of Imprecise Computation to Enhance Dependability of Real-Time Systems. In *Proc. of the Foundations of Dependable Computing: Paradigms for Dependable Applications. Koob and Lau, Ed*. Kluwer Academic, 1994.

[49] D. Malkhi, M. Reiter, and R. Wright. Probabilistic Quorum Systems. In *Proc. of the 16th ACM Symposium on Principles of Distributed Computing*, August 1997.

[50] Microsoft. Microsoft Windows Active Directory: An Introduction to the Next Generation Directory Services. Technical report, Microsoft Corporation, 1999. `msdn.microsoft.com/library/backgrnd/html/msdn_actdirintro.htm`.

[51] Microsoft. Windows 2000 Server Resource Kit. Technical report, Microsoft Press, 2000. `www.microsoft.com/technet/default.asp`.

[52] P. Mockapetris and K. Dunlap. Development of the Domain Name System. In *Proc. of the ACM SIGCOMM*, August 1988.

[53] C. Mohan, R. Barber, et al. Evolution of Groupware for Business Applications: A Database Perspective on Lotus Domino/Notes. In *Proc. of the 26th VLDB Conference*, pages 684–687, September 2000.

[54] S. Mohapatra and N. Venkatasubramanian. A Distributed Adaptive Scheduler for QoS Support in Compose|Q. In *Proc. of the IEEE Workshop on Object-Oriented Real-Time Dependable Systems*, pages 326–333, January 2002.

[55] L. Moser, P. Melliar-Smith, and P. Narasimhan. A Fault Tolerance Framework for CORBA. In *Proc. of the IEEE Intl. Symp. on Fault-Tolerant Computing*, pages 150–157, June 1999.

[56] S. Mullender, editor. *Distributed Systems*. Addison-Wesley, second edition, 1995.

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

[58] OpenLDAP. *OpenLDAP 2.0 Administrator's Guide*. `www.openldap.org`.

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

[60] C. Pu and A. Leff. Replica Control in Distributed Systems: An Asynchronous Approach. In *Proc. of the ACM SIGMOD Intl. Conference on Management of Data*, pages 377–386, May 1991.

[61] S. Raman and S. McCanne. A Model, Analysis, and Protocol Framework for Soft State-based Communication. In *In Proc. of the ACM SIGCOMM*, Cambridge, MA, September 1999.

[62] K. Ranganathan, A. Iamnitchi, and I. Foster. Improving Data Availability through Dynamic Model-Driven Replication in Large Peer-to-Peer Communities. In *Global and Peer-to-Peer Computing on Large Scale Distributed Systems Workshop*, May 2002.

[63] J. Ren. *AQuA: A Framework for Providing Adaptive Fault Tolerance to Distributed Applications*. PhD thesis, University of Illinois at Urbana-Champaign, 2001.

[64] Y. Ren, M. Cukier, and W. H. Sanders. An Adaptive Algorithm for Tolerating Value Faults and Crash Failures. *IEEE Transactions on Parallel and Distributed Systems*, 12(2):173–192, 2001.

[65] Y. (J.) Ren, T. Courtney, M. Cukier, C. Sabnis, W. H. Sanders, M. Seri, D. A. Karr, P. Rubel, R. E. Schantz, and D. E. Bakken. AQuA: An Adaptive Architecture that Provides Dependable Distributed Objects. *IEEE Transactions on Computers*. To appear.

[66] *Real-Time Specification for Java*. `www.rtj.org`.

[67] P. Rubel. Passive Replication in the AQuA System. Master's thesis, University of Illinois at Urbana-Champaign, 2000.

[68] C. Ryan et al. Evaluating Policies and Mechanisms for Supporting Embedded, Real-Time Applications with CORBA 3.0. In *Proc. of the IEEE Real-Time Technology and Applications Symposium*, June 2000.

[69] C. Sabnis et al. Proteus: A Flexible Infrastructure to Implement Adaptive Fault Tolerance in AQuA. In *Proc. of the IFIP International Working Conference on Dependable Computing for Critical Applications*, pages 149–168, January 1999.

[70] Y. Saito and H. Levy. Optimistic Replication for Internet Data Services. In *Proc. of the 14th International Conference on Distributed Computing (DISC)*, Oct 2000.

[71] M. Satyanarayanan, J. Kistler, P. Kumar, et al. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers*, 39(4):447–459, April 1990.

[72] M. Sayal, Y. Breitbart, P. Scheuermann, and R. Vingralek. Selection Algorithms for Replicated Web Servers. In *Proc. of the Workshop on Internet Server Performance*, June 1998.

[73] U. Sayyid. *The ACE Programmer's Guide*. Hughes Network Systems. `www.cs.wustl.edu/~schmidt/ACE-papers.html`.

[74] D. Schmidt and F. Kuhns. An Overview of the Real-Time CORBA Specification. *IEEE Computer*, 33(6):56–63, June 2000.

[75] A. Singla, U. Ramachandran, and J. Hodgins. Temporal Notions of Synchronization and Consistency in Beehive. In *Proc. of the 9th Annual ACM Symp. on Parallel Algorithms and Architectures*, pages 211–220, June 1997.

[76] D. Terry. Towards a Quality of Service Model for Replicated Data Access. In *In Proc. of the 2nd Intl. Workshop on Services in Distributed and Networked Environments*, pages 118–122, June 1995.

[77] D. Terry, A. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. Welch. Session Guarantees for Weakly Consistent Replicated Data. In *Proc. of the Intl. Conference on Parallel and Distributed Information Systems (PDIS)*, pages 140–149, September 1994.

[78] D. Terry, K. Petersen, M. Speitzer, and M. Theimer. A Case for Non-Transparent Replication: Examples from Bayou. *In Proc. of the IEEE Intl. Conf. on Data Engineering*, pages 12–20, December 1998.

[79] F. Torres-Rojas, M. Ahamad, and M. Raynal. Timed Consistency for Shared Distributed Objects. In *Proc. of the ACM Symp. on Principles of Distributed Computing*, pages 163–172, May 1999.

[80] A. Vaysburd. *Building Reliable Interoperable Distributed Applications with Maestro Tools*. PhD thesis, Cornell University, May 1998.

[81] F. Wang, K. Ramamritham, and J. Stankovic. Determining Redundancy Levels for Fault Tolerant Real-Time Systems. *Special Issue of IEEE Transactions on Computers on Fault Tolerant Computing*, 44(2):292–301, February 1995.

[82] V. Wolfe, L. Dipippo, R. Ginis, M. Squadrito, S. Wohlever, and I. Zykh. Expressing and Enforcing Timing Constraints in a Dynamic Real-Time CORBA System. *Real Time Systems*, 16:253–280, 1999.

[83] H. Yu and A. Vahdat. Design and Evaluation of a Continuous Consistency Model for Replicated Services. In *Proc. of the 4th Symp. on Operating Systems Design and Implementation (OSDI)*, October 2000.

[84] H. Zou, N. Soparkar, and F. Jahanian. Probabilistic Data Consistency for Wide-Area Applications. In *Proc. of the 16th Intl. Conference on Data Engineering*, 1998.

Appendix A

We have so far described the protocols we have developed at the middleware layer in order to support time-sensitive clients that access replicated services. The research described in this thesis has resulted in the implementation of two main replica selection handlers in AQuA: 1) a handler for server applications that are either stateless or have static application state; and 2) a handler for server applications that have dynamic state. In our implementation, we call the first handler `Selection_Handler` and the second handler `Stateful_Selection_Handler`. Each of these handlers supports different protocols for providing different application guarantees. The `Selection_Handler` supports three protocols that implement different selection mechanisms: 1) probabilistic, 2) static, and 3) round-robin. The `Stateful_Selection_Handler` supports two protocols that provide different ordering guarantees: 1) sequential and 2) FIFO. An AQuA client or server application has to choose the appropriate handler and the appropriate protocol in order to communicate with another application. Furthermore, in order to provide a meaningful response, the client and the server application should choose the same handler and protocol to communicate with each other. In this appendix, we explain how an application can be configured dynamically to use the appropriate handler and protocol, so that the application can choose different guarantees for different sessions without having to be recompiled.

A.1 ACE Service Configurator

In Chapter 2, we mentioned that AQuA applications are implemented as CORBA objects using the TAO implementation of CORBA, which is part of the ACE communication framework [73]. Each AQuA application may be made up of different objects, with each object responsible for handling a specific

functionality. TAO relies on object factories to construct the required objects. The ACE service configurator [33] is a framework that can be used for creating dynamically configured applications. In order to configure an application, a programmer can supply directives that direct the service configurator to perform functions, such as loading or unloading an object, and starting or stopping the execution of an object that makes up an application. The directives may be supplied in a file or specified on the command line when the service configurator is invoked. In the next section, we explain the directives we used to configure the client and server applications we have targeted in our research. We first explain the `dynamic` directive supported by the ACE service configurator. The directive allows a programmer to configure applications dynamically and triggers the objects to be loaded and initialized from dynamically linked libraries. The following line shows the general format of the `dynamic` directive [33]:

`dynamic` *service_name base_object_type library:factory_method "options"*

The following is a brief explanation of the components of the `dynamic` directive.

service_name: the name used to register the newly created object with the service configurator.

base_object_type: the type of object returned by the factory method.

library: the name of the library containing the code for the object. On Unix systems, the naming convention is the same as the one supplied to the compiler using the `-l` option. The library should reside in a directory accessible to the application.

factory_method: the name of the method used to create a new instance of the object.

options: the list of options passed as an argument to the newly created object.

A.2 Configuration File

Figure A.1 is a sample configuration file that we have used to configure an AQuA server application accessed by time-sensitive clients, using the ACE service configurator. Each server replica has its own

```
1: dynamic AQUA_Starter Service_Object*
   AQUA_Base:_make_AQUA_Starter_Factory()
   "dummy -x./server,1"

2: dynamic AQUA_Dynamic_Send Service_Object*
   AQUA_Dynamic_Send_Handler:_make_AQUA_Dynamic_Send_Handler_Factory()
   "dummy -aserver -hProteus"

3: dynamic Timed_DII_Processor Service_Object*
   AQUA_Selection_Handler_Base:_make_Timed_DII_Processor() ""

4: dynamic AQUA_Selection_Group_Factory Service_Object*
   AQUA_Selection_Handler_Base:_make_AQUA_Selection_Group_Factory () ""

5: dynamic Sequential Service_Object *
   AQUA_Sequential:_make_AQUA_Sequential_Factory() ""

6: dynamic Stateful_Selection_Handler_Factory Service_Object *
   AQUA_Stateful_Selection_Handler:_make_AQUA_Stateful_Selection_Handler_Factory()
   "server  -f -m  -hclient -gPinger -tS"
```

Figure A.1: Example of a service configuration file

configuration file. The configuration file used to configure a time-sensitive client application in AQuA is similar, but has some minor differences. We explain those differences later in this section.

We now explain all of the directives listed in Figure A.1 in the order in which they appear. Note that the numbers that precede the directives are provided for explanation purposes, and are not part of the configuration file.

The first directive directs the ACE service configurator to dynamically load the library called `AQUA_Base` (actual file name on Unix is libAQUA_Base.so) and create a service object with the name `AQUA_Starter` using the factory method `_make_AQUA_Starter_Factory()`. The type of the object returned by the factory method is a pointer to the default `Service_Object`. The `AQUA_Starter` object is an instance of a service that can load a specified application process before the associated AQuA gateway handlers are loaded. The name of the application and the arguments passed to the application are specified as options. In our example above, the `AQUA_Starter` object loads an application called `server`, which is the name of our test server application. All the replicas of an application are configured using the same application name. The numerical argument passed to the

Table A.1: Directives for loading gateway protocols for time-sensitive applications in AQuA

Gateway communication protocol	Type of application state	Directive for loading
sequential ordering	dynamic	`dynamic Sequential Service_Object * AQUA_Sequential:_make_AQUA_Sequential_Factory() ""`
FIFO ordering	dynamic	`dynamic Fifo Service_Object * AQUA_Fifo:_make_AQUA_Fifo_Factory() ""`
probabilistic selection	static	`dynamic Probabilistic Service_Object * AQUA_Probabilistic:_make_AQUA_Probabilistic_Factory() ""`
`static_all` selection	static	`dynamic Static Service_Object * AQUA_Static:_make_AQUA_Static_Factory() ""`
round robin selection	static	`dynamic RoundRobin Service_Object * AQUA_RoundRobin:_make_AQUA_RoundRobin_Factory() ""`

server application is the seed used to initialize the random number generator, which is used to generate the samples of the appropriate service time distribution. Each replica of the server application uses a different seed, as that would result in some variability in the service times across the different replicas. In our example, the replica uses a seed of value 1.

The second directive creates a *dynamic send* handler object. The dynamic send handler allows the application object created by the previous directive to communicate with another application by joining the group of the peer application at the time of communication. The application object ceases to be in the peer group once the communication is completed. This handler is mainly used for event notification purposes, such as sending a notification of a crash failure or reporting the need for additional replicas to the dependability manager, as explained in Chapter 7. The values of the different components of this dynamic directive can be interpreted in the same way as those for the previous directive. The options passed to the handler's factory specify that the newly created dynamic send handler will be used by an instance of an application called `server` to send notifications to the Proteus group, which is the designated group for the dependability managers in AQuA.

The third directive loads the *DII processor* for time-sensitive applications. It is one of the components of the gateway handler and is responsible for placing an incoming request in the request queue and delivering the request to the application object created by the first directive. It is also responsible for measuring the service time and queuing delay for a request. The DII processor is mainly used on the server side. The option list indicates that no arguments are passed to the timed DII processor.

The fourth directive loads an instance of the replication group. It allows the newly created application object to join the appropriate Ensemble group. All replicas of a service join the same replication

Table A.2: Configuration options for time-sensitive applications

Option	Type of application	Type of application state	Description
`-f`	server	static, dynamic	indicates that the replicated server object is the first member of its replication group.
`-m`	server	dynamic	indicates that the replica should join the primary group. The default is to place the replica in the secondary group.
`-h`$< peer\_application\_name >$	client, server	static, dynamic	indicates the name of the peer application. Specification of this option is mandatory for both clients and servers.
`-g` $< qos\_group\_name >$	client, server	static, dynamic	indicates the name of the QoS group containing the clients and the server replicas. This group is used by the clients to send their requests and receive performance updates from the servers.
`-c`	client	static, dynamic	indicates that the object is joining the QoS group as a Maestro client. If this option is omitted, the object is considered a Maestro server.
`-d`$< deadline >$	client	static, dynamic	indicates the client's deadline.
`-p`$< probability >$	client	static, dynamic	indicates the probability of timely response requested by the client.
`-a`$< staleness >$	client	dynamic	indicates the client's staleness threshold.
`-r` $< read\_only\_list >$	client	dynamic	indicates the list of comma-separated, read-only methods that the client invokes.
`-s` $< selection\_mechanism >$	client, server	static, dynamic	indicates the selection mechanism. The selection mechanism can take on one of the following values; the probabilistic mechanism is the default. p, P - Probabilistic s, S - Static r, R - Round-Robin
`-t` $< order >$	client, server	dynamic	indicates the ordering guarantee. The ordering guarantee can take on one of the following values: s, S - Sequential f, F - FIFO

group. If an application is not replicated, the object will be the sole member of its replication group.

The fifth directive loads an instance of the gateway communication protocol that provides sequential ordering guarantees. Table A.1 lists all the gateway communication protocols that resulted from the research described in this thesis, along with the directives used to load the protocol objects. The second column of Table A.1 indicates whether the protocols are used in the context of static state applications or dynamic state applications.

Finally, the sixth directive creates an instance of the gateway handler that is used by stateful, time-sensitive applications. A similar directive is used to create a gateway handler that selects replicas for time-sensitive applications that have static state; however, the `Stateful_Selection_Handler` is replaced by `Selection_Handler`. The option list includes the arguments that are passed to the factory method that creates the new handler object. The first argument is the name of the application, which in our example is `server`. Table A.2 lists all the options supported by the factory methods that create the handler object for the static and dynamic states. Some of the options are common to both

the client and server applications, while others are specific to either clients or servers, as the second column of Table A.2 indicates. For example, the `-m` option, which identifies a replica as a primary group member, is specified for a server application; the QoS options, such as `-d`, which specifies the deadline, are used only by clients. On the other hand, the `-h` option, which identifies the peer application, is specified by both client and server applications. The third column identifies the options that are specific to the dynamic state handler and those that are used by both the static and dynamic handlers. For example, the `-m` option is used only in the case of dynamic state, while the `-h` option is supported by both the static and dynamic state handlers.

We can now use Table A.2 to interpret the sixth directive in Figure A.1, which creates a handler for a server application that has dynamic state. The directive specifies that the handler is used by an instance of the application called `server` to communicate with a peer application called `client`, which is the name of our test client application. Further, the application object created by the first directive in Figure A.1 is the first member of its replication group; it is also a member of the primary group of servers, and has to join the QoS group called `Pinger`. The directive also specifies that the server application provides sequential ordering guarantees. One has to ensure that the ordering guarantee specified using the `-t` option is consistent with the communication protocol loaded in the handler. For example, if the `-t` option specifies sequential ordering, the file has to include a directive to load the communication protocol that provides sequential ordering, as shown by directive 5 in Figure A.1.

We now mention the changes we need to make to the configuration file shown in Figure A.1 in order to configure a client application. First, the name of the server application must be replaced by the name of the client application. Second, the arguments specified in the option list of the first directive must be replaced by the arguments passed to the client application, if any. Finally, the sixth directive has to be modified to specify the client-related options, as shown by the following example.

```
dynamic Stateful_Selection_Handler_Factory Service_Object *
AQUA_Stateful_Selection_Handler:_make_AQUA_Stateful_Selection_Handler_Factory()
"client -c -hserver -gPinger -d120 -p0.9 -a2 -tS -rread_only"
```

The above example creates a handler for a time-sensitive client application called `client`, which

communicates with a server application called `server` that has dynamic replicated state. The client object joins a QoS group called `Pinger`, which allows it to send its requests to the server replicas and receive performance updates from them. The directive specifies that the client's deadline is 120 milliseconds, the probability of timely response is 0.9, and the staleness threshold is 2. Further, the client uses the sequential ordering protocol to communicate with the server. The only read method that the client invokes on the server is a method called `read_only`. Any other method invocation may involve updates to the server state.

Vita

Sudha Krishnamurthy received a B.S degree in 1990, majoring in Physics, Chemistry, and Mathematics; an M.E. degree in Computer Engineering from the Indian Institute of Science, Bangalore, India, in 1994; and her Ph.D. degree in Computer Science from the University of Illinois, Urbana-Champaign, in 2002.

At the master's level, she worked on research projects in the area of compilers. Subsequently, she worked on research projects that involved developing and experimentally analyzing communication protocols for cluster-based systems. She worked briefly as a graduate research assistant in the cluster computing group at NCSA. She joined the PERFORM research group in the Coordinated Science Laboratory at the University of Illinois as a graduate research assistant in January 2000 in order to pursue her doctoral degree. Her research interests are in the area of high-performance and dependable distributed computing. She is a member of the Phi Kappa Phi honor society. She is also a member of IEEE.