An Adaptive Quality of Service Aware Middleware for Replicated Services

Sudha Krishnamurthy, William H. Sanders, Michel Cukier

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 (QoS) is degraded due to uncertain variations in load and unanticipated failures. In this paper, 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 timeliness and consistency constraints. We also propose an adaptive framework that dynamically selects replicas to service a client's request based on the prediction made by probabilistic models. These models use the feedback from online performance monitoring of the replicas to provide probabilistic guarantees for meeting a client's QoS specification. The experimental results we have obtained demonstrate the role of feedback and the efficacy of simple analytical models for adaptively sharing the available replicas among the users under different workload scenarios.

Index Terms: replica consistency, middleware, quality-of-service, timeliness, probabilistic modeling.

I. Introduction

Our motivation for building a QoS-aware middleware stems from two main observations. First, distributed systems have different degrees of uncertainty arising from factors, such as transient overloads and failures. Second, different distributed applications have diverse requirements. Hence, it is useful to design middleware-based solutions for sharing access to distributed services based on the QoS requirements of the clients. In our work, we target *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 those 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

This research has been supported by DARPA contract F30602-98-C-0187.

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.

The approach we use for providing fault-tolerant and responsive services makes use of replication. Replicating the servers provides robustness in times of failure by allowing access to a 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., [1], [9], [20], [5], [21]). 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 [3], [7]. 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 immediate convergence of replicated state by forcing all the replicas to commit the modifications at the same time (e.g., [2], [24], [21], [25]). However, such a strategy that ensures strong replica consistency limits the degree of concurrency and results in reduced responsiveness. On the other hand, in the weak consistency model (e.g., [27], [13], [8]), operations are performed on some subset of replicas, and the updates are propagated to the other replicas either lazily or on demand. Typically, 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., [6], [22]) 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 to only 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.

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 appropriate 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.

A. Paper Contributions

To address the above issues for managing replicated resources, we have developed a middleware-based framework that allows us to construct customized protocols tailored to the semantics of specific applications. We have implemented this framework in AQuA, a CORBA-based middleware that supports transparent replication of objects across a LAN [24]. The framework we have built uses simple analytical models to establish a relationship between a client's QoS specification and properties of the replicas. In [14] and [15] we described an adaptive replica allocation scheme that uses a probabilistic approach for providing temporal guarantees to the clients in two different cases: 1) when the replicated state is static, and 2) when the replicated state is dynamic. In the first case, we assume that the replicas are always consistent and therefore do not address the issue of maintaining replica consistency. This is useful in applications such as compute servers, search engines and directory servers, which mainly export interfaces for information retrieval. In the second case in which the replicated state is time-varying, some of the replicas may have obsolete state. We target time-sensitive applications that can tolerate a certain

degree of relaxed consistency in exchange for better response time and express their timeliness and consistency requirements in the form of a QoS specification. 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 developed an adaptive framework that supports tunable consistency and timeliness. Some of the applications that motivate the need for such a framework include real-time database applications, such as electronic patient recording systems and ticket reservation systems. In this paper, we compare and contrast the replica selection approaches used for the static and dynamic replicated states, and present additional experimental results that extend our earlier performance evaluation [16], [17].

B. Paper Organization

The remainder of this paper is organized as follows. In Section II, we describe our QoS model that allows a broad spectrum of applications to express their timeliness and consistency requirements. Section III provides a brief overview of the AQuA architecture. In Section IV we describe the replica organization that allows us to build protocols for providing different consistency guarantees and to use them on demand. These protocols use a combination of immediate and lazy update propagation to ensure that the states of the replicas do not diverge in an unacceptable manner. As specific examples, we describe the protocols we have implemented that allow the replicated services to provide sequential and FIFO ordering guarantees. In Section V we compare the probabilistic approach that uses the performance history of the replicas to predict the ability of the replicas to meet a client's QoS requirement, for static and dynamic replicated states. In Section VI we summarize the algorithms that use the prediction made by the probabilistic models to select replicas to meet the QoS requirements of the clients. In Section VII, we present experimental results. Finally, we discuss ideas for future extensions in Section VIII and present our conclusions in Section IX.

II. QoS Model for Accessing Replicated Services

Our QoS model allows a broad spectrum of applications to express their requirements at a fairly high level of abstraction using a uniform interface. Applications may either specify their QoS requirements at start-up time or negotiate them at runtime as often as they want. In order to distinguish invocations that modify the state of an object from those that merely retrieve state, 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. 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. 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.

A. Timeliness

Time-sensitive applications require timely execution of operations and timely responses to their requests. However, due to the uncertainty in the distributed environment, it is impossible to provide deterministic guarantees for meeting the temporal requirements. Instead, our goal is to provide probabilistic temporal guarantees. To achieve this, our QoS model allows a client to specify its temporal requirements as 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 read request, and the minimum probability with which it expects its temporal constraint to be met. Failure to meet a client's response time constraint results in a *timing failure* for the client. 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.

B. Consistency

Replica inconsistency may arise when multiple clients access an object concurrently, as 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. Since different applications have different views of consistency, it is hard to capture the different consistency requirements using a single metric. 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 [26], [30], [23].

Several researchers have extended traditional consistency models by incorporating the notion of time in order to bound the degree of inconsistency. For example, the notion of *epsilon-serializability* (defined in [23]), and timed consistency models (defined in [28], [18]), 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 TACT middleware [30] is another related work that attempts to provide a middleware framework

for tunable consistency and availability. 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 for propagating the writes among the replicas.

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 of 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 [2], [4]. In our work, 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 our framework, the staleness of a response denotes the staleness of the state of the replica that sent the response. 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. We compute the staleness of a replica by associating a timestamp with each update operation. We use timestamps based on "logical clocks" [19] 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." 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.

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 it wishes to obtain a copy of the document that is no more than 5 versions old within 2.0 seconds with a probability of at least 0.7. 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.

III. Overview of AQuA

We now briefly describe the AQuA architecture. AQuA enhances the capabilities of CORBA objects by transparently replicating the objects across a LAN. A *dependability manager* manages the replication level for different applications based on their dependability requirements. Replicas offering the same service are organized into a group. Communication between members of a group takes place through the Maestro-Ensemble group communication layer [29], [11], above which AQuA is layered. The use of group communication in AQuA is transparent to the end applications. Hence each of the clients, which are all CORBA objects, is given the perception that it is communicating with a single server object using CORBA's remote method invocation, although the client's request may be processed by multiple 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. While previous work in AQuA has focused on gateway handlers for providing fault tolerance using the active and passive handlers [24], we have enhanced AQuA by developing gateway handlers that provide tunable consistency and timeliness guarantees for time-sensitive applications.

IV. Hierarchical Replica Organization

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 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.

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. The QoS group allows clients and servers to exchange messages and it allows the servers to publish their performance updates to the client subscribers. In our implementation, all of these groups are derived from Maestro groups [29], and members of a group communicate with each other by making use of the Maestro-Ensemble group communication protocol [11], above which AQuA is layered. For each group, Ensemble elects

one of the members of the group as the *leader*. 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 different 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 semantics. 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., [24], [21]), in which all the replicas implement strong consistency semantics. On the other hand, by placing one of the replicas in the primary group, and all the remaining replicas in the secondary group, one can implement a primary/backup protocol with multiple backup replicas.

In the case of static replicated state in which the servers permit only read transactions, the primary group is empty and we place all the replicas offering a service in the secondary group. However, in the case of dynamic replicated state, we organize the replicas into the primary and secondary tiers. This 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.

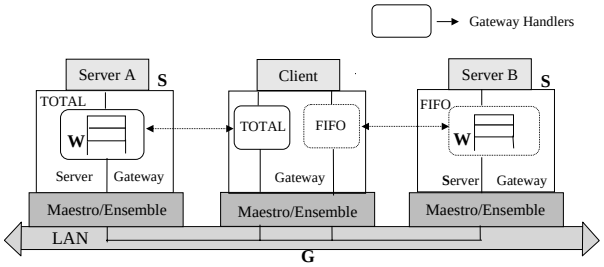


Fig. 1

TIMED CONSISTENCY HANDLERS IN THE AQUA GATEWAY

A. Ordering Guarantees

We now describe how we maintain consistency across the replicas in the case of dynamic state. As mentioned in Section II, 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 1. We have implemented gateway handlers that provide sequential and FIFO ordering. The sequential handler was motivated by applications, such as document-sharing applications, in which all the clients access a common replicated state, and the servers globally order the requests of the clients in order to prevent conflicts. On the other hand, the FIFO handler, which provides weaker consistency, was designed to support applications, such as banking transactions, in which the replicated servers maintain states that are specific to each client. A client can communicate with a replicated service by using the gateway handler appropriate for the service. For example, Figure 1 shows a client communicating with Service A using a sequential handler and with Service B using a FIFO handler. We have designed the protocols to ensure that the ordering guarantees are provided even when replica failures occur. The symbols S, W, and G in Figure 1 represent different performance parameters of our probabilistic model. They will be elaborated later in Section V-C.

We now compare and contrast the sequential and FIFO handlers with respect to the way they service a client's update and read-only requests. In case of both handlers, a client's update request is forwarded by the client gateway handler to all the primary replicas. 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 this member the *lazy publisher*. When a client invokes a read-only request, the client gateway handler forwards the request to a subset of primary and secondary replicas. In Section VI, we will describe the selection of this subset. A selected replica responds

to the read request immediately, if its most recently updated state is no more than x versions old, where x is the staleness threshold specified by the client in its QoS specification. In other words, the replica performs an *immediate read* operation if it meets the staleness specification of the client. However, a secondary replica may have a state that is more stale than the staleness threshold specified by the client. The reason being that the secondary replicas update their state only upon receiving the state update from the lazy publisher. In such a case, the replica performs a *deferred read* by buffering the read request and responding to the client upon receiving the next state update from the lazy publisher.

The sequential and FIFO handlers differ in the order in which the replicas commit the updates and the manner in which a replica determines if its state meets the staleness threshold specified by a client. In the sequential consistency case, all the replicas see the effects of the updates in the same sequential order. The order in which the replicas commit updates is determined by the *Global Sequence Number (GSN)* of the update operation, which is assigned by the leader of the primary group and broadcast by the leader to the other primary replicas. The leader merely serves as the *sequencer* and does not actually service the client's request. The value of the GSN at any instant of time may be considered to be the value of the leader's logical clock. For read-only operations, this GSN serves as the basis for determining the staleness of a replica. In contrast, 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 along with their invocations. 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.

Failure handling is another point in which the FIFO handler differs from the sequential handler. 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. 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 that were sent before the crash. Hence, to ensure sequential ordering, the new sequencer begins by assigning the GSN to the pending requests, if any, before making the GSN assignment for the newly incoming requests. Failure handling for FIFO ordering is relatively simpler because the FIFO handler does not use a dedicated sequencer. Hence, the FIFO handler has to handle only the failure of the lazy publisher. In the case of FIFO protocol, the leader of the primary group is designated as the lazy publisher. When a new leader is elected by Ensemble to replace a failed leader, the leader-elect takes over as the new lazy publisher by first propagating its state to the secondary replicas. It then schedules the subsequent lazy updates with the appropriate frequency, and then continues to service the requests from the clients.

V. Probabilistic Modeling of the Response Time Distribution

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. As mentioned in Section II, 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; and the minimum probability of meeting this constraint, $P_c(d)$. In the case of dynamic replicated state, the client also specifies 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, one of the important responsibilities of the client gateway handlers is to select an appropriate subset of replicas that can deliver a timely and consistent response to the clients, thereby reducing the occurrence of timing failures.

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 mul-

ticasting 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 IV-A. 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 first present the probabilistic model we have developed for the static replicated state and then describe how we extend it for the dynamic state.

A. Static State

Let M be the set of replicas offering the service requested by a client and R_i be the random variable denoting the time to receive a response from a replica $i \in M$, after a request was transmitted to it. 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. We denote this probability by $P_K(d)$. 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 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 experimental results we obtained show that the resulting model makes reasonably good predictions most of the time [16], [17]. We use the independence assumption to compute the probability, $P_K(d)$, for the replicas in subset K, as follows:

$$\begin{aligned} P_K(d) &= 1 - P(\text{no replica in } K \text{ responds before d}) \\ P_K(d) &= 1 - \prod_{i \in K} P(R_i > d) \\ P_K(d) &= 1 - \prod_{i \in K} (1 - F^I_{R_i}(d)) \end{aligned} \tag{1}$$

where $F^I_{R_i}(d)$ is the response time distribution function for replica i, under the condition that the replica responds to the request without waiting for a state update.

B. Dynamic State

We now explain how we extend the above model to take into account the state of the replica when estimating its responsiveness. Let t denote 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. Let $A_i(t)$ denote the staleness of the state of replica i at time t, and $P(A_i(t) \leq a)$ be 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. Let $P(R_i \leq d)$ be the probability that a response from replica i will be received by the client within the client's deadline, d. As before, let $P_K(d)$ be 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.

We now derive the expression for $P_K(d)$ Unlike the static case, which made the selection from a single tier of replicas, the set K in the case of dynamic state 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

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

As in the case of the static state, we assume that the response times of the replicas are independent because they process their requests independently. Thus, using the independence assumption, we obtain

$$P_K(d) = 1 - \left[P(\text{no } i \in K_p \ni R_i \leq d) \cdot P(\text{no } j \in K_s \ni R_j \leq d)\right] \tag{2}$$

B.1 Primary Replicas

In Section IV-A, 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

$$P(\text{no } i \in K_p \ni R_i \leq d) = \prod_{i \in K_p} P(R_i > d) = \prod_{i \in K_p} (1 - F^I_{R_i}(d)) \tag{3}$$

where $F^I_{R_i}$, as in the case of the model for static state, 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.

B.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 IV-A, the replica has to 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, the client gateway uses a probabilistic approach to estimate the staleness of the secondary replicas. 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.

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

$$P(\text{no } j \in K_s \ni R_j \leq d) = \Big[\prod_{j \in K_s} (1 - F^I_{R_j}(d))\Big] \cdot P(A_s(t) \leq a) + \Big[\prod_{j \in K_s} (1 - F^D_{R_j}(d))\Big] \cdot (1 - P(A_s(t) \leq a)) \quad (4)$$

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 lazy state update. 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.

B.3 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

$$P(A_s(t) \leq a) = P(N_u(t_l) \leq a) = \sum_{n=0}^{a} \frac{(\lambda_u t_l)^n e^{-\lambda_u t_l}}{n!} \quad (5)$$

The sequential and FIFO handlers differ slightly in the way they evaluate the update arrival rate, λ_u. In sequential ordering, the replica state is shared by all the clients, and therefore λ_u is the rate at which updates are received by the primary replicas from all the clients. However, in the case

of FIFO ordering, since the updates to the replicated object are specific to the individual replicas, λ_u is the rate at which the client that is making the replica selection updates the replicated object. In either case, the staleness of the secondary replicas can be determined probabilistically if we know the arrival rate of the update requests and the time elapsed since the last lazy update. We measure those two parameters at runtime by instrumenting the gateway handlers and we have explained this in detail in [15]. 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 3, 4, and 5 in Equation 2 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.

C. Evaluating the Response Time Distribution

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 make use of the performance history recorded by online performance monitoring to compute the value of the distribution function for a replica i. In the case in which a replica can respond to a request without waiting for a state update, the response time random variable for a replica i is given by Equation 6:

$$R_i = S_i + W_i + G_i \tag{6}$$

For a deferred read, in which 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 7:

$$R_i = S_i + W_i + G_i + U_i \tag{7}$$

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; and U_i is the duration of time the replica spends waiting for the next lazy update. 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. These three parameters are depicted in Figure 1 by the terms S, W, and G, respectively.

For each read request, we experimentally measure the values of the above performance parameters by instrumenting the gateway handlers. The values of S_i, W_i, and U_i for a read request are measured by the server-side handler. The server handler then publishes the new measurements

to all the clients. The value of the two-way gateway delay, G_i, is measured by the client-side handler when it receives a response from replica i. For each replica, the client handlers record the most recent l measurements of these parameters in separate sliding windows in an information repository that is local to each client. The size of the sliding window, l, is chosen so as to include a reasonable number of recent requests, while eliminating obsolete measurements. The details of the gateway instrumentation and online performance monitoring are provided in [15].

Given that we can measure the performance parameters and record them at runtime, we can now compute the value of the distribution function for a replica 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 $F^I_{R_i}(d)$. 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.

VI. Replica Selection Algorithm

Given the ability to predict the probability that an individual replica will meet a client's time constraint based on the replica's state, we designed two algorithms that use this prediction to select a set of replicas that can meet the time constraint with the probability the client has requested. We call these two algorithms BEST_PROBABILITY_FIRST and LEAST_USED_FIRST and we presented the former in [14] and the latter in [15]. The selection algorithms are executed by each 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 sends the request to all the primary replicas. In this section, we summarize these algorithms and highlight their key differences. The BEST_PROBABILITY_FIRST algorithm selects replicas in decreasing order of the probability that they can individually meet the client's response time requirement. It includes just enough replicas in K such that the condition $P_K(d) \geq P_c(d)$ is satisfied, where $P_K(d)$ is computed using the models presented in the previous section. The LEAST_USED_FIRST algorithm, on the other hand, selects replicas in decreasing order of their elapsed time of response (ETR). The ETR of a replica is the duration that has elapsed since a reply was last received by the client from that replica, and is measured at runtime by instrumenting the gateway handler on the client side. Like the BEST_PROBABILITY_FIRST algorithm, the LEAST_USED_FIRST algorithm includes just enough replicas in K such that the condition $P_K(d) \geq P_c(d)$ is satisfied. Both algorithms are designed

to choose replicas in such a way that the QoS requirements of the client can be met even if one of the selected replicas fails before responding.

The BEST_PROBABILITY_FIRST algorithm is a greedy algorithm because it always picks the best replicas first. While, this results in a smaller subset of replicas, it also increases the potential for the occurrence of hot-spots due to the following reason. 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 performance information is broadcast to all the clients and the gateway delays of different client-replica pairs are not significantly different in a LAN, the information repositories of different clients may contain almost identical performance histories for the replicas. That may cause the clients to select the same or common replicas for their requests, resulting in hot-spots. In the case of the LEAST_USED_FIRST algorithm, while the response time distributions of a replica, which are computed from the performance history, are nearly identical in all the client information repositories, the ETR information is specific to each client-replica pair and is likely to be different for different clients. That results in a more balanced utilization of the available replicas and thereby reduces the occurrence of hot-spots.

VII. Experimental Results

We have conducted experiments to study the overhead of the selection algorithm and studied the effectiveness as well as the adaptability of the probabilistic model under different workload scenarios for static and dynamic replicated states [16], [15]. We also experimentally analyzed the tradeoffs between timeliness and consistency, using the sequential and FIFO ordering handlers we implemented in AQuA [17]. All of our 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 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 [12]. We now summarize the key results of the experiments we have published previously and present additional experimental results that provide a further evaluation of our work.

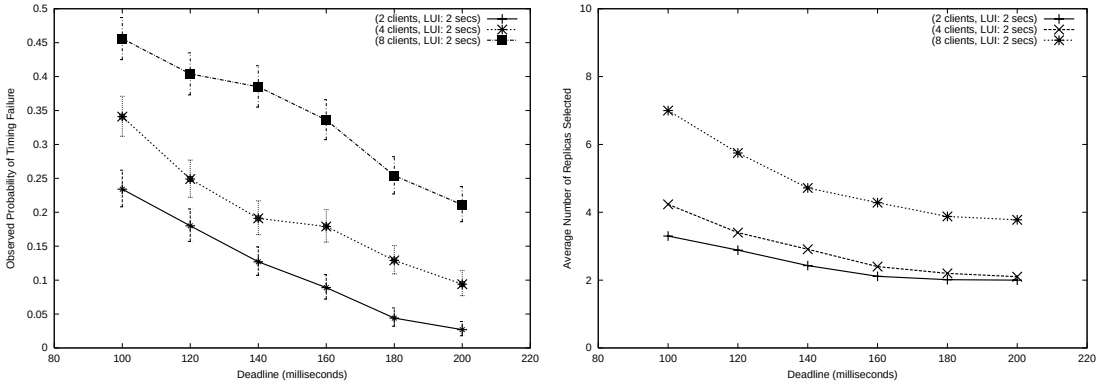
Our experiments showed that both the BEST_PROBABILITY_FIRST algorithm as well as the LEAST_USED_FIRST algorithm adapt to less stringent QoS requirements by choosing fewer replicas. The reason is that the algorithms use the model's prediction to select just enough replicas that can meet a client's QoS request, even if a replica failure should occur. 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, fewer replicas are needed to satisfy the request. We were also able to validate our models experimentally and show that while the observed probability of timing failures increases when the requested QoS is more stringent, the replicas selected by the model were able to maintain the observed failure probability to be within the threshold specified by the client, for the workloads we considered. To experimentally justify the need for a hierarchical replica organization, we compared the performance of a single-tier replica organization in which all the replicas were in the primary group with a two-tier organization in which 40% of the replicas were in the primary group with the remaining in the secondary group. The size of the primary group represents a tradeoff between the buffering delay due to deferred reads and queuing delay due to update requests. When there are more replicas in the primary group, a greater number of replicas have consistent state and therefore the buffering delays are smaller. However, since more replicas are involved in committing the updates, the queuing delays experienced by the read requests are higher. Our experiments showed that for smaller update rates, the single and two-tier replica organizations perform comparably. However, for larger update rates, it is possible to tune the lazy update interval (LUI) such that the two-tier scheme results in lower probability of timing failures compared to the single-tier scheme. The LUI is the periodicity with which the lazy publisher publishes its state to the secondary group of replicas. In Section VII-B, we will discuss the cost/performance tradeoffs associated with lazy updates in more detail.

A. Performance Under Load

We now describe the experiments we carried out to determine how well our model adapts to meet the client's QoS specification under different client-induced workloads. We present experimental results for the dynamic state using the sequential consistency guarantee, but we observed similar behaviour for the dynamic state with FIFO order and for static replicated state as well. Our experimental setup had 10 server replicas in addition to the sequencer, of which 40% were in the primary group and the remainder in the secondary group. The service time was normally distributed with a mean of 100 milliseconds and variance of 50 milliseconds. We present results obtained by varying two different parameters: 1) the number of clients accessing a service, and 2) the think time between the requests.

In the first case, the client-induced load increases with the number of clients accessing a service. Each client sent 1000 alternating update and read requests with a think time of 1000 milliseconds between successive requests. One of the clients specified a staleness threshold of



(a) Probability of timing failures with varying clients

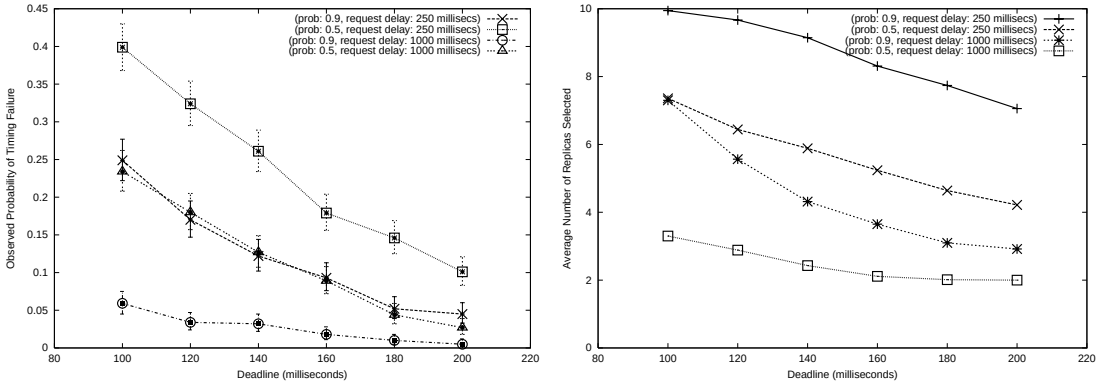
(b) Number of replicas selected with varying clients

Fig. 2

PERFORMANCE UNDER LOAD: MULTIPLE CLIENTS

value 2. It varied its deadline from 100 to 200 milliseconds and requested that its deadline be met with a probability ≥ 0.5. All of the remaining clients specified a staleness threshold of 4, deadline of 200 milliseconds, and requested that this deadline be met with a probability ≥ 0.1 in each run. The lazy update interval was 2 seconds. Figures 2a and 2b evaluate the performance of the probabilistic scheme using 2, 4, and 8 clients. Figure 2a 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 2b shows the average number of replicas selected by the probabilistic scheme to meet the QoS specifications of this client in each case. As expected, the observed timing failure probability increased as the number of clients requesting service increased, because of the higher queuing delays. However, we find that for the range of workloads we considered, the model was able to adapt appropriately to select a subset of replicas that could meet the client's QoS specification.

In the second case, when we varied the client-induced load by varying the think time, we used a constant number of clients in our experimental setup, which in our case was two. In all of the runs, Client1 specified a staleness threshold of 4, deadline of 200 milliseconds, and a minimum probability of timely response of 0.1. Client2 specified a staleness threshold of 2 and minimum probability of timely response of 0.9 in all of the runs, but varied its deadline from 100 to 200 milliseconds. The clients used different think times between their requests. The induced load on the servers was higher for smaller think times. Figures 3a and 3b present the results, using a



(a) Probability of timing failures with varying think time

(b) Number of replicas selected with varying think time

Fig. 3

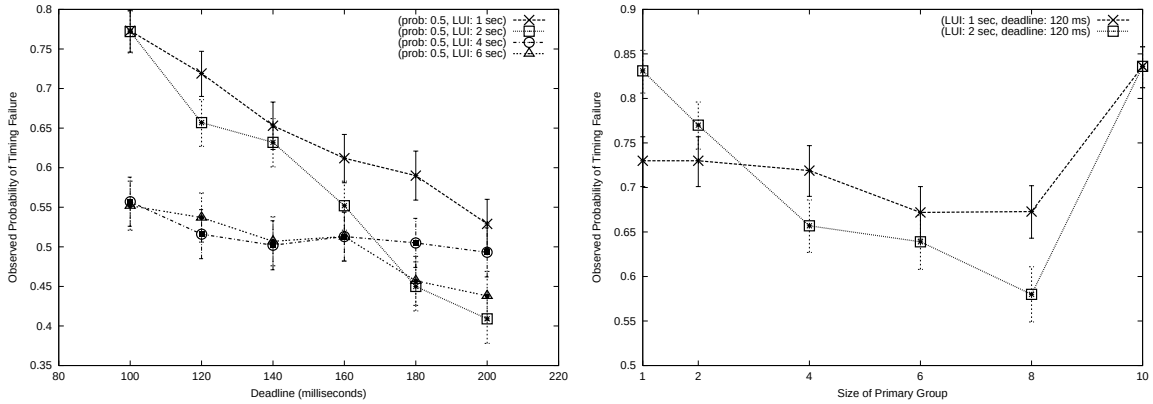
PERFORMANCE UNDER LOAD: VARIABLE THINK TIME

lazy update interval of 2 seconds, for two different values of the think time: 1000 milliseconds and 250 milliseconds.

The first observation from Figure 3a is that the observed failure probability increases as the think time reduces from 1000 milliseconds to 250 milliseconds. That is because 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. We also observe from the graphs in Figure 3b that as the queuing delay increases, the probabilistic scheme is sometimes unable to find enough replicas to meet the deadline with the probability requested by the client. For instance, when the think time is 250 milliseconds, the replica subset chosen by the probabilistic scheme is unable to meet deadline values $\leq$ 140 milliseconds with a probability $\geq$ 0.9, although the request is sent to all 10 available replicas. 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 either to renegotiate its QoS specification or to send its requests at a later time when the system is less loaded. Alternatively, the middleware can choose to create more replicas to meet the demand and we describe how we do that in AQuA, later in this section.

B. Cost/Performance Tradeoffs of Lazy Updates

We now look at the cost/performance tradeoffs associated with lazy update propagation. Our earlier results showed that increasing the frequency of lazy updates resulted in smaller buffering



(a) Impact of lazy update propagation

(b) Impact of the primary group size

Fig. 4

TIMELINESS/CONSISTENCY TRADEOFFS

delays for deferred reads, and thereby reduced the occurrence of timing failures [15]. 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. To study how the cost affects the performance, we repeated our experiments using a more intense workload than the one we had used in our earlier studies. In the following experiment, we used 4 clients, which is twice the value we used in our earlier study. The think time is 250 milliseconds, which is one-fourth the value we used in our earlier study. This resulted in a client update rate of 5 updates/second, which is nearly five times the update rate of our earlier experiments.

Figure 4a 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*, and stabilizes around 4 seconds, which is in contrast to the behavior we observed under the less intense workload in our earlier study. 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 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.

C. Impact of the Primary Group Size

We now present experimental results that show how the size of the primary group impacts the performance, for a given number of server replicas. We used the same experimental setup with 10 servers and 4 clients having a think time of 250 milliseconds, as in the previous experiment. We used two different values of LUI: 1 second and 2 seconds. Figure 4b shows the probability of 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). From Figure 4b, 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. Those observations can be explained as follows.

As mentioned earlier, 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 4b, 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 groups according to the workload.

Another observation from Figure 4b 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 when we analyzed the result shown in Figure 4a, which used 40%

of replicas in the primary group. In effect, Figure 4b shows that as we increase the size of the primary group, it may be more beneficial to reduce the frequency of lazy updates, because a larger fraction of the replicas are consistent.

D. Time-Varying Workload

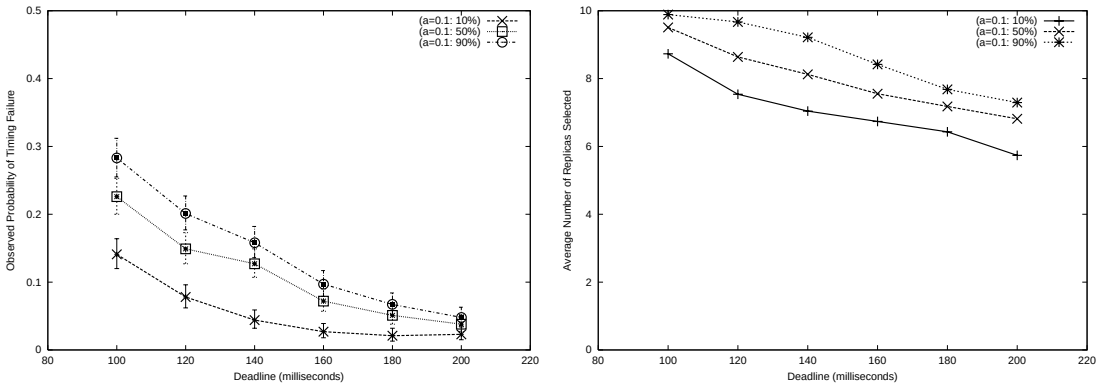
In the experiments we presented so far, the service time was normally distributed and the mean service time was stationary. The results we presented showed that the probabilistic scheme was able to use the performance history of the replicas effectively and adapt the selection of replicas to meet the QoS requested by the clients, when the service time distribution was stationary in the stochastic sense. We now discuss the experimental evaluation of our probabilistic framework using a time-varying workload. In the experiments we present below we used a workload that showed *heavy-tailed* properties. This was motivated by the evidence that the workloads in many well-known distributed services exhibit *heavy-tailed* distribution [10]. A heavy-tailed distribution is characterized by high variability and is defined as follows:

Heavy-Tailed Distribution: A random variable X follows a heavy-tailed distribution with a tail index α if $P[X > x] \backsim x^{-\alpha}, 0 < \alpha < 2$. The variance increases as α decreases. 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 [10]. 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

$$f(x) = \frac{\alpha \cdot k^{\alpha}}{1 - (k/p)^{\alpha}} x^{-\alpha-1}, k \leq x \leq p$$

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 significantly less than 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. 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.



(a) Probability of timing failures

(b) Number of replicas selected

Fig. 5

PERFORMANCE USING A TIME-VARYING WORKLOAD

As before, we used two clients, Client1 and Client2, each of which sent 1000 alternating read and update requests with a think time of 250 milliseconds. 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. 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 $f\%$ of the requests, and repeated the cycle. Thus, the length of the cycle is 100 requests.

Figure 5 presents the observed probability of timing failures and the average number of replicas selected for Client2 for different values of f, for the sequential consistency case. The first observation from Figure 5a is that the probability of timing failures increased as the percentage of requests processed in the HIGH state increased from 10% to 90%. This is to be expected as an increasing percentage of requests experience higher service times with a mean close to 121 milliseconds. The second observation is that when the value of f increased, the observed fail-

ure probability exceeded 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 NORMAL 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 opinion 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 5, for the time-varying workload.

Having ruled out the first hypothesis, we considered a second possible explanation, which was that there are not enough replicas that can deliver a timely response with a probability ≥ 0.9 for smaller deadline values, under a higher workload. From Figure 5b 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. Therefore, we repeated the experiments in the case of time-varying workload by reducing the lazy update interval from 2 seconds to 1 second. That helped reduce the timing failure probability significantly. These results show that the model can adapt to a time-varying work-

load. However, under stringent demands, there may not be enough replicas available to meet the demands. In such cases, we can adapt by either propagating the lazy updates more frequently, so that we have more replicas with up-to-date state. Alternatively, we can increase the size of the available replica pool by creating more replicas on demand.

We now briefly describe how our middleware addresses the problem of creating replicas on demand. In order to support dynamic replica creation, the middleware needs to determine when, where, and how many replicas to create. In our approach, the replica selector in a client's gateway requests the dependability manager, which is one of the components of the AQuA middleware, 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. The new replica is placed on the least loaded host. The new replica joins the secondary group, but does not have the most up-to-date state initially. When the lazy publisher subsequently disseminates its state to the secondary group, the new replica inherits the correct state and henceforth, begins to service a client's request.

VIII. Future Extensions

Our work motivates some interesting avenues for future work. First, 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 intuitive. It is easy to extend our framework so that the clients can replace the probability of timely response with a higher-level specification, such as the "importance" level, which takes on an integer value between 0 and 10. Alternatively, the client can specify the cost it is willing to pay for timely delivery. The middleware can then internally map these higher-level inputs to an appropriate probability value and perform adaptive replica selection as described.

Second, our middleware currently admits all the clients. If the observed timing failure probability exceeds a client's expectations, the middleware informs the client through a callback. The client can then renegotiate its QoS requirements. An alternative approach would be to incorporate some kind of admission control at the middleware layer, in order to determine which clients can be admitted based on the current availability of the replicas.

Third, we currently associate QoS attributes with read operations only. Although we allow different ordering guarantees for write operations, we do not currently support QoS requirements for write operations that can be specified at runtime. Our work can be enhanced to incorporate

write-specific QoS attributes, such as the maximum tolerable delay in propagating an update to a specified fraction of replicas.

Finally, in a large 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. That may result in a greater degree of inaccuracy in the performance histories. Hence, in order to extend our work to large scale networks, we need a way to track the performance histories of the replicas in a scalable manner. One way to address this issue is by organizing the replicas into groups, based on their geographic proximity, and propagating the performance updates to the clients in such a way that clients that are closer to a replica group can track the performance information of the replicas in that group more accurately. At the time of replica selection, the inaccuracy in the performance histories can be factored in by associating the response time distribution functions of the replicas with a weight that is proportional to their accuracy.

IX. 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. It actively monitors the replicas at runtime and uses the feedback to guide the adaptation. 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 selection 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 probabilistic model-based replica selection scheme we have developed, would be useful in an environment in which time-sensitive clients that have different QoS requirements access servers that display significant variability in their response times. Our experiments also helped in understanding the tradeoffs between timeliness and consistency for different consistency semantics and our results 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.

Although our probabilistic approach was mainly developed to adaptively share 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 that rely on feedback-based control are likely to play an increasing role in solving a range of problems related to building dependable systems.

Acknowledgments: We are thankful to the anonymous reviewers for their detailed feedback, which helped us to improve our work. We thank the rest of the AQuA team for their contributions to the AQuA project. We are thankful to Jenny Applequist for her editorial comments.

REFERENCES

[1] 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.

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

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

[4] G. V. Chockler, R. Vitenberg, and R. Friedman. Consistency Conditions for a CORBA Caching Service. In *Proc. of the Intl. Symposium on Distributed Computing (DISC)*, October 2000.

[5] 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.

[6] 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.

[7] 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.

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

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

[10] 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.

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

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

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

[14] S. Krishnamurthy, W. H. Sanders, and M. Cukier. A Dynamic Replica Selection Algorithm for Tolerating Timing Faults. In *Proc. of the International Conference on Dependable Systems and Networks*, pages 107–116, July 2001.

[15] S. Krishnamurthy, W. H. Sanders, and M. Cukier. An Adaptive Framework for Tunable Consistency and Timeliness Using Replication. In *Proc. of the International Conference on Dependable Systems and Networks*, pages 17–26, June 2002.

[16] S. Krishnamurthy, W. H. Sanders, and M. Cukier. Performance Evaluation of a Probabilistic Replica Selection Algorithm. In *Proc. of the Workshop on Object-Oriented Real-time Dependable Systems*, pages 119–127, January 2002.

[17] S. Krishnamurthy, W. H. Sanders, and M. Cukier. Performance Evaluation of a QoS-Aware Framework for Providing Tunable Consistency and Timeliness. In *Proc. of the International Workshop on Quality of Service*, pages 214–223, May 2002.

[18] 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.

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

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

[21] 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.

[22] 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.

[23] 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.

[24] 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*, pages 31–50, January 2003.

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

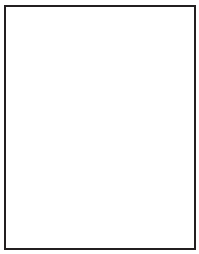
[26] D. Terry. Towards a Quality of Service Model for Replicated Data Access. In *In Proc. of the 2nd Intl. Workshop on Services in Distributed and Networked Environments*, pages 118–122, June 1995.

[27] 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.

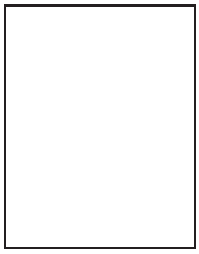
[28] F. Torres-Rojas, M. Ahamad, and M. Raynal. Timed Consistency for Shared Distributed Objects. In *Proc. of the ACM Symp. on Principles of Distributed Computing*, pages 163–172, May 1999.

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

[30] 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.

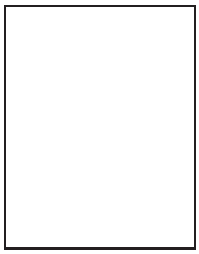


Sudha Krishnamurthy received her Ph.D. in Computer Science in 2002 from the University of Illinois, Urbana-Champaign. She is currently a research associate at the University of Virginia. Her research interests include the design and experimental analysis of software protocols for networked and wireless distributed systems.



William H. Sanders is a Professor in the Department of Electrical and Computer Engineering and the Coordinated Science Laboratory at the University of Illinois. He is Vice-Chair of IFIP Working Group 10.4 on Dependable Computing. In addition, he serves on the editorial board of IEEE Transactions on Reliability, and is the Area Editor for Simulation and Modeling of Computer Systems for the ACM Transactions on Modeling and Computer Simulation. He is a past Chair of the IEEE Technical Committee on Fault-Tolerant computing. He is a Fellow of the IEEE. Dr. Sanders's research interests include performance/dependability evaluation, dependable computing, and reliable distributed systems. He has published more than 140 technical papers in these areas. He is currently serving as the General Chair of the 2003 Illinois International Multiconference on Measurement, Modelling, and Evaluation of Computer-Communication Systems. He has served as co-Chair of the program committees of the 29th International Symposium on Fault-Tolerant Computing (FTCS-29), the Sixth

IFIP Working Conference on Dependable Computing for Critical Applications, Sigmetrics 2003, PNPM 2003, and Performance Tools 2003, and has served on the program committees of numerous conferences and workshops. He is a co-developer of three tools for assessing the performability of systems represented as stochastic activity networks: METASAN, UltraSAN, and Mobius. Mobius and UltraSAN have been distributed widely to industry and academia; more than 300 licenses for the tools have been issued to universities, companies, and NASA for evaluating the performance, dependability, security, and performability of a variety of systems. He is also a co-developer of the Loki distributed system fault injector and the AQuA/ITUA middlewares for providing dependability/security to distributed and networked applications.



Michel Cukier is currently an Assistant Professor in the Center for Reliability Engineering, part of the Department of Materials and Nuclear Engineering at the University of Maryland, College Park. His research interests include intrusion tolerance, fault tolerance in distributed systems, modeling and fault injection. He is a member of the IEEE and the IEEE Computer Society.