An Adaptive Framework for Tunable Consistency and Timeliness Using Replication *

Sudha Krishnamurthy, William H. Sanders
Coordinated Science Laboratory,
Dept. of Computer Science, and
Dept. of Electrical & Computer Engineering
University of Illinois at Urbana-Champaign

Michel Cukier
Center for Reliability Engineering
Dept. of Materials & Nuclear Engineering
University of Maryland, College Park

E-mail: {krishnam,whs}@crhc.uiuc.edu, mcukier@eng.umd.edu

Abstract

One of the well-known challenges in using replication to service multiple clients concurrently is that of delivering a timely and consistent response to the clients. In this paper, we address this problem in the context of client applications that have specific temporal and consistency requirements. These applications can tolerate a certain degree of relaxed consistency, in exchange for better response time. We propose a flexible QoS model that allows these clients to specify their temporal and consistency constraints. In order to select replicas to serve these clients, we need to control the inconsistency of the replicas, so that we have a large enough pool of replicas with the appropriate state to meet a client's timeliness, consistency, and dependability requirements. We describe an adaptive framework that uses lazy update propagation to control the replica inconsistency and employs a probabilistic approach to select replicas dynamically to service a client, based on its QoS specification. The probabilistic approach predicts the ability of a replica to meet a client's QoS specification by using the performance history collected by monitoring the replicas at runtime. We conclude with experimental results based on our implementation.

1. Introduction

Replicating distributed services enables us to service multiple clients concurrently, and deliver good response times, by selecting different replicas to service different clients. However, since concurrency has the potential to introduce replica inconsistency, 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 with *immediate* convergence, or *weak* consistency with *eventual* convergence. Both of these consistency models have been studied extensively. In the strong consistency model, concurrent operations on replicated data are equivalent to a sequential execution on non-replicated data. Pessimistic replication algorithms, such as active and passive replication (e.g., [1, 11, 8, 12]), 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 maintaining 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, 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., [2, 9]) 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.

In this work, our goal is to support applications that have specific time constraints. 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. Examples of applications that benefit from relaxed but bounded inconsistency in exchange for timeliness in-

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

clude real-time database applications, such as online stock-trading and traffic-monitoring applications. In order to support such applications that have specific temporal and consistency requirements effectively, we use an approach that allows the users to express their timeliness and consistency requirements as a quality-of-service (QoS) specification. To study the trade-offs between timeliness and consistency, we propose an adaptive middleware framework that allows us to explore the intermediate space between the above binary views of consistency. We have implemented this framework in AQuA, a CORBA-based middleware that supports transparent replication of objects across a LAN [11].

We now list the main contributions of this paper. In Section 2, we propose a QoS model that allows a broad spectrum of applications to express their timeliness and consistency requirements. In Section 3 we describe our framework that allows us to build protocols for providing different consistency guarantees. 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 a proof-of-concept, in Section 4, we describe the protocol we have implemented to maintain sequential consistency among the replicas. In Section 5 we describe a probabilistic approach that allows a middleware to dynamically select replicas to service the clients based on the QoS specification of the clients. Similar to the work we presented in [5], this approach 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 were stateless, our current model addresses this selection problem in the context of replicas with state. In Section 6, we present a few experimental results based on our implementation.

2. QoS Model for Timeliness and Consistency

Several other 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 [10]), and timed consistency models (defined in [13, 6]), 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 [15] 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 of propagating the writes among the replicas. However, while these models provide a way to quantify consistency, they do not address the problem of tuning consistency requirements in the presence of specific transaction deadlines or response time constraints. We now describe our QoS model that allows the clients to express their consistency and response time requirements.

Our request model enables a middleware to distinguish invocations that modify the state of the object they invoke from those that merely retrieve the state. To do this, a client application has to explicitly specify all the *read-only* methods it invokes on an object by their names. 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.

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 provided by a service 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 [1]. In this paper, we target services that provide sequential 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. We compute the staleness of a replica by associating a timestamp with each update operation. We use timestamps based on "logical clocks" [7] 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." A replica whose staleness is x has a state that 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. 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.

The timeliness specification includes 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. 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, con-

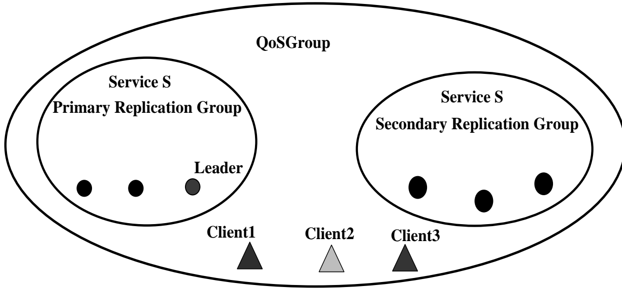


Figure 1. Replica organization

sider 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*.

3. Framework for Tunable Consistency

Given the above QoS model, our goal is to build a framework that can be 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 the protocols that implement different consistency semantics and design of an infrastructure that would allow these protocols to be used on demand, and 3) development of a mechanism to select replicas dynamically to service a client based on the client's QoS requirements. We now describe the approach we have used to address these issues in the context of AQuA.

All the replicas offering the same service are organized into two groups: a *primary replication group* and a *secondary replication group*, as shown in Figure 1. 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 [14], and members of a group communicate with each other by making use of the Maestro-Ensemble group communication protocol [3]. 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 we 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 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. The above two-level replica organization was motivated by the need to favor the operations that can tolerate relaxed consistency to a certain degree in exchange for a timely response. We reduce the overheads incurred by a write-all scheme, such as an active replication 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. 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.

4. Tunable Consistency Protocols

In Section 2, we mentioned that to maintain replica consistency, the replicas should serve their clients by respecting the ordering guarantee associated with the service. Our framework allows different ordering guarantees to be implemented as *timed consistency handlers* in the AQuA gateway, as shown in Figure 2. A client can communicate with a replicated service by using the gateway handler appropriate for the service. For example, Figure 2 shows a client communicating with two different services. Service A is an example of 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 sequential consistency handler to communicate with service A, while it uses the FIFO handler to communicate with service B. In this paper, we will describe the sequential consistency handler we have implemented in the AQuA middleware. 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 sequential consistency across the two groups of replicas, and in the next section we will describe how the client-side protocol uses these replicas to meet the client's QoS specification.

4.1. Sequential Consistency Protocol

In our sequential consistency model, the update requests of the clients are executed by all of 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 this member the *lazy publisher*. Thus, although the replicas may update their state at different points

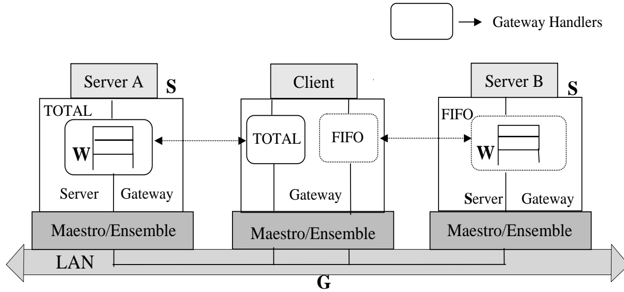


Figure 2. Timed consistency handlers in AQuA

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. The leader merely serves as the *sequencer* and does not actually service the client's request.

We now describe how the updates and read-only requests are processed by the replicas. The processing depends on whether the replica is a primary or secondary replica. All of this processing is done at the middleware layer, within the gateway handler of the replicas. Each gateway handler maintains a pair of variables, `my_GSN` and `my_CSN`, which are used by the protocol to provide sequential consistency. `my_GSN` is the replica's local view of the current GSN, and `my_CSN` is the replica's commit sequence number (CSN), which indicates the GSN of the most recent update committed by the replica. The commit sequence number increases strictly in monotonic order, and a replica is assumed to have committed every update whose global sequence number is less than or equal to the value of its `my_CSN`. Our protocol ensures that the consistency guarantees are preserved even when replica failures occur. This is done by handling the failures of the sequencer and the lazy publisher, which play a crucial role in providing sequential consistency semantics. However, we omit the details of the failure handling in this paper due to the space constraint.

4.1.1. 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.

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. Otherwise, the replica stores the request in a buffer and processes it upon receiving the GSN assignment from the sequencer. If the update request is in sequential order, 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 and commits it at a later time, after the intermediate requests have been committed.

4.1.2. Read-Only Operations. In our sequential consistency model, a read-only request is sent to the sequencer and a subset of the primary and secondary replicas. Different replicas may service different sets of read-only requests. 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. 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 this GSN to measure the staleness of their state. To determine its staleness, the replica first sets its value of `my_GSN` to the value of the GSN broadcast by the sequencer. The replica then computes the value of (`my_GSN - my_CSN`). This value is a measure of how stale the state of the replica is. If the replica's state is less stale than the threshold specified by the client in its QoS specification, the replica can service the client's request immediately. However, a secondary replica may have a state that is more stale than the staleness threshold specified by the client, because the secondary replicas update their state only when they receive the state update from the lazy publisher. In that case, the replica performs a *deferred read* by buffering the read request and responding to the client immediately after receiving the next state update from the lazy publisher.

5. Probabilistic Model-Based Replica Selection

Having described the protocol processing in the server-side gateway handler, we now describe the processing performed in the client-side handler to meet the QoS specification of the client. 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, one of the important responsibilities of the client handler is to select an appropriate subset of replicas to service the clients, and reduce the occurrence of such timing failures.

A simple approach would be to allocate all the available replicas to service a single client. However, such an approach is not scalable, as it increases the load on all the replicas and results in higher response times for the remaining clients. On the other hand, assigning a single replica to service each client allows us to service multiple clients concurrently. However, should a replica 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. Therefore, 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 of the replicas.

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 4 for the sequential ordering case. 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, a client can be certain that the state of the primary replicas is always up-to-date, because of the immediate update propagation. However, it cannot make such guarantees about the state of the secondary replicas, which update their state only when they receive the lazy update 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. We will now describe our probabilistic models and replica selection algorithm. They enhance the selection approach we presented in [5], which made the assumption that the replicas were stateless. We first define the notation we use to explain our model.

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 R_i be the random variable that denotes the response time of replica i. 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, and $P_K(d)$ 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 these individual probabilities 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.

5.1. Modeling the Response Time Distribution

We 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 be solved online, especially for the time-sensitive applications we target in our work. Furthermore, the results we present in Section 6 show that the resulting model makes reasonably good predictions for the scenarios we have considered. Thus, using the independence assumption, we obtain

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

5.1.1. Case 1: Primary Replicas. In Section 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

$$P(\text{no } i \in K_p \ni R_i \leq d) = \prod_{i \in K_p} P(R_i > d) = \prod_{i \in K_p} (1 - F^I_{R_i}(d)) \quad (2)$$

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.

5.1.2. 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. As mentioned in Section 4.1.2, if the replica's state is more stale than the staleness threshold specified by the client, the replica has to buffer the request until it receives the next lazy update, at which point it can respond to the request. Therefore, for a replica $j \in K_s$,

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

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, we associate staleness with the entire secondary group of replicas, instead of with an individual replica j as above. 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.

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

$$P(\text{ no } j \in K_s \ni R_j \leq d) = \left[\prod_{j \in K_s} (1 - F^I_{R_j}(d))\right] \cdot P(A_s(t) \leq a) + \left[\prod_{j \in K_s} (1 - F^D_{R_j}(d))\right] \cdot (1 - P(A_s(t) \leq a)) \quad (3)$$

where $F^I_{R_j}$, as before, denotes the response time distribution function for the replica j, given that it 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 performs a deferred read.

We now describe how we compute the staleness factor, $P(A_s(t) \leq a)$, for the secondary replicas, and then follow this 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.

5.1.3. 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 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

$$P(A_s(t) \leq a) = P(N_u(t_l) \leq a) = \sum_{n=0}^{a} \frac{(\lambda_u t_l)^n e^{-\lambda_u t_l}}{n!} \quad (4)$$

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 5.4.1, we will explain how we measure these two parameters at runtime. Although we have assumed Poisson arrivals in our work, it should be possible to evaluate $P(N_u(t_l) \leq a)$ for the case in which the arrival of update requests follows a distribution that is not Poisson. Finally, we can use the expressions in Equations 2, 3, and 4 in Equation 1 to evaluate $P_K(d)$.

5.2. 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 extend the method we described in [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.

5.2.1. 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 that has the appropriate state, the response time random variable for a replica i is given by Equation 5:

$$R_i = S_i + W_i + G_i \quad (5)$$

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 it includes the time the replica spends waiting for the sequencer to send the GSN for the request); and G_i is the random variable denoting the two-way gateway-to-gateway delay between the client and replica i. The service time and queuing delay are specific to the individual replicas, while the gateway delay is specific to a client-replica pair.

5.2.2. Deferred Reads. In the case in which the replica has to wait for a state update before responding to the request, the response time random variable is given by Equation 6, 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.

$$R_i = S_i + W_i + G_i + U_i \quad (6)$$

For each request, we experimentally measure the values of the above performance parameters as described in Section 5.4. The client handlers record the most recent l measurements of these parameters in separate sliding windows in an information repository. The size of the sliding window, l,

is chosen so as to include a reasonable number of recently measured values, while eliminating obsolete measurements. To evaluate $F^I_{R_i}(d)$, we first compute the probability mass functions (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 also record a performance history of U_i and include the pmf of U_i in the convolution.

5.3. State-Based Replica Selection Algorithm

Algorithm 1 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 above. 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.

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, this may cause the clients to select the same or common replicas. Hence, Algorithm 1 has been designed 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 this by using information that is specific to a client-replica pair, in addition to the replica-specific performance information, as we now describe.

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 response time, ert_i, which is the duration that has elapsed since a reply was last received by the client from replica i. The response time distributions, which are computed from the performance history as explained in Section 5.2, are specific to the individual replica and are nearly identical in all the client information repositories. However, the ert 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 4.

Algorithm 1 State-Based Replica Selection Algorithm

```
Require: V =< i, F^I_{R_i}(d), F^D_{R_i}(d), ert_i > , staleFactor
Require: Client Inputs: a : staleness threshold, d : deadline, P_c(d): minimum probability of meeting this deadline
 1: primCDF ⇐ 1 ; secImmedCDF ⇐ 1; secDelayedCDF ⇐ 1
 2: sortedList ⇐ sort V in decreasing order of ert_i.
 3: K ⇐ [first(sortedList)] ; maxCDFReplica ⇐ [first(sortedList)] ; advance(sortedList)
 4: for all i in sortedList do {visit the remaining replicas in sorted order}
 5:   K ⇐ K ∪ i
 6:   if F^I_{R_i}(d) > maxCDFReplica.immedCDF() then
 7:     found ⇐ includeCDF(maxCDFReplica, maxCDFReplica.immedCDF(), maxCDFReplica.delayedCDF())
 8:     maxCDFReplica ⇐ i
 9:   else
10:     found ⇐ 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:     K ⇐ K ∪ Sequencer; return K
14:   end if
15: end for
16: K ⇐ K ∪ Sequencer; return K {return the set comprising all the replicas}
17: includeCDF(replica, immedCDF, delayedCDF)
18: begin
19: if replica ∈ PrimaryGroup then
20:   primCDF ⇐ primCDF * (1 - immedCDF)
21: else
22:   secImmedCDF ⇐ secImmedCDF * (1 - immedCDF); secDelayedCDF ⇐ secDelayedCDF * (1 - delayedCDF)
23:   secCDF ⇐ secImmedCDF * staleFactor + secDelayedCDF * (1 - staleFactor)
24: end if
25: if 1 - (primCDF * secCDF) ≥ 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
```

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 while servicing the request. To do this, 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 selected member, m, 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 [5] we have provided a formal justification for this proposal. We now describe the steps of Algorithm 1, which makes use of this proposal to select the replicas to service the client.

The algorithm first sorts the replicas in decreasing order of their elapsed response times, ert. This allows the clients

to favor the selection of replicas that it used least recently and thereby obviate the hot-spot problem mentioned above. Replicas that have the same value of ert 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 such that the terminating condition $P_K(d) \geq P_c(d)$ is satisfied. The function `includeCDF()` uses the values of $F^I_{R_i}(d)$ and $F^D_{R_i}(d)$, which it receives as inputs, to compute the value of $P_K(d)$ according to Equation 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. Notice that when evaluating $P_K(d)$, we exclude the response time distribution of the selected member, `maxCDFReplica`, that has the highest probability (among the selected members) of responding by the requested deadline. This 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 choose a set K that can tolerate a single replica failure, as proposed above. Finally, the selected set K is extended to include the sequencer.

5.4. Online Performance Monitoring

We now explain some of the main implementation details of how the client and server gateway protocols interact to measure and record the different performance parameters that are used to compute the distribution function and staleness factor. 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 5.3. 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, after processing it according to the sequential consistency gateway protocol described in Section 4. We instrumented the gateway handler so that it can record the service time, t_s, and queuing time, t_q, for the request. In addition, if a replica performs a deferred read, it records the duration of time, t_b, for which it buffered the request until 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. All of this information, 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. This is used by the client to compute the elapsed response time for the replica, when the client executes Algorithm 1 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.

5.4.1. Measuring the Staleness. From Equation 4, we infer that if a client gateway knows the arrival rate of the update requests (λ_u) and the time 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. To measure the values of these parameters, the server that is designated as the lazy publisher broadcasts the following additional information when it publishes its performance measurements to the clients: 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 $\lambda_u = \sum n^i_u / t^i_u$, 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. Note that in order to collect any of the timing data as explained above, it is not necessary to synchronize the clocks across the machines, because we always measure the two end-points of a timing interval on the same machine.

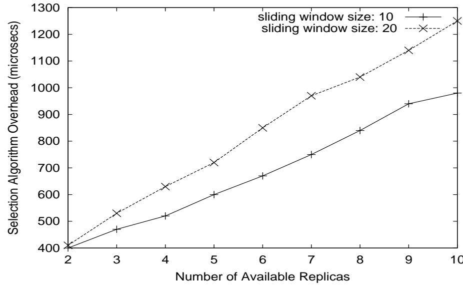
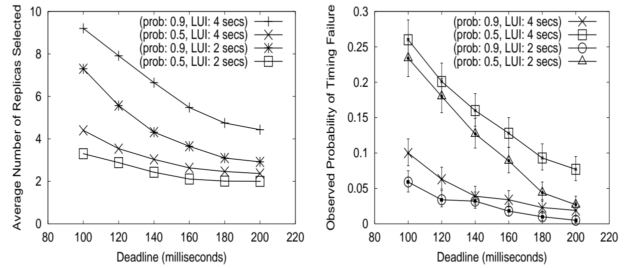


Figure 3. Overhead of selection algorithm



(a) Number of replicas selected

(b) Timing failure probability

Figure 4. Adaptivity of probabilistic model

6. Experimental Results

We now discuss the experiments we conducted to analyze the performance of the probabilistic replica selection algorithm, as implemented in AQuA. Our experimental setup is composed of a set of uniprocessor Linux machines distributed over a 100 Mbps LAN. The processor speeds range from 300 MHz to 1 GHz. 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 [4].

Figure 3 shows how the overhead of the probabilistic selection algorithm varies with the number of available replicas for sliding windows of sizes 10 and 20. We account for these overheads when selecting the replicas. Computation of the response time distribution function contributes to 90% of these overheads, while selection of the replica subset using Algorithm 1 contributes to the remaining 10%. The larger the sliding window, the greater the number of data points used to compute the response time distribution, and therefore the higher the selection overhead. We used a sliding window of size 20 for the experiments we describe below.

6.1. Validation of Probabilistic Model

We conducted experiments to validate the probabilistic model by evaluating how effectively the subset of replicas chosen by the probabilistic selection algorithm was able to meet the QoS requested by the client. To do this, we used an experimental setup composed of 10 server replicas, in addition to the sequencer. 4 of the server replicas were in the primary group, and the remaining ones were in the secondary group. 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 our experiments, we used two clients that ran on two different machines and independently issued requests to the replicated service with a 1000 millisecond *request delay*, which we define as the duration that elapses before a client issues its next request after completion of its previous request. In every run, each of the two clients issued 1000 alternating write and read requests to the service. One of the clients requested the same QoS for all of the runs; the QoS included a staleness threshold of value 4, a deadline of 200 milliseconds, and a minimum probability of timely response of 0.1. The second client specified a staleness threshold of value 2 in all of the runs, but requested a different deadline in each run. To study the behavior of the 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 the second client in its QoS specification: 0.9 and 0.5.

For each of the deadline values of the second client, we experimentally computed the probability of timing failures in a run by measuring the number of requests in the run for which the second client failed to receive a response within the requested deadline. Further, in order to study the effect of the staleness of the replicas on the timeliness of their response, we repeated the experiments using different lazy update intervals (LUI, also denoted by T_L in Section 5.4.1). Here we present the results for LUI of 2 seconds and 4 seconds.

Figure 4a shows the average number of replicas selected by the selection algorithm to service the second client for each of its QoS specifications. From this figure we observe that the number of replicas chosen by the algorithm to service a request reduces as the client's QoS specification becomes less stringent. The reason for this is that our algorithm, as outlined in Section 5.3, never selects more replicas than are required in order to meet a client's QoS requirement. The less stringent a client's QoS specification, 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.

Figure 4b shows how successful the replicas selected in Figure 4a were in meeting the QoS specifications of the second client. The first observation from Figure 4 is that in each case, the set of replicas selected by Algorithm 1 was able to meet the client's QoS requirements successfully by maintaining the timing failure probability within the failure

probability acceptable to the client. For example, consider the case in which the LUI is 4 seconds and the client has specified that the probability of timely response must be at least 0.9. The observed probability of timing failures in this case varies from 0.1 to 0.02 as the deadline varies from 100 milliseconds to 200 milliseconds. We observe similar behavior for the other cases. Thus, for the experimental runs we conducted, the model we used was able to predict the set of replicas that would be able to return the appropriate response by the client's deadline, with at least the probability requested by the client. A second observation from Figure 4b is that as the interval between lazy updates increases, the observed probability of timely failures also increases. The reason is that as the interval between lazy updates increases, the replica's state becomes increasingly stale. 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. This delayed response results in a higher probability of timing failures.

7. Conclusions

We have presented a framework for providing tunable consistency and timeliness at the middleware layer using replication. In addition to the experiments we have presented, we have also conducted other extensive experiments by varying the different parameters, such as the lazy update interval and request delay. All of the experimental results we obtained show that our probabilistic approach can adapt the selection of replicas to meet a client's timeliness and consistency constraints in the presence of delays and replica failures, if enough replicas are available. Since we provide probabilistic temporal guarantees, we currently admit all the clients and inform a client if the observed failure probability exceeds the client's expectations after the failures have been detected. However, with some modifications, we can also use our framework to perform admission control, in order to determine the clients that can be admitted based on the current availability of the replicas. Finally, 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 priority or the cost the client 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.

Acknowledgments: We thank the reviewers for their feedback. We thank Kaustubh Joshi for his feedback on the probabilistic models. We are grateful to Jenny Applequist for her comments.

References

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

[2] 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.

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

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

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

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

[15] 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.