Performance Evaluation of a QoS-Aware Framework for Providing Tunable Consistency and Timeliness

Sudha Krishnamurthy, William H. Sanders
Coordinated Science Laboratory,
Dept. of Computer Science,
and Dept. of Electrical & Computer Engineering
University of Illinois at Urbana-Champaign
Email: {krishnam,whs}@crhc.uiuc.edu

Michel Cukier
Center for Reliability Engineering
Dept. of Materials & Nuclear Engineering
University of Maryland, College Park
Email: mcukier@eng.umd.edu

***Abstract*— Strong replica consistency models ensure that the data delivered by a replica always includes the latest updates, although this may result in poor response times. On the other hand, weak replica consistency models provide quicker access to information, but do not usually provide guarantees about the degree of staleness in the data they deliver. In order to support emerging distributed applications that are characterized by high concurrency demands, increasing shift towards dynamic content, and timely delivery, we need quality-of-service models that allow us to explore the intermediate space between these two extreme approaches to replica consistency. Further, to better support time-sensitive applications that can tolerate relaxed consistency in exchange for better responsiveness, we need to understand how the desired level of consistency affects the timeliness of a response. The QoS model we have developed to realize these objectives considers both timeliness and consistency, and treats consistency along two dimensions: order and staleness. In this paper, we experimentally evaluate the framework we have developed to study the timeliness/consistency tradeoffs for replicated services and present experimental results that compare these tradeoffs in the context of sequential and FIFO ordering.**

I. Introduction

Distributed applications often have to contend with different degrees of unpredictability, which arise from several factors, such as unanticipated faults in the system and transient overloads caused by sharing the network and computing resources with other users and applications. Owing to this unpredictability, users can benefit if they are allowed to control the quality-of-service (QoS) they receive from the services they access. To support the diverse QoS requirements of modern applications, we need to build *QoS-aware* middleware layers that allow the clients to express their application-specific requirements using the right level of abstraction. The middleware should be able to map the application-specific requirements to the properties of the resources it manages, so that the requirements can guide the middleware in effectively sharing the resources among the users. Furthermore, owing to the dynamic nature of the distributed environment, the middleware should be able to adapt the allocation of resources to meet the requirements of the users, based on feedback from the environment. Considerable work on building such QoS-aware middleware layers for specific applications has been done. For example, Agilos [8] uses a control-theoretic approach to adaptively allocate resources, such as network bandwidth and CPU cycles, to multimedia-related applications. Globus [2] is another middleware that manages the allocation of multiple resources, such as the network, CPU, and disk resources for a variety of distributed applications, by combining features of a reservation-based approach and an adaptation-based approach.

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

In contrast, we have a designed a distributed, QoS-aware, middleware framework that manages all of the server replicas offering a service, and adaptively assigns replicas to service a client based on the client's QoS requirements and the responsiveness of the replicas [6]. This work was done in the context of AQuA [9], which is a CORBA-based dependable middleware that transparently replicates objects across a LAN. Replicating a service provides dependability and enables us to deliver good response times, by selecting different replicas to service different clients concurrently. However, since concurrent operations have the potential to introduce replica inconsistency, one of the challenges in replicating distributed services is the problem of delivering a consistent and timely response to the clients. Traditional replica consistency models use a binary approach: *strong consistency* models ensure that the data delivered by a replica always includes the latest updates, although this may result in poor response times; *weak consistency* models provide quicker access to information, but do not usually provide guarantees about the degree of staleness in the information they deliver. In our research, we target 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. Our framework allows clients to access replicated services by requesting quality-of-service along two dimensions: timeliness of response and consistency of delivered data. While our framework ensures that the consistency requirements are always met, the uncertainty in most distributed environments makes it hard to provide deterministic guarantees for meeting the timeliness requirements of applications. Hence, our approach provides probabilistic temporal guarantees.

One of the important responsibilities of our QoS-aware layer is the selection of appropriate replicas to service the clients to meet their QoS requirements. One approach would be to select 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 [5]. 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. We use an approach based on an application-aware model of adaptation, in which the middleware and application collaborate to adapt. This supports application diversity by allowing the applications to determine the mapping between their QoS requirements and the replicas. The middleware keeps track of the current state and responsiveness of the replicas by monitoring them at runtime. It then uses probabilistic models to select an appropriate subset of replicas to service a client, by using the runtime measurements as inputs to the model.

Our framework can be used to support multiple consistency semantics and to study the timeliness/consistency tradeoffs for replicated services. In [6], we described how the framework can be used to support relaxed consistency semantics using sequential ordering [1]. In this paper, we have extended the framework to support FIFO ordering, which provides weaker consistency semantics than sequential ordering. We also provide an experimental evaluation of the framework for both types of ordering and present results that compare the timeliness/consistency tradeoffs for sequential and FIFO ordering. The results we present are promising and show that the probabilistic approach we have developed allows a middleware to effectively adapt the allocation of replicated servers to meet the QoS requirements under different scenarios.

The remainder of this paper is organized as follows. In Section II we describe our QoS model for accessing replicated objects. In Section III we review the main features of the adaptive framework we have designed to support multiple consistency semantics at the middleware layer. In Section IV, we explain the probabilistic models that form the basis for evaluating the timeliness/consistency tradeoffs. We present our experimental analysis in Section V. We present our conclusions in Section VI, and discuss ideas for future extensions in Section VII.

II. QoS Model for Accessing Replicated Objects

We use a request model that allows the middleware to distinguish update invocations, which modify the state of the object they invoke, from read-only invocations that merely retrieve the state. Our QoS model allows a client to specify its timeliness requirements using a pair of attributes: $<$*response time, probability of timely response*$>$. This pair specifies the time by which a client expects a response after it has transmitted its request, and the probability with which it expects its temporal constraint to be met. Failure to meet a client's deadline results in a *timing failure* for the client. In our QoS model, the timeliness attribute is applicable only for read-only requests and not for update operations.

Several metrics have been proposed to quantify consistency. For example, the TACT middleware [11], a closely related work, uses numerical error, order error, and staleness in real-time to assess the tradeoffs between consistency and availability of replicas. On the other hand, 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 its clients about the order in which their requests will be processed by the servers, so as to prevent conflicts between operations. We currently support sequential and FIFO ordering. 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. It bounds the degree of staleness in a way that is meaningful to the users. 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 each update operation with a timestamp. We use timestamps based on "logical clocks" [7] because this obviates the need for synchronized clocks across the distributed replicas. A replica whose staleness is x is one whose state has not yet been updated to reflect the modifications ensuing from the most recent x updates, but reflects all updates committed prior to that. As an example of the use of the above model, a client of a document-sharing application can specify that he wishes to obtain a copy of the document that is not more than 5 versions old within 500.0 milliseconds with a probability of at least 0.8.

III. Support for Tunable Consistency

We now review the main ideas of the framework that makes use of the above QoS model to support tunable consistency and timeliness at the middleware layer. The main components of this framework are the hierarchical organization of the replicas, the protocols that implement different ordering guarantees, and a mechanism that dynamically assigns replicated servers to service a client based on the client's QoS requirements. Before we review these individual components, we first provide an overview of the AQuA middleware, shown in Figure 1.

A. Overview of AQuA

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 [10], [3], above which AQuA is layered. The use of group communication in AQuA is transparent to the end applications. Hence each of the clients,

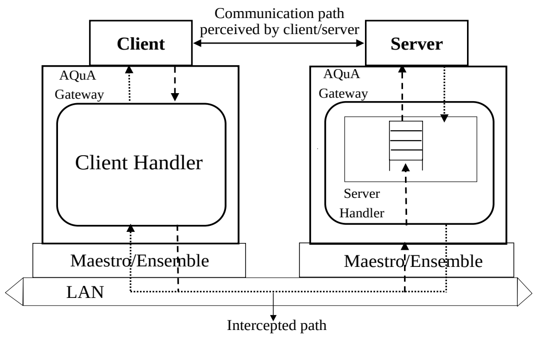


Fig. 1

AQUA OVERVIEW

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, as shown in Figure 1. For the sake of clarity, in this figure we have illustrated a server replica group that has only a single member. In reality, this group may have multiple replica members. While previous work in AQuA [9] has focused on gateway handlers for providing fault tolerance, we have enhanced AQuA by developing gateway handlers that support tunable consistency and timeliness for time-sensitive applications.

B. Hierarchical Replica Organization

We organize all the replicas offering a service into two groups: a *primary replication group* and a *secondary replication group*. We also use 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. We depend on Maestro-Ensemble to provide reliability, virtual synchrony, and FIFO messaging guarantees, and build upon these guarantees to provide the different end-to-end consistency guarantees. Maestro-Ensemble is also responsible for informing the group members when changes in the group membership occur.

The primary and secondary replication groups of an object may be adaptively organized to implement multiple consistency semantics. The primary replication group is used to implement strong consistency semantics, whereas the secondary group implements weaker consistency semantics. This two-level replica organization was motivated by the need to favor 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. Our approach, on the other hand, performs 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

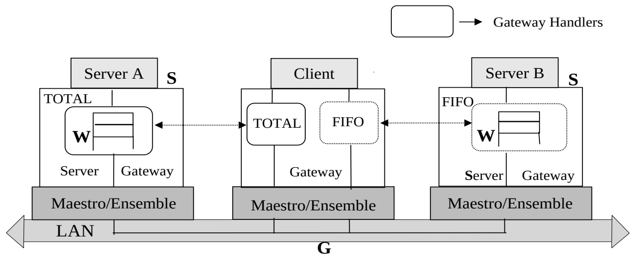


Fig. 2

TIMED CONSISTENCY HANDLERS IN THE AQUA GATEWAY

propagation. We can bound the degree of divergence between the states of primary and secondary replicas 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. In Section V-B, we will present experimental results that justify the use of a hierarchical replica organization.

C. Ordering Guarantees

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 2. The framework we described in [6] supported only sequential ordering. In this paper, we extend this framework by adding FIFO ordering. The sequential handler was motivated by applications, such as document-sharing applications, in which the replicated servers provide total ordering by maintaining a common state that is shared by all the clients. 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 2 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. We do not, however, describe the details of the failure handling in this paper, owing to the space constraint.

We will 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 sent 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*. A client may send its read-only request to a subset of primary and secondary replicas. In Section IV, 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 for this is 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 of the primary group is elected by Ensemble. 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.

IV. Replica Selection

Having described the replica organization and the protocols that provide the different application-specific ordering guarantees for the replicated servers, we now review the third main component of our adaptive framework. This is a selection module that is local to each client's gateway handler and provides a mechanism to select replicas to service a client's read request, 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 in its QoS specification. For an update request, the mechanism selects all the primary replicas.

The selection problem is similar to a resource allocation problem and is challenging, because in most distributed environments it is impossible for a client to know with certainty if any single replica can meet its deadline. Further, while the client can be certain that the state of the primary replicas is always up-to-date because of the immediate update propagation, it cannot make such guarantees about the state of the secondary replicas, which update their state lazily. Hence, our selection scheme makes use of spatial redundancy by selecting multiple replicas to service a read request, and delivers the earliest response to the client. Since distributed services may have high concurrency demands, the selection algorithm has to choose the degree of redundancy to service a request judiciously. Our algorithm 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 predictions based on information gathered by monitoring the replicas at run-time. Our selection algorithm then uses this online prediction to choose a subset of replicas that can 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.

A. Parameters of the Probabilistic Model

To identify the parameters of our probabilistic model, we conducted experiments to determine the factors that affect the response time in AQuA. Based on our experimental analysis, we found that in the case of an immediate read, the response time random variable R_i for a replica i is given by Equation 1:

$$R_i = S_i + W_i + G_i \tag{1}$$

For a deferred read, R_i is given by Equation 2:

$$R_i = S_i + W_i + G_i + U_i \tag{2}$$

where S_i is the random variable denoting the service time for a read request processed by replica i; W_i is the random variable denoting the queuing delay experienced by a request waiting to be serviced by i; 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. As shown in Figure 2, the service times (S) and queuing delays (W) are specific to the individual replicas, while the gateway delays (G) are specific to a client-replica pair. 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 [6].

B. Estimating the Probability of a Timely Response

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; 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. Using the above performance history, collected by monitoring the performance of the replicas online, our selection module predicts the probability, $P_K(d)$, that at least one response from the subset $K \in M$, consisting of $k > 0$ replicas, will arrive by the client's deadline, d, and thereby prevent a timing failure. M denotes the complete set of replicas that offer the service requested by the client. 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 independently 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. Using this independence assumption, we have

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

$$P_K(d) = 1 - P(\text{no } i \in K_p \ni R_i \leq d) \cdot P(\text{no } j \in K_s \ni R_j \leq d) \quad (3)$$

Although the independence 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. This tradeoff was necessary since our work targets time-sensitive applications. The experimental results we present in Section V show that, despite this approximation, the resulting model makes reasonably good predictions most of the time.

Before we proceed to explain how we evaluate $P_K(d)$, we will introduce a few notations. Let t denote the time at which a request is transmitted by the client to the server. Since replicas are selected at the time of request transmission, we also use t to denote the time at which the replica selection is done. Let $P(R_i \leq d)$ denote the probability that a response from replica i will be received by the client within the client's deadline, d. This probability 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. Let $A_i(t)$ denote the staleness of the state of replica i at time t, and let $P(A_i(t) \leq a)$ denote the *staleness factor* for replica i at time t. The staleness factor is defined as the probability that the state of replica i at the time of request transmission, t, is within the staleness threshold, a, specified by the client. Since the update requests of the clients are propagated to the primary group immediately, the staleness factor $P(A_i(t) \leq a) = 1$ for a primary replica. Hence, in the case of the primary subset, we have

$$P(\text{no } i \in K_p \ni R_i \leq d) = \prod_{i \in K_p} P(R_i > d) = \prod_{i \in K_p} (1 - F^I_{R_i}(d)) \quad (4)$$

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

For a secondary replica, the staleness factor does not always have a value of 1. Therefore, for a replica $j \in K_s$,

$$\begin{aligned} P(R_j > d) = {} & P(R_j > d | A_j(t) \leq a) \cdot P(A_j(t) \leq a) \\ & + P(R_j > d | A_j(t) > a) \cdot P(A_j(t) > a) \end{aligned}$$

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 associating the staleness factor with an individual secondary replica as above, we use $A_s(t)$ to denote the staleness of the secondary group at the time of request transmission t. The expression for the probability that no replica in the secondary subset can respond within the deadline d is then given by

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

$$\begin{aligned} P(\text{no } j \in K_s \ni R_j \leq d) = {} & \left[\prod_{j \in K_s} (1 - F^I_{R_j}(d)) \right] \cdot P(A_s(t) \leq a) + \\ & \left[\prod_{j \in K_s} (1 - F^D_{R_j}(d)) \right] \cdot (1 - P(A_s(t) \leq a)) \end{aligned} \quad (5)$$

where $F^I_{R_j}$, as before, denotes the response time distribution function of the replica j for an immediate read, and $F^D_{R_j}$ is the response time distribution function, given that the replica defers the read until it has received the next lazy state update.

We make use of the performance history of the replicas recorded at runtime, as explained in Section IV-A, to compute the values of $F^I_{R_i}$ and $F^D_{R_i}$ for a replica i . To evaluate $F^I_{R_i}(d)$, we first compute the probability mass function (pmf) of S_i and W_i based on the relative frequency of their values recorded in the sliding window. We then use the pmf of S_i, the pmf of W_i, and the most recently recorded value of G_i to compute the pmf of the response time R_i as a discrete convolution of S_i, W_i, and G_i. For G_i, we decided to use its most recently recorded value rather than record its history over a period of time, because the gateway-to-gateway delay in a LAN does not fluctuate as much as the other parameters do. In environments where this observation is not true, it would be simple to extend our approach to record the gateway delay over a sliding window, as we do for the service time and queuing delay. Finally, the pmf of R_i obtained using the discrete convolution can 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.

We now describe how a client's selection module estimates the staleness factor, $P(A_s(t) \leq a)$, at the time it selects replicas to service the client's read transaction. The staleness of a secondary replica at the instant t is the number of update requests that has 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 number of update requests committed by the primary group in the duration t_l. Since $A_s(t) = N_u(t_l)$, we have $P(A_s(t) \leq a) = P(N_u(t_l) \leq a)$. Using the assumption that the update arrivals follow a Poisson distribution with rate λ_u, we obtain

$$P(A_s(t) \leq a) = P(N_u(t_l) \leq a) = \sum_{n=0}^{a} \frac{(\lambda_u t_l)^n e^{-\lambda_u t_l}}{n!} \qquad (6)$$

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 the client handler knows the arrival rate of the update requests and the time elapsed since the last lazy update. Like the performance parameters in Section IV-A, this information is obtained by instrumenting the gateway handlers [6]. Although we have assumed Poisson arrivals in our work, it should be possible to evaluate the staleness factor for other kinds of update distributions by using the appropriate models. Now that we have a way to compute all the components of our probabilistic model, we can use the expressions from Equations 4, 5, and 6 in Equation 3 to evaluate $P_K(d)$.

C. *Selection of Replicas with Dynamic State*

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 in the previous section. The algorithm uses the model's prediction to select no more than the number of replicas necessary to meet the client's response time constraint with the probability the client has requested. The algorithm is executed by the selection module in a client gateway when the client performs a read-only request on a server object. If the client makes an update request, the selection module chooses all the primary replicas to service the request.

Algorithm 1 Replica Selection Algorithm: Dynamic State

```
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:   $K \Leftarrow K \cup i$
 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:     return $K$
14:   end if
15: end for
16: 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
```

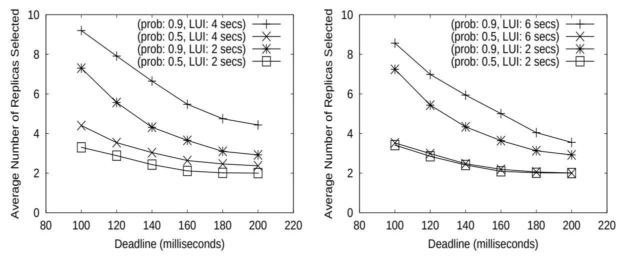
The algorithm receives as inputs the QoS specification, and the list of secondary and primary replicas, along with relevant information about them. For each replica i, the algorithm also receives the values of the replica's immediate and deferred response time distribution functions, which are denoted by $F^I_{R_i}(d)$ and $F^D_{R_i}(d)$. However, for a primary replica i, $F^D_{R_i}(d)$ is not used. The algorithm also receives the staleness factor for the secondary replicas, which is computed using Equation 6. Furthermore, the algorithm 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. While the response time distributions, which are computed from the performance history as explained in Section IV-B, are specific to the individual replicas and are nearly identical in all the client information repositories, the ert information is specific to each client-replica pair and is likely to be different in all the repositories. The algorithm first sorts the replicas in decreasing order of their elapsed response time, ert. This allows the clients to favor the selection of replicas that it used least recently, and thereby alleviate the occurrence of "hot-spots." Replicas that have the same value of ert are sorted in decreasing order of the values of their distribution functions.

The steps of the algorithm are as follows. The algorithm traverses the replica list in sorted order, including each visited replica in the candidate set K, until it includes enough replicas in K to satisfy the terminating condition $P_K(d) \geq P_c(d)$. Each time it includes a new replica, the algorithm invokes the function `includeCDF()` to obtain the value of $P_K(d)$ for the replicas that are currently included in the set K. This function receives as arguments the values of $F^I_{R_i}(d)$ and $F^D_{R_i}(d)$ for the newly included replica, and uses them to compute the value of $P_K(d)$ according to Equation 3. 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. Finally, in the case of sequential ordering, the selected set K is extended to include the sequencer, which is responsible for broadcasting the global sequence number.

Notice that when evaluating $P_K(d)$, we exclude the response time distribution of the member, `maxCDFReplica`, that has the highest probability among the selected members, of responding by the requested deadline. We do this in order to choose a subset that can meet a client's time constraint despite a single replica failure. We propose that if we can choose a set of replicas that can satisfy the timing constraint with the specified probability despite the failure of the member that has the highest probability of meeting the client's deadline, then such a set should be able to handle the failure of any other member in the set. In [5] we have provided a formal justification for this proposal. The above exclusion in effect simulates the failure of the replica with the highest probability of meeting the client's deadline among the selected replicas, and therefore allows us to tolerate single replica failures. Although Algorithm 1 addresses only single replica crashes, it should be possible to extend it to handle multiple replica crashes.

V. Experimental Results

We conducted experiments to evaluate the performance of our selection approach and experimentally analyzed the tradeoffs between timeliness and consistency, using the sequential and FIFO ordering handlers we implemented in AQuA. In this section, we discuss the results we obtained. Our experimental setup is composed of a set of uniprocessor Linux machines distributed over a 100 Mbps LAN. The overhead of the probabilistic selection algorithm increases with the number of replicas and the size of the sliding window used to record the performance histories. For a sliding window of size 20, we found that the selection overhead increased linearly from 400 microseconds to 1200 microseconds when we increased the number of replicas from 2 to 10. The overhead includes the time to compute the distribution functions and the time to select the replica subset. Computation of the response time distribution function contributes to about 90% of the overhead, while selection of the replica subset using Algorithm 1 contributes to the remaining 10%. The overhead is incurred during each request and is accounted for during replica selection. All confidence intervals for the results we present in the following sections are at a 95% level, and have been computed under the assumption that the number of timing failures follows a binomial distribution [4].



(a) Number of replicas selected: sequential

(b) Number of replicas selected: FIFO

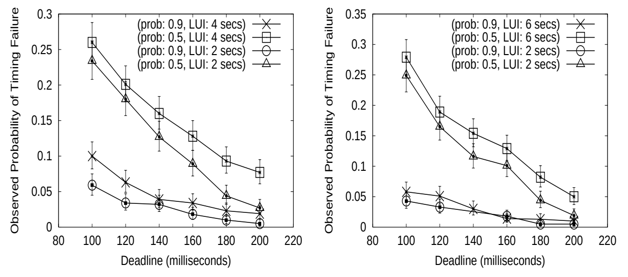
Fig. 3

Sequential vs. FIFO ordering: adaptivity of probabilistic model

A. Effectiveness of Probabilistic Model

One of the main objectives of Algorithm 1 is to assign the replicated servers to service a request adaptively, based on the current responsiveness of the replicas and the QoS requested by the clients. We now present the results of the experiments we conducted to evaluate the adaptivity and effectiveness of the probabilistic selection algorithm for sequential and FIFO ordering. We used an experimental setup composed of 10 server replicas. 4 of the server replicas were in the primary group, and the remaining replicas were in the secondary group. In addition, we had a sequencer when we were using sequential ordering. The service time for all the read and update requests was normally distributed with a mean of 100 milliseconds and a variance of 50 milliseconds. We used two clients that ran on two different machines and independently issued requests to the replicated service with a 1000 millisecond *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 (which we refer to as Client1) requested the same QoS for all of the runs; it specified a staleness threshold of value 4, a deadline of 200 milliseconds, and a minimum probability of timely response of 0.1. The second client (which we refer to as Client2) specified a staleness threshold of value 2 in all the runs, and 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 Client2 in its QoS specification: 0.9 and 0.5. In order to study the effect of the staleness of the replicas on the timeliness of their response, we conducted experiments using different values for the lazy update interval (LUI). The LUI is the periodicity with which the lazy publisher propagates its updates to the secondary replicas. For sequential ordering, we carried out experiments using lazy update intervals of 2 seconds and 4 seconds. In the case of FIFO ordering, we observed nearly identical performances for our experiments using LUI



(a) Probability of timing failures: sequential

(b) Probability of timing failures: FIFO

Fig. 4

SEQUENTIAL VS. FIFO ORDERING: TIMELINESS/CONSISTENCY TRADEOFFS

values of 2 seconds and 4 seconds. Hence, for FIFO ordering, we have shown the results using LUI values of 2 seconds and 6 seconds instead.

We evaluated the adaptivity of our algorithm by determining how the degree of redundancy chosen to service a request varies with the requested QoS. In our experiments we did this by measuring the average number of replicas selected by the selection algorithm to service Client2, as the client varied its QoS specification. Figure 3a shows the results for sequential ordering and Figure 3b shows the results obtained for FIFO ordering. From the graphs we observe that, regardless of the ordering, the number of selected replicas decreases as the requested deadline varies from 100 milliseconds to 200 milliseconds. Similarly, the number of selected replicas decreases as the requested probability of timely response reduces from 0.9 to 0.5. Another observation from these two figures is that for the same QoS specification and the same LUI values, the number of replicas selected in the case of sequential ordering is higher than in the case of the weaker, FIFO ordering. All of these observations lead us to conclude that as the QoS specification becomes less stringent, Algorithm 1 is able to adapt by choosing no more than the number of replicas that are needed in order to meet the client's QoS requirement, as predicted by the probabilistic model. The less stringent a client's QoS specification is, the higher the probability that a chosen replica will meet the client's specification, and hence the fewer the replicas required to meet the specification.

We evaluated the effectiveness of our probabilistic model by experimentally determining if the selected replicas were indeed able to meet the QoS specification of the clients. As mentioned in Section IV, our algorithm guarantees that a client will always receive a response that is within the staleness threshold that it specifies. However, the algorithm provides only probabilistic guarantees about meeting the temporal requirement. In our experiments, we used the observed probability of timing failures as the metric to assess the effectiveness of our model. For each of the QoS specifications of the second client, we experimentally computed the probability of timing failures in a run by monitoring the number of requests in the run for which the client failed to receive a response within the requested deadline. This monitoring was done by the *timing failure detector*, which is a component of the client's gateway handler. Figure 4 shows how successful the selected replicas (shown in Figure 3) were in meeting the QoS specifications of Client2 for sequential and FIFO ordering.

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

A second observation from Figures 4a and 4b is that as the interval between lazy updates increases, the observed probability of timing failures also increases. That is because the replica's state becomes increasingly stale as the lazy updates become less frequent. That, in turn, increases the probability that a chosen replica may have to defer its response until it has received the next lazy update, in order to meet the client's staleness threshold. The delayed response results in a higher probability of timing failures. A final observation from Figure 4 is that for the same QoS specification and lazy update frequency, the timing failures observed in the case of sequential ordering is higher than in the case of FIFO. The reason is that in sequential consistency, we allow both clients to update the same replicated state. The common updates cause the replicated state to become obsolete faster, thereby increasing the probability of a delayed response. Therefore, in order to achieve timing failure probabilities that are closer to those obtained using the FIFO handler, we need to propagate the lazy updates more frequently when sequential ordering is used.

B. *Single-Tier vs. Hierarchical Replica Organization*

In Section III-B, we mentioned that our motivation for using a two-tier replica organization was to favor read operations that can tolerate relaxed consistency to a certain degree, in exchange for better responsiveness. Our thesis was that by limiting the writes to a small subset of replicas, we can reduce the occurrence of timing failures by allowing the remaining replicas to service the read operations, which we assume to be more frequent than the write operations. However, one may argue that a write-all scheme that writes to all the replicas concurrently may result in fewer timing failures. The reason is that unlike the

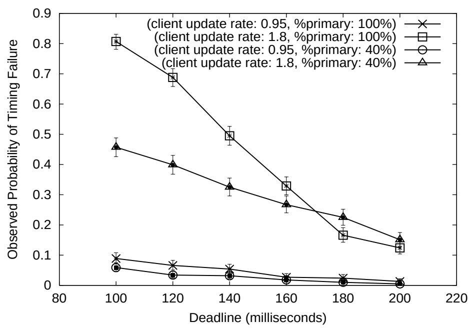
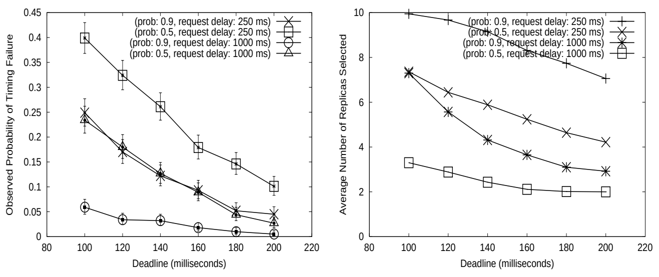


Fig. 5

SINGLE-TIER VS. HIERARCHICAL REPLICA GROUPS



(a) Probability of timing failures (b) Number of replicas selected

Fig. 6

SEQUENTIAL CONSISTENCY: VARIABLE REQUEST DELAY

replicas in a two-tier scheme, the replicas in a write-all scheme can respond without waiting for a state update.

Figure 5 presents the results of the experiments we carried out to compare the performance of the two schemes under different scenarios. We used the same setup with 10 replicas and 2 clients, as explained in Section V-A. In the two-tier organization, 40% of the replicas were in the primary group, and the lazy update interval for updating the secondary replicas was 2 seconds. For the write-all scheme, we used a single-tier organization in which all the replicas were designated as primaries. We carried out our experiments with two clients having the same QoS specification that was used for the results presented in Section V-A. Figure 5 compares the probability of timing failures observed for different update rates using the sequential handler. These failure probabilities were measured for Client2, which varied its deadline from 100 milliseconds to 200 milliseconds and requested a probability of timely response of at least 0.9.

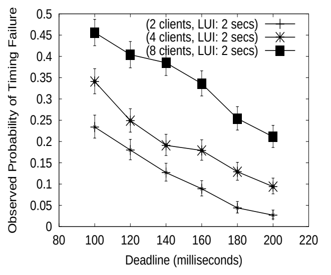
From the results in Figure 5 we see that when the arrival rate of the updates from the clients is small (about 1 update per second), the performance of the two schemes is nearly identical. However, when the update arrival rate of the clients is almost doubled, the write-all scheme results in a significantly higher number of timing failures. The reason is that as the update rate increases, the client-induced load on all the replicas in the write-all scheme increases. Although the replicas in the write-all scheme do not have to wait for a state update in order to respond to a request, they have to wait for the previous write to complete before they can respond. Since all the replicas are involved in writes, the response time for the following read operations is higher. On the other hand, in the case of the two-tier organization, only 40% of the replicas are involved in write operations. Although some of the remaining 60% may have to defer their response until they receive the next state update, the value of the lazy update interval we have chosen ensures that the probability of that is small. We observed similar results comparing the two schemes using FIFO ordering guarantees. The above results justify the choice of a hierarchical replica organization to support relaxed consistency models. Although in this work we restrict ourselves to a two-tier organization of replicas to study the tradeoffs between timeliness and consistency, it should be easy to extend this architecture to multiple tiers representing intermediate degrees of staleness in the replica states.

C. *Performance Under Load*

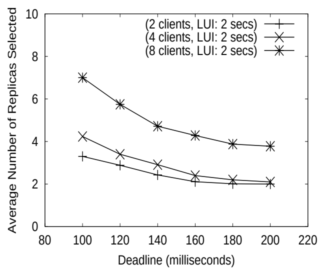
We now describe the experiments we carried out to determine how well our model adapts to meet the client's QoS specification under different client-induced loads. We varied the client-induced load by varying the request delay and the number of clients accessing a service. When we varied the client-induced load by varying the request delay, we used the same experimental setup with two clients that we described in Section V-A. The induced load on the servers is higher for smaller request delays. Figure 6 presents the results for sequential ordering, using a lazy update interval of 2 seconds, for two different values of the request delay: 1000 milliseconds and 250 milliseconds. The graphs in Figure 6a compare the observed timing failure probability, while the graphs in Figure 6b compare the average number of replicas selected.

The first observation from Figure 6 is that the observed failure probability increases as the request delay reduces from 1000 milliseconds to 250 milliseconds. That is because as the request delay 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 6 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 request delay is 250 milliseconds, the replica subset chosen by the probabilistic scheme is unable to meet deadline values $\leq$ 120 milliseconds with a probability $\geq$ 0.9, although the request is sent to all 10 available replicas, as shown in Figure 6b. 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.

We also varied the client-induced load by varying the number of clients accessing a service concurrently. The induced load on the servers increased with the number of clients. All the clients



(a) Probability of timing failures



(b) Number of replicas selected

Fig. 7
SEQUENTIAL CONSISTENCY: VARIABLE NUMBER OF CLIENTS

used the same request delay value of 1000 milliseconds. One of the clients specified a different deadline in each run and requested that its deadline be met with a probability ≥ 0.5. All of the remaining clients specified a deadline of 200 milliseconds in each run and requested that this deadline be met with a probability ≥ 0.1. The plots in Figure 7 evaluate the performance of the probabilistic scheme using 2, 4, and 8 clients when the lazy update interval is 2 seconds. Figure 7a shows the timing failure probability for each case, as measured at the client that specified that its probability of timely response should be at least 0.5, and Figure 7b shows the corresponding average number of replicas selected by the probabilistic scheme to meet the QoS specifications of this client. 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 in each case, the model was able to adapt appropriately to select a subset of replicas that could meet the client's QoS specification.

VI. CONCLUSIONS

The experimental results we have presented show that the model we developed to assign the replicated servers to service the clients allows a QoS-aware middleware to adapt the assignment based on changes in the load and the QoS requirements of the clients. The model also helped in understanding the tradeoffs between timeliness and consistency for different consistency semantics. The results we presented show that the frequency of lazy updates is an important parameter that allows us to tune the tradeoffs between the desired levels of consistency and timeliness. Some of the factors that need to be considered when choosing the frequency include the arrival rate of the updates from the clients, the ratio between the size of the primary and secondary replica groups, and the timeliness, as well as the consistency specification of the clients.

VII. FUTURE EXTENSIONS

Our research has demonstrated the need for feedback and the efficacy of simple analytical models that map a user's QoS specification onto the underlying properties of the resources. Although our probabilistic approach was mainly developed to adaptively share replicated servers in uncertain environments, similar techniques can be applied to a range of systems problems, including scheduling and other resource allocation problems. Given the increasing demand for different services and the diversity of the requirements of client applications, 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.

One of the limitations of our work is that it measures staleness in terms of "versions" or logical clocks. We made that choice to avoid the dependence on synchronized clocks. However, in environments in which clocks are synchronized, we can modify our approach to measure staleness in terms of real-time clocks. We can then use our modified approach to allow users of real-time, distributed database applications, such as traffic-monitoring, online stock-trading, and electronic patient recording systems in a hospital, to access information that is no older than a specified interval of time, within a certain deadline, probabilistically. Another limitation is that we currently admit all the clients, and only after the timing failures have been detected, inform a client if the observed failure probability exceeds its expectations. However, with some modifications, we could instead use our framework to perform admission control, in order to determine which clients 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.

REFERENCES

[1] K. Birman. *Building Secure and Reliable Network Applications*. Manning, 1996.
[2] I. Foster, A. Roy, and V. Sander. A Quality of Service Architecture that Combines Resource Reservation and Application Adaptation. In *Proc. of the 8th International Workshop on Quality of Service*, 2000.
[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] 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*, June 2002. To appear.
[7] L. Lamport. Time, Clocks, and the Ordering of Events in Distributed Systems. *Communications of the ACM*, 21(7):558–565, July 1978.
[8] B. Li. *Agilos: A Middleware Control Architecture for Application-Aware Quality of Service Adaptations*. PhD thesis, University of Illinois, 2000.
[9] 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.
[10] A. Vaysburd. *Building Reliable Interoperable Distributed Applications with Maestro Tools*. PhD thesis, Cornell University, May 1998.
[11] 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.