# A Dynamic Replica Selection Algorithm for Tolerating Timing Faults*

Sudha Krishnamurthy, William H. Sanders, and Michel Cukier
Center for Reliable and High-Performance Computing
University of Illinois at Urbana-Champaign, Urbana, Illinois 61801

E-mail: {krishnam,whs,cukier}@crhc.uiuc.edu

## Abstract

*Server replication is commonly used to improve the fault tolerance and response time of distributed services. An important problem when executing time-critical applications in a replicated environment is that of preventing timing failures by dynamically selecting the replicas that can satisfy a client's timing requirement, even when the quality of service is degraded due to replica failures and excess load on the server. In this paper, we describe the approach we have used to solve this problem in AQuA, a CORBA-based middleware that transparently replicates objects across a local area network. The approach we use estimates a replica's response time distribution based on performance measurements regularly broadcast by the replica. An online model uses these measurements to predict the probability with which a replica can prevent a timing failure for a client. A selection algorithm then uses this prediction to choose a subset of replicas that can together meet the client's timing constraints with at least the probability requested by the client. We conclude with experimental results based on our implementation.*

## 1. Introduction

Server replication is a popular approach for building fault-tolerant distributed services (e.g., [1, 7, 12, 3, 14, 6]). Replication is also a commonly used solution for improving the scalability of a distributed service, i.e., to ensure that the response time of a service does not significantly degrade with an increase in the number of clients accessing the service (e.g., [10, 13, 4]). Achieving both fault tolerance and scalability at the same time, however, is a challenging goal, especially when the number of available replicas is constrained. We can achieve good fault tolerance by allocating all the available replicas to service a single client. 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 multiple clients to be serviced in parallel. 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. Thus, in order to build a dependable service, we need a method that attempts to prevent the occurrence of such timing failures for a client by selecting replicas from the available replica pool, based on an understanding of the client's timing requirements and the responsiveness of the replicas. The research described in this paper presents the approach we have used to realize this goal.

Several other replica selection algorithms have been formulated with the objective of choosing the replica that can deliver the lowest possible response time. These algorithms often target clients of stateless, distributed services, such as the World Wide Web, in which the servers do not maintain any records of ongoing client transactions. Some of these algorithms choose the nearest replica based on a distance metric [9], and some choose the replica with the best historical average response time [19]. Some predict the time to propagate a message to different replicas using regression analysis of previously collected data [2] and pick the replica that has the lowest future propagation time. Finally, some of them actively monitor both replica load and network delays, use these to estimate the response times of the replicas, and select the replica that has the smallest estimated response time [5]. All of these efforts assign a single replica to each client and do not consider the case in which a replica may fail while servicing a request. As such, it is the responsibility of the client to retransmit its request upon failure to receive a response. Such a simple retransmission strategy, however, may not be suitable for clients with specific time constraints.

In contrast, our work targets clients that have specific response time requirements and require that these be met with

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



0-7695-1101-5/01 $10.00 © 2001 IEEE

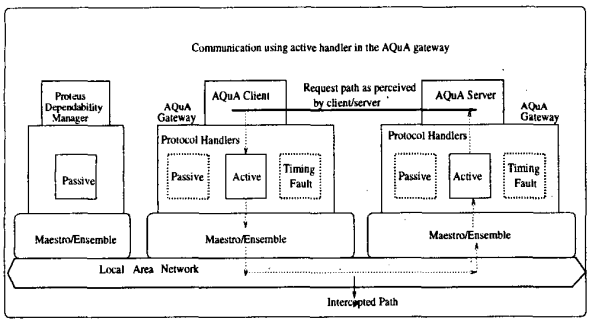


**Figure 1. Communication using AQuA gateway handlers**

certain probabilities. As in the above efforts, our work also targets clients of stateless applications such as search engines and radar-tracking applications. However, the replicas in our case are distributed across a local area network (LAN). The approach we use first estimates a replica's response time distribution based on performance measurements regularly disseminated by the replica. An online model uses these measurements to estimate the probability with which the replica can prevent a timing failure for a client. A selection algorithm uses this estimate to choose a subset of available replicas that can together meet a client's timing constraints with at least the probability requested by the client. Each replica in the selected set independently processes the request and sends its response. However, only the earliest reply is delivered to the client. The selected subset is chosen in such a way that the client's probabilistic timing requirement can be met even when one of the members in the selected subset crashes before responding to the request. We have implemented our algorithm in AQuA, a CORBA-based middleware that supports transparent replication.

The rest of this paper is organized as follows. Section 2 presents an overview of AQuA. Section 3 describes our assumptions and our system model. In Section 4 we describe the replica the selection problem. In Section 5 we present the dynamic replica selection algorithm that we have developed to tolerate timing faults in AQuA. We present experimental results based on our implementation in Section 6. In Section 7 we compare our work with some of the related efforts, and finally, in Section 8 we present our conclusions.

## 2. Overview of AQuA

Figure 1 presents an overview of the AQuA middleware. Proteus, a component of AQuA [3], enhances the capabilities of CORBA [15] objects to provide fault tolerance for distributed applications. Fault tolerance is provided by transparently replicating objects using *active* and *passive* replication. The Proteus *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 protocol stack [20, 8], above which AQuA is layered. Maestro-Ensemble also detects and notifies the members of changes to the group membership. The use of group communication in AQuA is transparent to the end applications. Hence each of the clients, which are all CORBA objects, is given the perception that it is communicating with a single server object using CORBA's IIOP [15]. This is achieved using an AQuA gateway, which transparently intercepts a local application's CORBA message and forwards it to the destination replica group through Maestro-Ensemble, as shown in Figure 1. For the sake of clarity, in this figure we have illustrated a server replica group having only a single member. In reality, this group may have multiple replica members.

The different replication schemes supported by AQuA are implemented as *protocol handlers* within the gateway. An AQuA client uses different gateway handlers to communicate with different server groups. These handlers are responsible for tolerating different kinds of faults. Previous work in AQuA has addressed the issue of tolerating crash failures using the active [18] and passive [17] handlers. [16] also discusses how AQuA simultaneously tolerates value faults and crash failures using an active handler. In this paper, we describe the approach we have used to tolerate timing faults, which has resulted in the development of the timing fault handler.

## 3. System Model

Given this overview of AQuA, we now describe the system for which we want to solve the dynamic replica selection problem. The machines hosting replicated services in this system are distributed across a local area network (LAN). A machine may host multiple replicas. The services in this system are frequently accessed by several clients concurrently. Clients requesting the use of these services demand specific response time guarantees that have to be met with a certain probability. Failure to receive a response for a request within the specified time results in a *timing failure* for the client.

We assume that the load on a replica may fluctuate and that periods of high load may make it less responsive. We also assume that while the links in the LAN connecting the system do not experience frequent fluctuations in traffic, they may experience occasional periods of high traffic, which may result in large delays in the message delivery

time. Finally, a replica may crash, making it unresponsive. Any of these factors may contribute to a timing fault.

## 4. Problem Description

Given the above sources of timing faults, the problem we address is that of finding a way to reduce the occurrence of timing failures by servicing as many requests as possible in a timely manner. We achieve this by devising an approach that will allocate the replicas to the clients based on their response time requirements. We now state how a client expresses its timing constraints and then outline the decisions that have to be made when allocating the replicas.

A client which requires that a service respond to its request within a specific time, expresses its requirements as a quality of service (QoS) specification. The client may either specify its QoS requirement at start-up time, or negotiate it at runtime as often as it wants. This specification includes the name of a service, the time by which the client wants to receive a response after it transmits its request to this service, and the minimum probability with which it wants this time constraint to be met. If a response does not meet this time constraint then it results in a timing failure for the client. If the frequency of timing failures is so high that the system is unable to deliver timely responses with at least the minimum probability that the client has specified, then the client receives a notification through a callback.

Our research objective is to reduce the occurrence of timing failures under normal conditions as well as when the responsiveness of a service is reduced, either due to failure of its replicas or due to the load induced when multiple clients with different QoS requirements access a service over a period of time. We achieve this objective by designing a request scheduler that transparently intercepts a client's request, estimates the response time of the different replicas offering the service that the client has requested, and selects a subset of available replicas that can meet the client's response time requirements with a probability at least as high as that requested by the client. The scheduler uses historical performance data collected at runtime as inputs to solve a probabilistic model, which estimates the probability that a response will be received on time. The scheduler then forwards the request to the selected replicas. Each of the selected replicas independently services the request and sends back its response. However, only the earliest response is delivered to the client. In this paper, we describe the design of a distributed scheduling system within the AQuA middleware, in which each client is associated with a local scheduling agent that makes the replica selection decisions on the client's behalf.

In a system in which a replica's responsiveness may change unpredictably due to either load or crashes, like the system we have described in Section 3, it is impossible for the scheduler to predict with certainty whether any single replica can meet a client's timing constraint. In order to satisfy our goals of providing a scalable service while at the same time providing a reasonable level of fault tolerance, an important decision our scheduler has to make is that of choosing the redundancy level with which a request has to be serviced. The scheduler we have designed makes its decisions adaptively based on the probability with which the individual replicas will meet the client's timing constraint. The higher the probability that the chosen replicas will meet the constraints, the lower is the redundancy level.

## 5. Dynamic Replica Selection in AQuA

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

### 5.1. Factors Influencing the Response Time

Figure 2 shows the stages along the path traversed by a typical request from an AQuA client that has specific timing constraints. In Stage1, an AQuA client invokes a remote method using CORBA's IIOP [15]. The request is then intercepted by a protocol handler in the AQuA gateway. The handler marshals the request into a Maestro message, and in Stage2 presents it to the Maestro/Ensemble protocol stack, from where it is transmitted across the network to the server gateway. This gateway-to-gateway communication may use point-to-point or multicast communication depending on the number of replicas to which the client request is forwarded. In Stage3, the protocol handler in the server gateway receives the Maestro message, demarshals it into a CORBA message, and enqueues it in the request queue of the server application using CORBA's dynamic invocation interface (DII) [15]. The server uses FIFO ordering for servicing the requests in the queue. After the server services the request in Stage4, it sends its response to the client. The protocol handler in the server gateway intercepts this response and forwards it to the client gateway along the Maestro/Ensemble protocol stack. The client gateway delivers the earliest response it receives for a request by making a CORBA upcall to the AQuA client. We conducted experiments to determine the factors that have

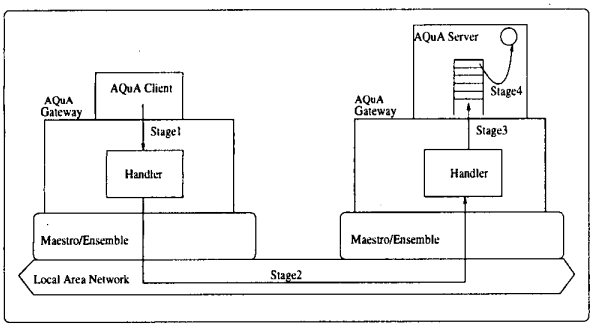


**Figure 2. Stages along the path traversed by a request in AQuA**

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

**Gateway-to-Gateway Delay:** the time for an AQuA request or response embedded within a Maestro message to travel between two AQuA gateways, as in Stage2 of Figure 2. From the figure, we see that this delay includes the time for a message to travel through the Ensemble/Maestro protocol stack and the time for it to travel on the wire across the LAN. This delay is incurred on both the request and response paths and the two delays together make up the two-way gateway-to-gateway delay. For a message of a given size, this delay varies mainly with the load on the network and the number of group members involved in the communication.

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

**Service Time:** the time spent by the server to process the request after dequeuing it from the request queue. For requests that are of the same kind, this time mainly varies with the load on the host.

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

## 5.2. Gateway Information Repository

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

As mentioned in Section 2, an AQuA client uses different gateway handlers to communicate with different servers. In other words, the gateway handler identifies the server group with which a client is communicating. Thus, a client that is communicating with multiple servers would have multiple handlers loaded in its gateway. We leverage this design and associate an information service with each timing fault handler within a client's gateway. Although this design has the drawback that the replica-related information is redundantly stored at multiple client gateways, it has several advantages compared to a global information service, which would avoid this drawback. First, having a repository local to each client handler avoids a single point of failure. Second, it avoids the overhead of making a call to a remote information service. Third, allowing each gateway handler to access its local repository avoids the need to enforce concurrency control, which would otherwise result in high access overheads. Finally, since a repository local to a handler only caches information relevant to the service associated with that handler, the search space is smaller, and as a result it takes less time to access information.

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

## 5.3. Model-Based Replica Selection Algorithm

Using the performance measurements collected above as inputs, the local scheduler that is part of a timing fault handler builds a model to predict the probability that a subset of

replicas will be able to meet a client's timing requirements. The selection is then done based on this resultant probability. We first define the notation we use to present the model:

- $M = \{m_1, m_2, ..., m_n\}$ is the set of replicas offering the service requested by a client. The scheduler obtains this set from its local information repository.
- $R = \{R_1, R_2, ..., R_n\}$, $R_i$ is the random variable denoting the time to receive a response from a replica $m_i \in M$, after a request was transmitted to it.
- $P_c(t)$ is the probability with which the client wants a response for its request by time $t$, as described in Section 4.

We now need to determine the probability that a response from a subset $K \subseteq M$, consisting of $k > 0$ replicas, will arrive by the client's deadline, $t$, and thereby avoid the occurrence of a timing failure. We denote this probability by $P_K(t)$. As stated earlier in Section 4, each replica in the subset independently processes a request and sends back its response. However, only the first response received for a request is delivered to the client. So a timing failure occurs only if no response was received from any of the replicas in the set $K$ within $t$ time units after the request was sent. Computing the distribution of the time until a response is received is straightforward if we assume that the response times of individual replicas are independent of one another. While this is not strictly the case in a shared network where the network delays may be correlated, we believe it is a reasonable assumption to make, since the network delay is usually a small fraction of the replica's response time in a LAN environment. We use this independence assumption to compute the probability, $P_K(t)$, for the replicas in subset $K$, as follows:

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

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

$$P_K(t) = 1 - \prod_{m_i \in K} (1 - F_{R_i}(t)) \quad (1)$$

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

**5.3.1. Computing the Response Time Distribution.** Given the above model, we now explain how we compute the value of $F_{R_i}(t)$ for a replica $m_i$. Henceforth, we will use the subscript $i$ to refer to the replica $m_i$. Based on the analysis presented in Section 5.1, we define the response time random variable, $R_i$, using Equation 2 below.

$$R_i = S_i + W_i + T_i \quad (2)$$

where $S_i$ is the random variable denoting the service time for a request serviced by $m_i$; $W_i$ is the random variable denoting the queuing delay experienced by a request waiting to be serviced by $m_i$; and $T_i$ is the random variable denoting the two-way gateway-to-gateway delay between the client and replica $m_i$. For each request, we experimentally measure the values of the service time, $S_i$, as described later in Section 5.4, and record the values of the most recent $l$ requests in the *service time vector* in the information repository. We do the same for the queuing delay, $W_i$, and record its recent values in the *queuing delay vector* in the information repository. Thus, these vectors represent a sliding window, $L$, of size $l$, over which the performance history is recorded. For the gateway-to-gateway delay, $T_i$, we decided to use its most recently measured value rather than recording its history over a period of time. This decision was based on the observation that the traffic in a LAN does not frequently fluctuate like the other two parameters. We verified this observation to be true for the environment we used. For environments in which this observation is not true, it would be simple to extend our approach to record the value of the gateway-to-gateway delay over a sliding window as we do above for the service time and queuing delay.

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

**5.3.2. Replica Selection Algorithm.** Given the ability to compute the probability that an individual replica will meet a client's time constraint, we now describe the algorithm that applies Equation 1 to select a set of replicas that can meet this time constraint with the probability the client has requested. The selection algorithm is outlined in Algorithm 1. The algorithm first sorts the replicas in decreasing order of the probability that they can individually meet the client's response time requirement. In Line 4, it includes the first element of this sorted replica list in the selected set, $K$. It then considers the remaining replicas in this list in sorted order, including each replica in the candidate set $X$, until it includes enough replicas in $X$ such that the condition $P_X(t) \geq P_c(t)$ is satisfied, where $P_X(t)$ can be computed using Equation 1. In Line 11, we extend this candidate set $X$ by including the first element, $m_0$, which was selected in Line 4, to form the final selected set of replicas,

$K$. Thus, we include the replica, $m_0$, that has the highest value of $F_{R_0}(t)$, in the final selected set, although we do not consider it when testing the condition in Line 10. We now explain the reason for this.

Since replicas may crash, our goal is to choose a set of replicas that can meet a client's time constraint with the probability the client has requested, even when one of the replicas in the selected set crashes before servicing the request. Our intuition is that if we can choose a set of replicas that can satisfy the timing constraint with the specified probability despite the failure of the member, $m_0$, which has the highest probability of meeting the client's deadline, then such a set should be able to handle the failure of any other member in the set. The loop in Lines 6-14 of Algorithm 1 attempts to find such a subset, $X$, that satisfies the condition in Line 10 by excluding the member $m_0$. If it finds such a set, it extends the set to include $m_0$ to form the final set, $K$. If, however, it is unable to find such a set, then it returns the complete set of available replicas, $M$. We now justify that the set $K$ found by Algorithm 1 does indeed handle single replica crashes. Let $g_0 = 1 - F_{R_0}(t)$, where $F_{R_0}(t)$ is the distribution function of the first member in the sorted list. Since $F_{R_0}(t) \geq F_{R_i}(t), \forall i$, we have,

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

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

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

$$1 - (g_0 * g_1 * \cdots * g_{i-1} * g_{i+1} * \cdots * g_x) \geq P_c(t) \quad (3)$$

Equation 3 follows from the condition in Line 10 of Algorithm 1. This equation shows that should any one of the members, $i$, belonging to the selected set $K$ crash without completing its transaction, the other members should still be able to meet the client's timing constraint with the probability the client has requested.

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

**5.3.3. Algorithm Overhead.** In a practical implementation, the overhead incurred by the selection algorithm has to be considered by modifying Algorithm 1 to select those replicas that can respond within $t - \delta$ time units rather than $t$ time units, where $t$ is the client's deadline as before, and $\delta$ is the overhead of the algorithm. As seen from Algorithm 1, the overhead mainly depends on the number of replicas, $n$, and the size of the sliding window, $l$, that we use to record the performance measurements broadcast by the replicas. In our implementation, we measure this overhead, $\delta$, each time the selection algorithm is executed, and use the most recently measured value of $\delta$ to compute the value of $F_{R_i}(t - \delta)$. We then include this overhead by merely modifying Algorithm 1 to use the value of $F_{R_i}(t - \delta)$ wherever it uses the value of $F_{R_i}(t)$. The rest of the algorithm remains unchanged.

**Algorithm 1** Replica Selection Algorithm

**Require:** $V =< i, F_{R_i}(t) >$ {set of replicas and their corresponding distribution function}
**Require:** Client Inputs:
$t$ : client's deadline,
$P_c(t)$: probability that this deadline should be met
1: $X \Leftarrow \phi$
2: $prod \Leftarrow 1$
3: sortedList $\Leftarrow$ sort V in decreasing order of $F_{R_i}(t)$
4: $K \Leftarrow [first(sortedList)]$ {always include the replica that has the highest probability in the selected list}
5: newSortedList $\Leftarrow$ sortedList - $K$
6: **for all** i in newSortedList **do**
7: $X \Leftarrow X \cup i$
8: $g_i \Leftarrow 1 - F_{R_i}(t)$
9: $prod \Leftarrow prod * g_i$
10: **if** $1 - prod \geq P_c(t)$ **then**
11: $K = X \cup K$
12: return $K$ {found an acceptable replica set}
13: **end if**
14: **end for**
15: return $M$ {return the set comprising all the replicas}

## 5.4. Design of the Timing Fault Handler

Given a QoS specification from a client as described in Section 4, we now explain how the timing fault handler tries to meet the client's response time requirements by making use of the above selection algorithm. A client may either negotiate its QoS requirements at runtime or specify them in a configuration file, which is read by the timing fault handler when it is loaded in the client gateway. The QoS requirement a client requests from a service is stored in the handler the client uses to communicate with that service. When a client makes a request to that service, the handler uses this QoS specification to select the set of server replicas to process the request.

The timing fault handler uses the Maestro-Ensemble group communication layer to manage communication transparently between a client application and a replicated service. Before they can communicate, the client and server applications should join the same *multicast group*. This multicast group is similar to a *connection group* [3] in AQuA except that it allows a message to be sent to a specified list of members in a group rather than be broadcast to all group members. The timing fault handler uses the multicast group for forwarding requests from a client to a selected subset of server replicas, as will be explained in the next subsection. The client handlers that are interested in receiving performance updates from the servers use this group to multicast their subscription request to the server replicas. Each server replica then keeps track of its subscribers and notifies them whenever its performance parameters change. This information, pushed from the server replicas, is then used to update the client's gateway information repository, as will be explained in further detail in the next subsection. When a member of a multicast group crashes, Maestro-Ensemble detects the failure and notifies all the group members about the change in the membership. This allows those clients that are members of the group to remove the entry for the failed replicas from their local information repositories. These failed replicas will therefore not be considered in the selection process for future requests.

**5.4.1. Request-Response Handling.** A typical request-response is processed by the timing fault handler as follows. After transparently intercepting a request from a client, the client's timing fault handler records the interception time, $t_0$, and hands over the request to its scheduler module. The scheduler first retrieves the replica list for the service from its local gateway information repository. If the service has never been accessed before, the information repository would not contain any performance data for the replicas offering that service. In this case, the selection strategy selects all the replicas in the list. This allows the replicas to publish their performance updates to the clients, as described below, and thereby initialize the information repositories. During subsequent requests, the scheduler uses this performance history from its local information repository to choose the replicas based on the client's QoS requirements using the selection strategy explained in Section 5.3. The handler then multicasts the client's request to the selected replicas using Maestro-Ensemble and records the sequence number of the message and its time of transmission, $t_1$.

Upon receiving the request, the timing fault handler at the server enqueues the request in the replica's request queue as shown in Stage3 of Figure 2. It then records the time, $t_2$, at which the request is enqueued. The AQuA gateway asynchronously processes the request queue in FIFO order. When the request is dequeued for service, the gateway records the dequeue time, $t_3$, before invoking the server application to service the request using CORBA's dynamic invocation interface [15]. When the server sends its response back to the client, the timing fault handler intercepts the response and records the service duration, $t_s$. The server's handler then forwards the reply back to the client gateway along with the performance data, which includes the service duration $t_s$, and the time, $t_q$, spent by the request in the queue, where $t_q = t_3 - t_2$. The handler publishes this new performance data, along with the replica's current queue length, to all its subscribers. This information is used by the subscribers to update their local information repositories. In our current implementation, the server publishes its performance update to its subscribers, each time it processes a request.

When the client handler receives a reply from a replica, it records the time of reception, $t_4$, and extracts the performance data embedded in the message. If the reply is the first one it has received for a request, the handler delivers the reply to the client. The handler then uses the extracted performance data to measure the new round-trip gateway-to-gateway delay, $t_d$, between the client and replica. This delay, $t_d$, is given by $t_d = t_4 - t_1 - t_q - t_s$, where $t_q$ and $t_s$ are obtained from the extracted data. The handler then updates the information in its local repository with this new value of the gateway-to-gateway delay.

Since we allow a request to be processed redundantly by multiple replicas, the client gateway may receive multiple responses for the same request. The client gateway, however, does not deliver any of the redundant replies to the client. Instead it merely discards them and uses the performance data it extracts from each of them to update its information repository with the new value of the gateway delay, in the same manner as it did for the first reply.

**5.4.2. Detecting Timing Failures.** We now explain how the timing fault handler detects timing failures and handles them when they occur. The handler maintains a counter that keeps track of the number of times its client has failed to receive a timely response from a service. When the handler receives the first reply for a request sent by its client to a service, the handler checks whether a timing failure has occurred by computing the response time, $t_r = t_4 - t_0$, where $t_4$ is the time at which the first reply arrived at the handler, and $t_0$ is the time at which the handler intercepted the request from its client. A timing failure occurs if $t_r > t$, where $t$ is the response time requested by the client. If the handler detects that a failure has occurred, it updates its counter. If the frequency of timely responses from the service does not meet the minimum probability the client has requested in its QoS specification, the handler notifies the client by issuing a callback. The client can then either choose to renegotiate its QoS specification or issue its re-

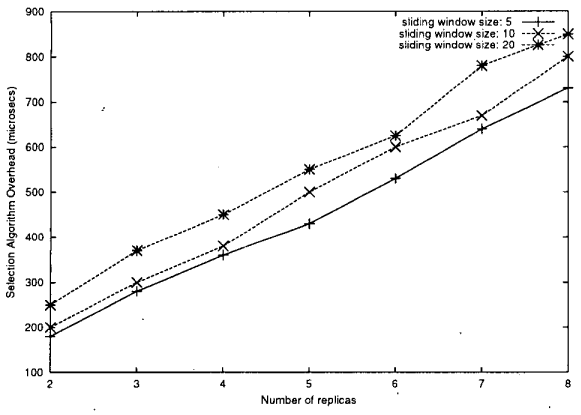


**Figure 3. Overhead of replica selection algorithm**

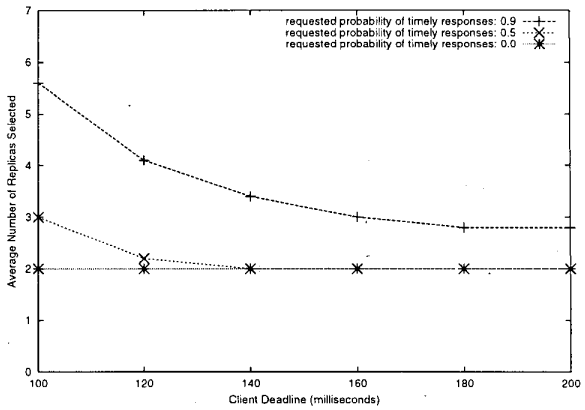


**Figure 4. Comparison of the number of selected replicas**

quests to the service at a later time. Note that when we collect the timing data as explained above, we do not require that the clocks be synchronized because we always measure the two end-points of a timing interval on the same machine.

## 6. Experimental Results

We conducted experiments using our implementation of the timing fault handler in AQuA to analyze the performance of the selection algorithm. Our experimental setup is composed of a set of uniprocessor Linux machines distributed over a LAN. For a minimum-sized request having negligible service time, the minimum value we achieved for the response time, $t_r$ (defined in Section 5.4.2), was about 3.5 milliseconds. Figure 3 shows how the overhead of the selection algorithm varies with the number of replicas for three different sizes of the sliding window : 5, 10, and 20. These overheads include the time to compute the distribution function and the time to select the replica subset. These overheads are incurred during each request. Computing the distribution function contributes to 90% of these overheads while selecting the replica subset using Algorithm 1 contributes to the remaining 10%. For our experiments below, we used a sliding window of size 5.

We also conducted experiments to evaluate how effectively the subset of replicas chosen by the model-based selection algorithm was able to meet a client's deadline with the probability requested by the client. To do this, we used two clients that ran on different machines and independently issued requests to the same service with a one second delay between receiving a response and issuing the next request. The number of server replicas available for selection during each experiment was seven. Each server replica ran on a different machine and responded with an integer data. Since the machines in our testbed had insignificant background load on them, we simulated the load on the servers by having each replica respond to a request after a delay that was normally distributed with a mean of 100 milliseconds and a variance of 50 milliseconds. In every run, each of the two clients issued fifty requests to the service. One of the clients requested a deadline of 200 milliseconds in each run and specified that this deadline be met with a probability $\geq 0$. The second client requested a different deadline in each run. For each of these deadline values of the second client, we computed the probability of timing failures in a run of fifty requests by measuring the number of responses in the run that had failed to arrive by the deadline specified by the second client. In order to study the behavior of the dynamic selection algorithm for different values of the probability of timely responses specified by a client, we repeated these experiments for three different probability values specified by the second client: 1) a probability value of 0.9 , 2) a probability value of 0.5, and 3) a probability value of 0. We chose a probability value of 0 because this represents the case in which the dynamic selection algorithm would achieve the highest timing failure rates. Hence, this case would provide a perspective on the worst-case behavior of the algorithm.

Figure 4 shows the expected number of replicas selected by the dynamic selection algorithm to service the second client for each of its QoS specifications. The first observation from this figure is that as the deadline increases, the algorithm chooses, on the average, fewer replicas to service the client. The second observation from this figure is that the algorithm chooses a lower redundancy level when the client requests a lower probability of timely responses. For

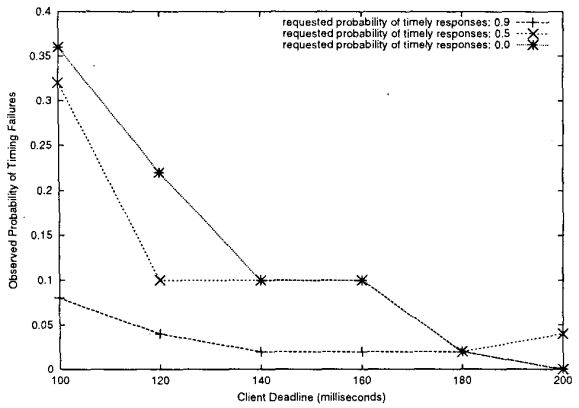


**Figure 5. Validation of the probabilistic model**

example, in the first case in which the client specifies that at least 90% of its responses should be timely, the algorithm chooses a redundancy value as high as 6 to meet some of the client's requests. However, in the third case in which the client is willing to tolerate any number of timing failures, the algorithm chooses only a redundancy level of 2, which is the minimum number of replicas selected by Algorithm 1. The reason for these observations is that our algorithm never selects more than the minimum number of replicas necessary 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. Therefore, as the client's QoS requirements become more flexible, the algorithm can satisfy them with fewer replicas.

Figure 5 shows how successful the selected set of replicas, shown in Figure 4, were in meeting the QoS specifications of the second client. Figure 5 shows that when the client specifies that the probability of timely responses must be at least 0.9, the maximum probability of timing failures we observe experimentally is only 0.08, which is lower than the 0.1 timing failure probability that the client is willing to tolerate. Similarly, for the cases in which the client is willing to accept a failure probability up to 0.5 and 1, we observe a maximum timing failure probability of 0.32 and 0.36, respectively, for the deadline values we used. These results show that, in each case, the set of replicas selected by Algorithm 1 was able to successfully meet the client's QoS requirements by maintaining the timing failure probability well below the failure probability that was acceptable to the client. Thus, for the experimental runs we conducted, the model we used was able to accurately predict the set of replicas that would be able to meet the client's deadline with at least the probability requested by the client.

The above results show that at the expense of some computational overhead, the dynamic selection scheme we have described makes effective use of the available replicas to meet the QoS specifications and thereby reduce the occurrence of timing failures, even when multiple clients access a service at the same time. We believe that such a replica selection scheme would be useful in an environment in which time-critical clients access compute-bound service providers that display variability in their response times.

## 7. Related Work

We now briefly mention a few of the related efforts that address the problem of detecting and preventing timing failures for clients in a distributed system.

The DREAM project [11] provides an integrated object based framework for tolerating crash faults, value faults, and timing faults in a real-time, distributed system by using a primary/backup replication scheme. While the DREAM approach tries to prevent timing failures from occurring as a result of replica crashes, it does not handle timing failures occurring due to the load induced when multiple clients concurrently access a service. Our research goals are related to the work done by Wolfe [22], which also addresses the problem of meeting the time constraints when a CORBA service is accessed by multiple clients. Their approach, however, uses a global scheduling service that assigns a global CORBA priority to a request, based on the timing requirements expressed by the client. This priority is then used to determine the order in which a server services the request. The work done by Wolfe does not, however, address replica crashes. Verissimo and Casimiro have proposed a general architectural construct called the *Timely Computing Base* (TCB) [21] that can verify timeliness and detect time failures, properties that are essential for building dependable and timely services. The timing fault handler we have implemented realizes some of these properties for a replicated service.

## 8. Concluding Remarks

We have presented a new approach that tolerates timing faults in replicated services. This approach uses an algorithm that chooses replicas dynamically at request time, based on their ability to meet a client's time constraints in the presence of delays and replica crashes. An important contribution of this paper is the definition of a probabilistic model to predict, at runtime, the probability that the response from a replica will arrive by a given time, based on the performance updates the model receives from the replica as inputs. This prediction made by a scheduler, which is part of the timing fault handler, is used to select a set of replicas that can meet a client's timing constraint with at

least the probability requested by the client. We have implemented the selection algorithm in AQuA, an infrastructure for building dependable distributed applications, and obtained experimental results that show its efficacy. Our model and selection algorithm can be easily extended to any environment that provides replicated services and supports a mechanism for tracking and recording the recent history of the performance of its replicas.

We now mention a few extensions to our work. First, in this work, we have assumed that the servers export a single method interface to the client. It is possible to extend our work to support the case in which a server exports multiple service interfaces. We can do this by modifying the information repository to classify performance data based on the method interfaces. The selection algorithm can then use the performance information appropriate to the method invoked. Second, in a system in which the middleware has knowledge about an application's request semantics, our selection algorithm can be extended to distinguish between requests made to the same server based on the arguments passed by the clients. Our infrastructure currently does not support this feature. Finally, our work can also be extended to use active probes [5] when a replica's performance information is obsolete.

**Acknowledgments:** We are thankful to the anonymous reviewers for their careful feedback, which helped us to improve our work. We would like to thank Mouna Seri and the rest of the AQuA team for their contributions to the AQuA project. We are also grateful to Jenny Applequist for helping us to improve the readability of the paper.

## References

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

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

[3] M. Cukier, J. Ren, C. Sabnis, et al. AQuA: An Adaptive Architecture that Provides Dependable Distributed Objects. In *IEEE Symposium on Reliable Distributed Systems*, pages 245–253, October 1998.

[4] G. Fegg, K. Moore, J. Dongarra, and A. Geist. Scalable Networked Information Processing Environment. `netlib2.cs.utk.edu/utk/people/JackDongarra/PAPERS/snipe.html`.

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

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

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

[8] M. Hayden. *The Ensemble System*. PhD thesis, Cornell University, January 1998. `www.cs.cornell.edu/Info/Projects/Horus/Papers.html`.

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

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

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

[12] M. C. Little. *Object Replication in a Distributed System*. PhD thesis, University of Newcastle upon Tyne, September 1991. `ftp://arjuna.ncl.ac.uk/pub/Arjuna/Docs/Theses/TR-376-9-91_EuropeA4.tar.Z`.

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

[14] L. E. Moser, P. M. Melliar-Smith, and P. Narasimhan. A Fault Tolerance Framework for CORBA. In *Proceedings of the IEEE International Symposium on Fault-Tolerant Computing*, pages 150–157, June 1999.

[15] OMG. *CORBA Object Request Broker Architecture-Version 2.2*. `www.omg.org/docs/orbos`.

[16] J. Ren, M. Cukier, and W. H. Sanders. An Adaptive Algorithm for Tolerating Value Faults and Crash Failures. *IEEE Transactions on Parallel and Distributed Systems, Special Issue on Dependable Network Computing, To appear.*

[17] P. Rubel. Passive Replication in the AQuA System. Master's thesis, University of Illinois at Urbana-Champaign, 2000. `www.crhc.uiuc.edu/PERFORM`.

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

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

[20] A. Vaysburd. *Building Reliable Interoperable Distributed Applications with Maestro Tools*. PhD thesis, Cornell University, May 1998. `www.cs.cornell.edu/Info/Projects/Horus/Papers.html`.

[21] P. Verissimo and A. Casimiro. The Timely Computing Base. Technical Report TR-99-2, Univ. of Lisboa, May 1999. `www.navigators.di.fc.ul.pt/docs/abstracts/tcbmodel.html`.

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