Performance Evaluation of a Probabilistic Replica Selection Algorithm *

Sudha Krishnamurthy
Coordinated Science Laboratory and
Department of Computer Science

William H. Sanders, Michel Cukier
Coordinated Science Laboratory and
Department of Electrical and Computer Engineering

University of Illinois at Urbana-Champaign, Illinois 61801

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

Abstract

When executing time-sensitive distributed applications, a middleware that provides dependability and timeliness is faced with the important problem of preventing timing failures both under normal conditions and when the quality of service is degraded due to replica failures and transient overload on the server. To address this problem, we have designed a probabilistic model-based replica selection algorithm that allows a middleware to choose a set of replicas to service a client based on their ability to meet a client's timeliness requirements. This selection is done based on the prediction made by a probabilistic model that uses the performance history of replicas as inputs. In this paper, we describe the experiments we have conducted to evaluate the ability of this dynamic selection algorithm to meet a client's timing requirements, and compare it with that of a static and round-robin selection scheme under different scenarios.

1. Introduction

Server replication is commonly used to improve the availability and response time of distributed services. Replicating the servers improves the availability of a service by allowing access to the service even when some of the servers are not functioning, and improves the response time of a service by allowing multiple clients to be serviced concurrently. When executing time-sensitive applications, a middleware that uses replicated servers to provide dependability and timeliness is faced with the important problem of preventing timing failures both under normal conditions and when the quality of service is degraded due to replica failures and transient overload on the server. In a previous paper [7] we described a dynamic replica selection algorithm we designed to address this problem in AQuA [2]. The algorithm we proposed 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.

Our work targets a system in which the machines hosting replicated services are distributed across a local area network (LAN). We mainly target applications such as search engines and radar-tracking applications, in which the requests made by a client do not modify the state of the server replicas. A client that requires a service to respond to its request within a specific time expresses its requirements as a quality of service (QoS) specification, which 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. Some of the factors that may cause a timing failure include transient overloads on a replica or on the network links, background load on the machines, and replica crashes. The uncertainty resulting from these factors makes it hard to provide deterministic temporal guarantees, and prompted us to use a probabilistic model instead.

The rest of this paper is organized as follows. Section 2 briefly describes some of the replica selection schemes that have been developed by other researchers. Section 3 describes the replica selection algorithms we have used in our experimental study, which includes the probabilistic selection algorithm we have developed, a static selection scheme that sends a request to a fixed set of replicas, and a round-robin selection scheme. In Section 4 we analyze the performance of the probabilistic replica selection algorithm. We compare the performance of the probabilistic scheme with the static scheme in Section 5 and with the round-robin scheme in Section 6. All of the experimental results are based on our implementation of these selection algorithms

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

in AQuA. We present our conclusions in Section 7.

2. Related Work

Several other replica selection algorithms have been formulated with the objective of choosing the replica that can deliver the lowest possible response time. Similar to our work, these algorithms often target stateless, distributed applications. Some of these algorithms choose the nearest replica based on a distance metric [5], and some choose the replica with the best historical average response time [8]. Some predict the time to propagate a message to different replicas using regression analysis of previously collected data [1], 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 [3]. 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 schemes that make use of temporal redundancy, however, may not be suitable for clients with specific time constraints. In contrast, the probabilistic selection scheme we have developed makes use of spatial redundancy to address the problem of meeting specific timing requirements of clients. It tunes the redundancy level to service a client's request based on the client's QoS requirements and the responsiveness of the replicas.

3. Replica Selection in AQuA

One of the main components of the AQuA middleware is the AQuA gateway, which is associated with each client and server object. The different replica selection schemes supported in AQuA are implemented as *selection handlers* within the gateway, as shown in Figure 1. Each selection handler has three main modules: a *replica selector*, an *information repository*, and a *timing failure detector*. An AQuA client can specify the selection scheme it wants to use in a configuration file. The corresponding selection handler is then loaded dynamically by the gateway during startup. The handler on the client side transparently intercepts a request from the client, and hands over the request to its selector module. The selector retrieves the replica list for the service from its information repository, and chooses the replicas to service the request based on the client's QoS requirements, using the selection strategy implemented in the handler. The handler then multicasts the request to the selected set of replicas through the Maestro-Ensemble group communication layer [9, 4]. We now describe the different replica selection schemes we have built into AQuA.

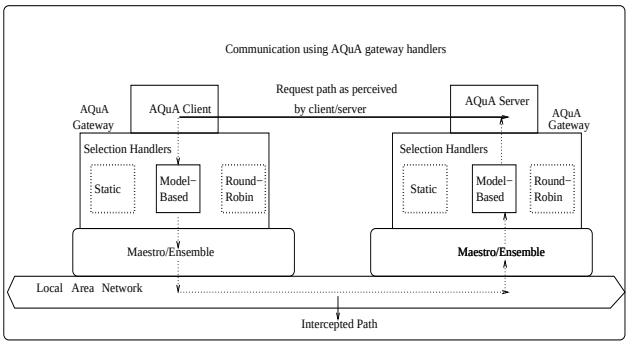


Figure 1. Replica selection in AQuA

3.1. Probabilistic Replica Selection Scheme

We first describe the probabilistic replica selection scheme we have developed. To identify the parameters of our probabilistic model, we conducted experiments to determine the factors that have a significant impact on the response time in AQuA. Based on our experimental analysis, we identified three factors: the *queuing delay*, W, which is the time that a request spends waiting in the request queue of a server; the *service time*, S, which is the time spent by a server in processing the request after dequeuing it from the request queue; and the *gateway-to-gateway delay*, which is the time it takes for an AQuA request or response to travel between the client and server gateways. The gateway delay is incurred on both the request and response paths, and the two delays together make up the two-way gateway-to-gateway delay, T. Equation 1 expresses the relationship between these three factors and the response time, R_i, which is the time it takes to receive a response from a replica i, after a request is transmitted to it.

$$R_i = S_i + W_i + T_i \quad (1)$$

We experimentally measure the values of the service time, S_i, and the queuing delay, W_i, as described later in Section 3.1.2, and record the values of the most recent l requests in a sliding window, L. The value of l is chosen so that it includes a reasonable number of recent requests but excludes obsolete measurements. We store these experimentally measured values on a per-replica basis in the information repository that is associated with each client's handler. For the gateway-to-gateway delay, T_i, rather than record its history over a period of time, we use its most recently measured value, because of the observation that the traffic in a LAN does not fluctuate as often as the other two parameters. For environments in which this observation is not true, it would be simple to extend our approach to record the history of the gateway-to-gateway delay as we do for the service time and queuing delay.

3.1.1. Probabilistic Model. Using the performance measurements collected above as inputs, the probabilistic se-

lection handler builds a model to predict the probability, $P_K(t)$, 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. M is the complete set of replicas offering the service requested by a client. The selection is then done based on this resultant probability. Let $P_c(t)$ be the minimum probability with which the client wants a response for its request by time t. Each replica in the selected subset independently processes a request and sends back its response. However, only the first response received for a request is delivered to the client. Therefore, a timing failure occurs only if no response was received from any of the replicas in the set K within t time units after the request was sent. It is straightforward to compute the distribution of the time to receive the earliest response if we assume that the response times of individual replicas are independent of one another. While this is not strictly the case in a shared network, in which the network delays may be correlated, we believe that it is a reasonable assumption to make, since the network delay across a LAN is usually a small fraction of the replica's response time, especially for compute-bound servers like the ones we are interested in. We use this independence assumption to compute the probability, $P_K(t)$, for the replicas in subset K, as follows:

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

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

$$P_K(t) = 1 - \prod_{i \in K} (1 - F_{R_i}(t)) \quad (2)$$

where $F_{R_i}(t)$ is the response time distribution function for replica i. Using the performance measurements we collect at runtime, we compute the probability mass function (pmf) of S_i and W_i for a replica 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 $F_{R_i}(t)$ for a replica i.

Algorithm 1 presents the selection algorithm used by the probabilistic selection handler when selecting the replicas to service a client. Since replicas may crash, our goal is to choose a set of replicas that can meet a client's time constraint with the probability the client has requested, even when one of the replicas in the selected set crashes while servicing the request. Our intuition is that if we can choose a set of replicas that can satisfy the timing constraint with the specified probability despite the failure of the member, m_0, that has the highest probability of meeting the client's deadline, then such a set should be able to handle the failure of any other member in the set. In [7] we have provided a formal justification for this intuition.

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 will 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}

The steps of the algorithm are as follows. The probabilistic selection handler first includes the replica, m_0, that has the highest probability of meeting a client's deadline, in the selected set, K. It then visits the remaining replicas in decreasing order of their response time distribution function, 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. $P_X(t)$ is computed using Equation 2. Thus, we effectively simulate the failure of m_0 by excluding it when building the set X. If the selection handler finds enough replicas to satisfy the above condition, then it extends this set to include m_0 and form the final selected set, K. If, however, it is unable to find enough replicas to meet this condition, it chooses all the available replicas to service the client. If a service is new and no performance history has been recorded for any of its replicas, the handler chooses all the available replicas offering the service. Although Algorithm 1 only addresses single replica crashes, it should be simple to extend it to handle multiple replica crashes.

Finally, since the replica selection is done when a request is invoked, the selection overhead, δ, itself may cause a delay in the response time. Hence, to account for this overhead in an actual implementation, we measure the algorithm overhead each time the selection algorithm is executed and use the most recently measured value of δ to compute the value of $F_{R_i}(t-\delta)$. We then modify Algorithm 1 to use the value of $F_{R_i}(t-\delta)$ wherever it uses the value of $F_{R_i}(t)$, so that it selects those replicas that can respond within $t - \delta$

time units rather than t time units, where t is the client's deadline as before.

3.1.2. Online Performance Monitoring. The client handlers that are interested in receiving performance updates from the servers initially multicast their subscription to the server replicas. The handler on the client side transparently intercepts a request from the client, and records the interception time, t_0. It then hands over the request to its selector module. Using the appropriate selection algorithm, the selector selects a set of replicas to service the request. The handler then multicasts the request to the selected set of replicas through the Maestro-Ensemble group communication layer.

Upon receiving the request from a client, the probabilistic selection handler at the server enqueues the request in the replica's request queue and records the time 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 before using CORBA's dynamic invocation interface to invoke the server application to service the request. The queuing delay, t_q, for a request is the difference between its dequeue time and enqueue time. When the server sends its response back to the client, the server handler intercepts the response and records the service duration, t_s. The handler then forwards the reply back to the client gateway along with the performance data, which includes the service duration, t_s, and the queuing delay, t_q. Each server replica also publishes the newly measured values of t_s and t_q to its subscribers, whenever it completes servicing a request. This information, pushed from the server replicas, is then used to update the client's gateway information repository.

When the client handler receives a reply from a replica, it records the time of reception, t_p, 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. For all the replies the handler receives for a request, it uses the extracted performance data to measure the new two-way gateway-to-gateway delay, t_d, between the client and the responding replica. This delay, t_d, is given by $t_d = t_p - t_m - t_q - t_s$, where t_q and t_s are obtained from the extracted data. t_m is the time at which the handler multicast the request to the selected set of replicas using Maestro/Ensemble. The handler then records this new value of the gateway delay in its local information repository. The timing failure detector in the handler checks whether a timing failure has occurred, by computing the response time, $t_r = t_p - t_0$. A timing failure occurs if $t_r > t$, where t is the response time requested by the client. The timing failure detector maintains a counter that keeps track of the number of times its client has failed to receive a timely response from a service. If the frequency of timely responses from the service is lower than the minimum probability of timely response the client has requested in its QoS specification, the handler notifies the client by issuing a callback. Note that when we collect the timing data as explained above, we do not require that the clocks be synchronized, because we always measure the two end-points of a timing interval on the same machine.

3.2. Static Selection Scheme

The static scheme allocates a fixed number of replicas to service a client. In our implementation, we chose to assign all the available replicas to service a client. Hence, we call this static scheme `static_all`. Two sources of overhead that are present in the probabilistic scheme are absent in the `static_all` scheme. These are the computational overhead incurred by the replica selection algorithm and the communication overhead incurred by the regular performance updates. Intuitively it would seem that we may be able to reduce the probability of timing failures for a client by sending its request to as many replicas as possible. On the other hand, by having all the replicas process a request, the `static_all` scheme may result in higher queuing delays for the requests, leading to higher response times. Thus, we were interested in finding out under what load conditions it would be beneficial to have all the available replicas process a request, and when it would be desirable to use a scheme, like the probabilistic scheme, that sends a request to no more than the number of replicas required to meet the client's timing requirements. The results presented in Section 5 provide this comparison.

3.3. Round-Robin Selection Scheme

In the round-robin scheme each client's request is sent to two different replicas, which are selected from the replica list in a round-robin fashion. We chose a redundancy level of two for the round-robin scheme so that the round-robin algorithm, like the probabilistic algorithm, can tolerate single crash faults. The round-robin scheme is dynamic because it chooses different replicas for each request. However, unlike the probabilistic algorithm, the round-robin scheme uses a fixed redundancy level, and does not use any knowledge of a replica's performance history during the replica selection. The results presented in Section 6 provide a comparison between the adaptive probabilistic selection scheme and the round-robin scheme.

4. Performance Evaluation

We now discuss the experiments we conducted to analyze the performance of the probabilistic replica selection algorithm, as implemented in AQuA. Our experimental setup is composed of a set of uniprocessor Linux machines

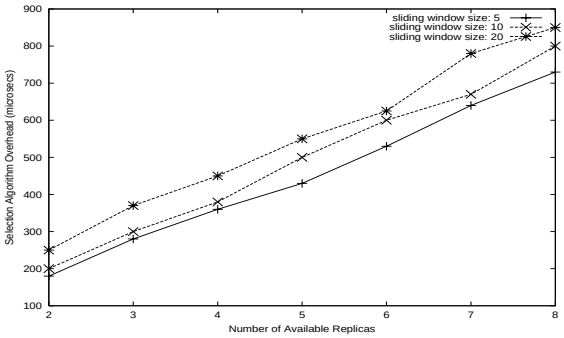
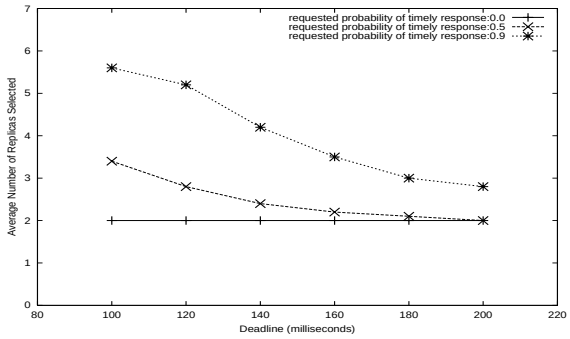
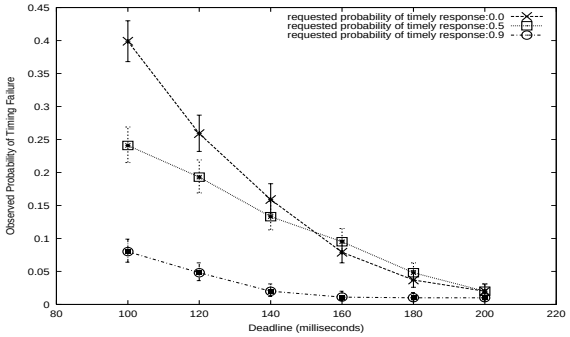


Figure 2. Overhead of probabilistic replica selection algorithm



(a) Number of replicas selected



(b) Validation of model

Figure 3. Performance of probabilistic scheme in a crash-free scenario

distributed over a 100 Mbps LAN. All confidence intervals for the results presented are at a 95% level, and have been computed under the assumption that the number of timing failures follows a binomial distribution [6].

4.1. Overhead of the Probabilistic Selection Algorithm

Figure 2 shows how the overhead of the probabilistic selection algorithm varies with the number of available replicas for three different sliding window sizes: 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. Computation of the response time distribution function contributes to 90% of these overheads, while selection of the replica subset using Algorithm 1 contributes to the remaining 10%. The larger the sliding window, the more the number of data points used to compute the response time distribution, resulting in higher selection overhead. For the following eperiments, we used a sliding window of size 5, unless otherwise stated.

4.2. Validation of the Probabilistic Model

We also conducted experiments to evaluate how effectively the subset of replicas chosen by the probabilistic 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 two different machines and independently issued requests to the same service with a 1000 millisecond delay between receiving a response and issuing the next request. We use the term *request delay* to define this duration that elapses before a client issues its next request to a service after it receives a reply for its previous request. The number of server replicas available for selection during each experiment was 7. Each server replica ran on a different machine and responded with an integer value. We simulated the background load on the servers by having each replica respond to a request after a delay that was normally distributed with a mean of 100 milliseconds and a variance of 50 milliseconds. In every run, each of the two clients issued 1000 requests to the service. One of the clients requested a deadline of 200 milliseconds in each run and specified that this deadline be met with a probability ≥ 0. The second client requested a different deadline in each run. For each of these deadline values of the second client, we computed the probability of timing failures in a run of 1000 requests by measuring the number of responses in the run that had failed to arrive by the deadline specified by the second client. In order to study the behavior of the probabilistic selection algorithm for different values of the probability of timely responses specified by a client, we repeated these experiments for three different probability values specified by the second client in its QoS specification: 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.

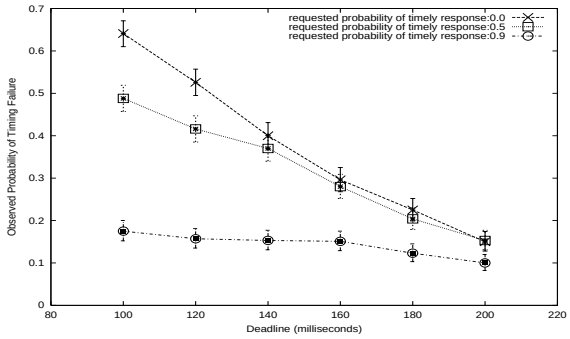
Figure 3a shows the average number of replicas selected

by the dynamic selection algorithm to service the second client for each of its QoS specifications. From this figure we observe that the redundancy level chosen by the algorithm to service a request reduces as the client's QoS specification becomes less stringent. The reason for this is that our algorithm 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. Hence, the algorithm can satisfy the QoS requirement with fewer replicas.

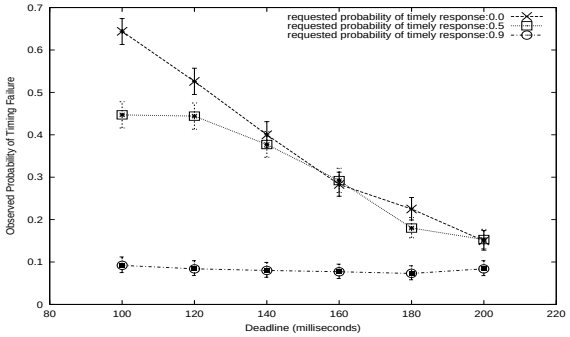
Figure 3b shows how successful the selected replicas, shown in Figure 3a, were in meeting the QoS specifications of the second client. Figure 3b shows that when the client specifies that the probability of timely responses must be at least 0.9, the maximum probability of timing failures we observe experimentally 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 failure probabilities up to 0.5 and 1, we observe maximum timing failure probabilities of 0.32 and 0.36, respectively, for the deadline values we used. These results show that, in each case, the set of replicas selected by Algorithm 1 was able to meet the client's QoS requirements successfully by maintaining 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 predict accurately the set of replicas that would be able to meet the client's deadline with at least the probability requested by the client.

4.3. Performance During a Replica Crash

In Section 3.1 we claimed that if we choose a set of replicas in a manner such that they can satisfy the timing constraint of a client with the specified probability even if the member that has the highest probability of meeting the client's deadline crashes, then such a set should be able to tolerate any single replica crash that occurs when the request is being serviced. In order to verify the validity of this claim experimentally, we used the same experimental setup with two clients that we used for the experiments in Figure 3 for the crash-free scenario. We simulated the crash of a replica by having the timing fault handler forward a client's request to all the selected replicas except to the one with the highest probability of meeting the client's deadline. Since the client never receives a response from the omitted member, this effectively simulates the scenario in which the member has either crashed before completing its service, or is unresponsive. Figure 4a shows the results of our experiments using a sliding window of size 5. From Figure 4a, we see that despite the failure of a replica, the probabilistic algorithm is able to meet the client's QoS specification when the client specifies a value of 0.5 or lower for the probability of timely response. However, when the client requests a higher value, such as 0.9, for the probability of timely response, the timing failure probability measured experimentally is about 0.2, which is higher than the 0.1 failure probability allowed by the client's specification. Since at the time of selection, Algorithm 1 is able to find a replica subset that has the ability to meet a client's timing requirements with a probability ≥ 0.9, even if one of the selected replicas crashes, one possible explanation for this behavior is that the performance measurements that our model uses as inputs to select the replica subset during a crash may not have the accuracy needed to ensure low timing failure probability, specifically values ≤ 0.1.



(a) Sliding window size = 5



(b) Sliding window size = 20

Figure 4. Performance of probabilistic scheme during a replica crash

To test this hypothesis, we repeated the experiments we used to obtain the results in Figure 4a by using a larger sliding window. Figure 4b shows the results we obtained using a sliding window of size 20. From Figure 4b we see that the observed probability of timing failures is now within 0.1 for the case in which the client requests that its deadline be met with a probability ≥ 0.9. Thus, by using more experimental measurements to improve the accuracy of the estimated response time, the model was able to improve its ability to

predict the set of replicas that can meet a client's QoS specification with more stringent requirements. So why was the model able to predict successfully the set of replicas that can meet a client's deadline with a probability ≥ 0.9 in the crash-free scenario in Figure 3, despite using a sliding window of size 5? We think the reason is that the inaccuracy in the experimental measurements in the crash-free case was largely offset by the fact that there is one additional replica processing the request. More importantly, this additional replica was also the one that had the highest probability of meeting the client's deadline.

In summary, we have shown that our model can successfully predict the ability of a set of replicas to meet a client's QoS specification. However, the accuracy of the prediction is dependent on the accuracy of the performance measurements the model uses as its inputs. One way of improving the accuracy of the prediction is to use a sliding window that includes more sample points for computing the response time distribution of the replicas. Using a large sliding window, however, has the disadvantage that it may allow obsolete performance information to be included when a service is not accessed frequently. Therefore, choosing a small window size may be more appropriate in environments in which a replica's responsiveness changes rapidly with time, as a smaller window allows quicker adaptivity to changes in a replica's responsiveness. Thus, the size of the sliding window has to be chosen based on the specific environment and access patterns of a service, through an assessment of the tradeoffs between the desired levels of accuracy and adaptivity.

5. Probabilistic vs. Static Selection

We now describe the experiments we conducted to compare the performance of the probabilistic replica selection algorithm with that of the `static_all` selection scheme, under different load conditions. When conducting experiments using the probabilistic scheme, all the clients involved in the experiment chose the probabilistic selection handler. Similarly, for the `static_all` scheme, all the clients involved in the experiment used the static selection handler. We carried out our experiments by varying the client-induced load in two ways.

In the first case, we varied the client-induced load by varying the request delay. The induced load on the servers is higher for smaller request delays. We used the same experimental setup with two clients, as described in Section 4.2. Figure 6 compares the performance of the probabilistic scheme with that of the static scheme for different values of the request delay. The graphs in the left column of Figure 6 compare the observed timing failure probability for different values of the request delay, while the graphs on the right provide the comparison for the average number of replicas selected for the corresponding cases.

In the second case, we varied the client-induced load by varying the number of clients accessing a service concurrently. The induced load on the servers increases with the number of clients. All the clients used the same request delay value of 250 milliseconds. One of the clients specified a different deadline in each run and requested that its deadline be met with a probability ≥ 0.5. All the remaining clients specified a deadline of 200 milliseconds in each run and requested that this deadline be met with a probability ≥ 0.0. The plots in Figure 7 compare the performance of the probabilistic scheme with that of the static scheme using 2, 4, and 8 clients. 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.

We observe from the graphs in Figures 6 and 7 that the `static_all` scheme results in fewer timing failures than the probabilistic scheme when the client-induced load is low. For example, when the request delay is considerably larger than the mean service time (which in our case was 100 milliseconds), the static scheme outperforms the probabilistic scheme. However, as the client-induced load increases, the performance of the static scheme degrades considerably compared to the probabilistic scheme. This is shown in Figure 6 for the case in which the request delay is 100 milliseconds, and in Figure 7 for the case in which there are 4 or more clients. This is because the `static_all` scheme uses all the available replicas to service the requests of both the clients, while the probabilistic scheme selects replicas as shown by the graphs in the right-hand column. The above results lead us to conclude that a dynamic selection scheme, such as the probabilistic scheme we have proposed, scales better with load because it adaptively changes the replica assignment to meet a client's timing requirements by monitoring the changes in the responsiveness of the replicas. A second important conclusion we arrive at is that we do not gain much in performance by assigning more than the required number of replicas to service a request.

From Figures 6 and 7 we observe that as the queuing delay increases, there are cases in which the probabilistic scheme is unable to find enough replicas to meet the deadline with the probability requested by the client. For instance, in Figure 6c, for the case in which the request delay is 100 milliseconds, the probabilistic scheme is unable to find a replica subset to meet deadline values ≤ 120 milliseconds with a probability ≥ 0.9. In this case, the probabilistic scheme sends the request to all 7 available replicas, as shown in Figure 6d. There are some alternative measures that the selection handler can take under such conditions. For example, it could inform the client that there are insufficient resources to satisfy its QoS requirement, so that the

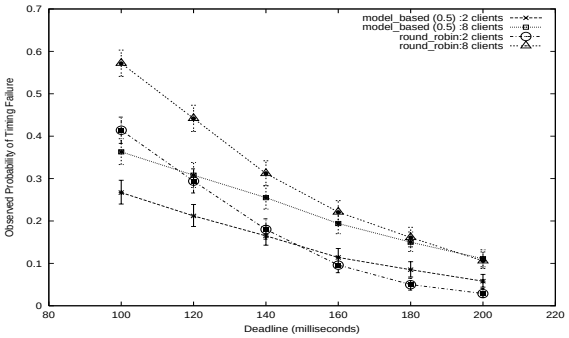


Figure 5. Probabilistic scheme vs. Round-Robin scheme

client can choose to either renegotiate its QoS specification or send its requests at a later time when the system is less loaded.

6. Probabilistic vs. Round-Robin Selection

In order to compare the performance of the probabilistic selection scheme and the round-robin scheme under different load conditions, we repeated the same experiments as above by varying the number of clients accessing a service, and used the round-robin selection scheme to assign replicas to service all the clients. Figure 5 shows the probability of timing failures observed by the client that requested a different deadline in each run and specified that this deadline be met with a probability ≥ 0.5.

When the client-induced load increases, the queuing delays in the case of the round-robin scheme are not as high as those for the `static_all` or the probabilistic schemes. This is because the round-robin scheme assigns pairs of replicas to service a client in a round-robin manner and is therefore able to balance the client-induced load across the available replicas better than the other two schemes. However, since the round-robin scheme chooses the replicas without assessing their ability to meet a client's QoS specification, it is hard to say conclusively whether the replicas chosen by the round-robin scheme will be able to meet a client's requested probability of timely response, even when sufficient replicas are available. For example, if the client had requested that its probability of timing failure be ≤ 0.1, then from Figure 5 we observe that for the case in which there were 8 clients accessing the service, the replica assignment made by the round-robin scheme would have failed to meet the client's QoS specification.

In summary, a round-robin scheme is good when the primary goal is load balancing, rather than meeting specific timing requirements of a client. It is also good when all of the replicas have nearly identical performance characteristics and exhibit little or no variability in their performance. However, in a system in which the replicas have different and variable performance characteristics, if the primary goal is to meet specific timing requirements of a client, a more adaptive replica selection scheme, like the probabilistic scheme, is more effective.

7. Conclusions

The experimental results we have presented show that a scheme like the `static_all` scheme or the round-robin scheme would be sufficient when all the clients have either the same or no specific timing constraints. They also show that an adaptive scheme, like the probabilistic replica selection scheme we have developed, would be useful in an environment in which time-sensitive clients with different QoS specifications access compute-bound service providers that display significant variability in their response times. Although we currently require the client to input the probability value, a possible extension to our model would be to allow a client to specify a priority or cost value as input, in addition to its deadline. The middleware can then map the priority or cost to a probability value, and use the probabilistic selection algorithm to select replicas to meet these higher-level requirements.

References

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

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

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

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

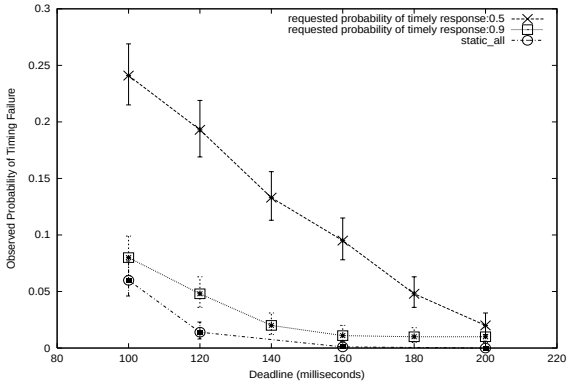
[5] J. Heidemann and V. Visweswaraiah. Automatic Selection of Nearby Web Servers. Technical report, University of Southern California, USC TR 98-688, 1998.

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

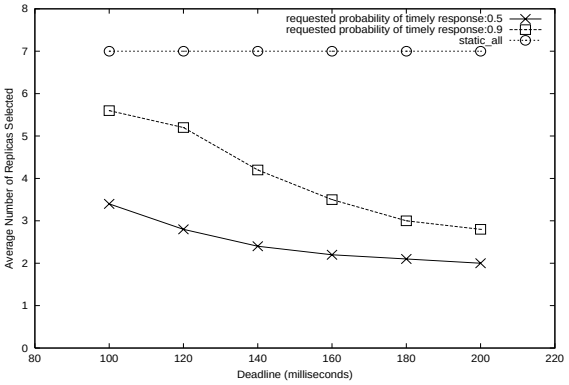
[7] S. Krishnamurthy, W. Sanders, and M. Cukier. A Dynamic Replica Selection Algorithm for Tolerating Timing Faults. In *Proc. of the International Conference on Dependable Systems and Networks*, pages 107–116, July 2001.

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

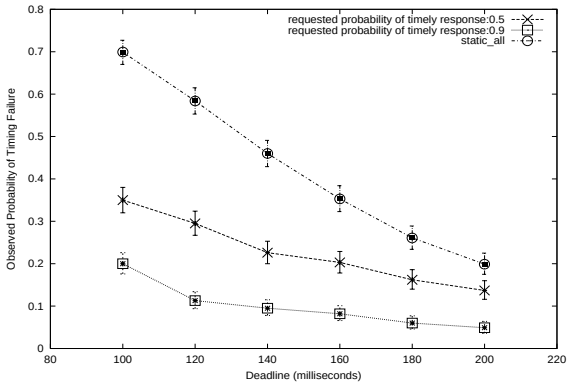
[9] A. Vaysburd. *Building Reliable Interoperable Distributed Applications with Maestro Tools*. PhD thesis, Cornell University, May 1998. `www.cs.cornell.edu/Info/Projects/Horus/Papers.html`.



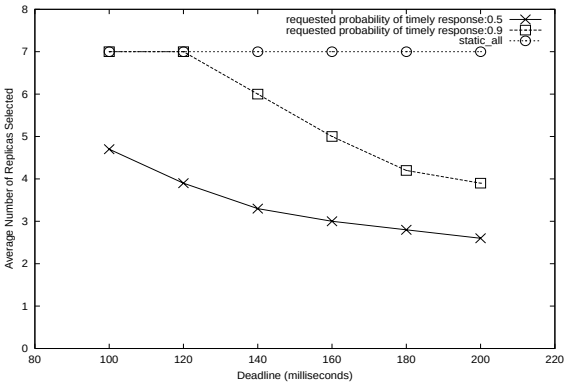
(a) Request delay: 1000 ms



(b) Request delay: 1000 ms

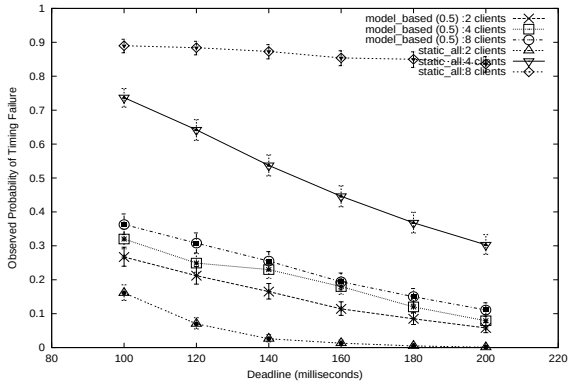


(c) Request delay: 100 ms

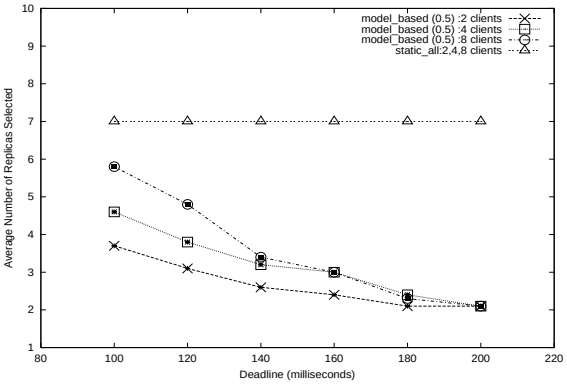


(d) Request delay: 100 ms

Figure 6. Probabilistic scheme vs. Static scheme: variable request delay



(a) Probability of timing failure



(b) Number of replicas selected

Figure 7. Probabilistic scheme vs. Static scheme: variable number of clients