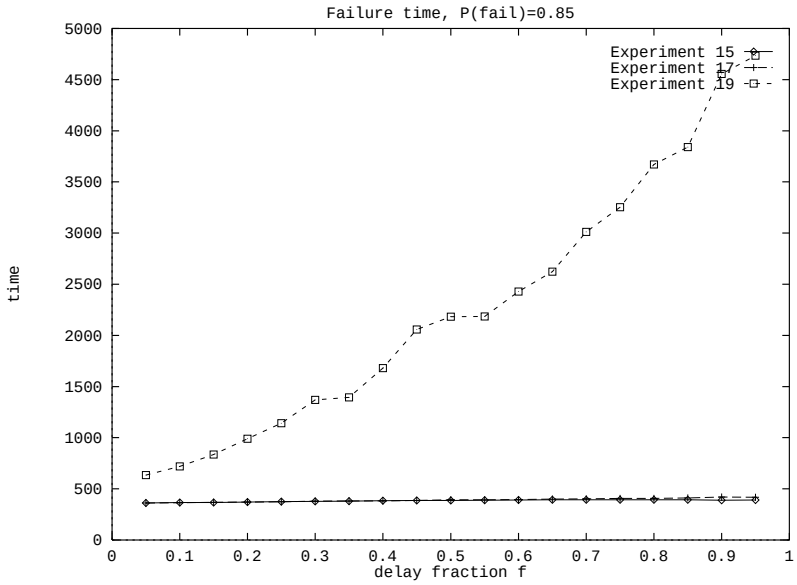


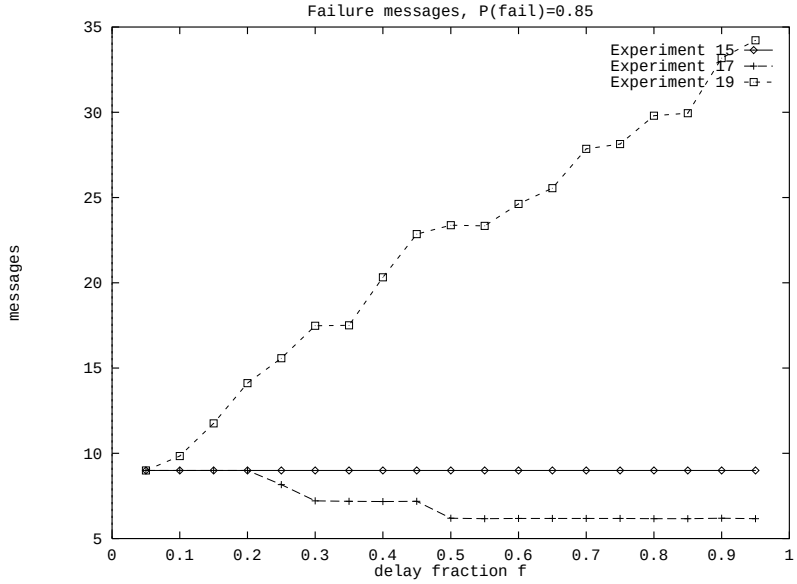
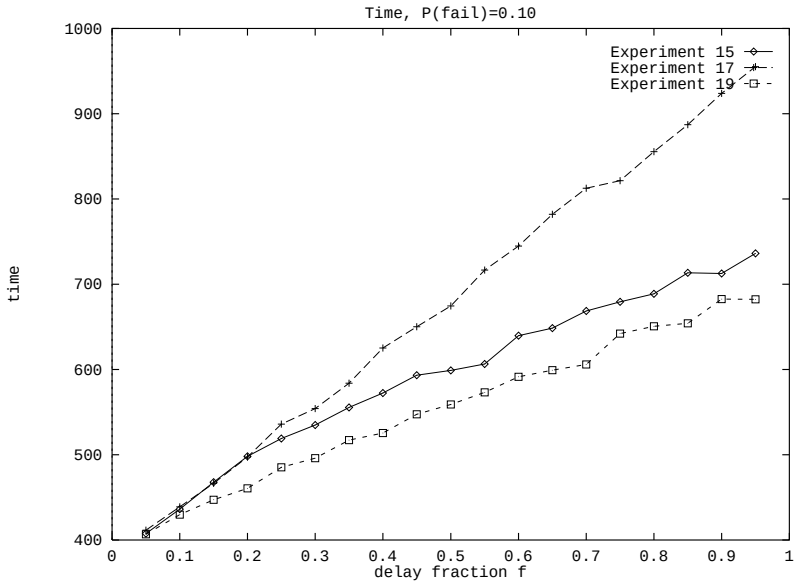


Figure 9: Failure latency and messages for $p_f(r) =$ exponential with base 0.85



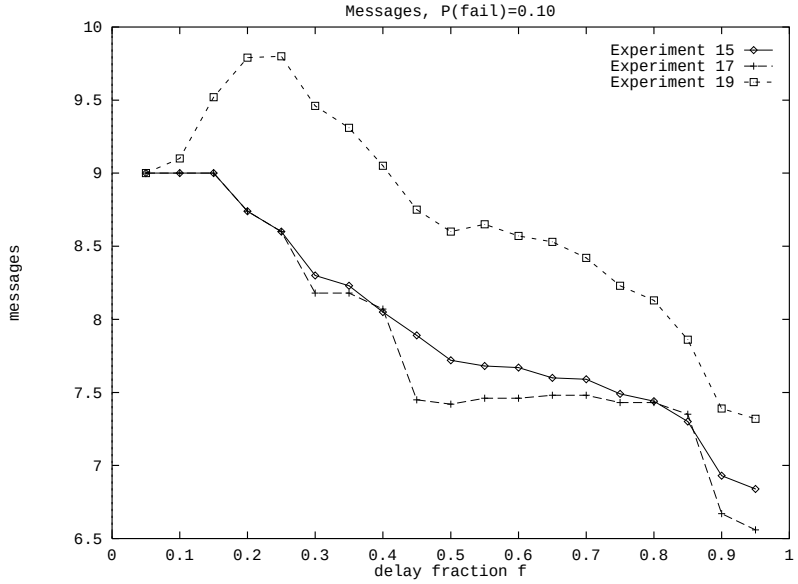
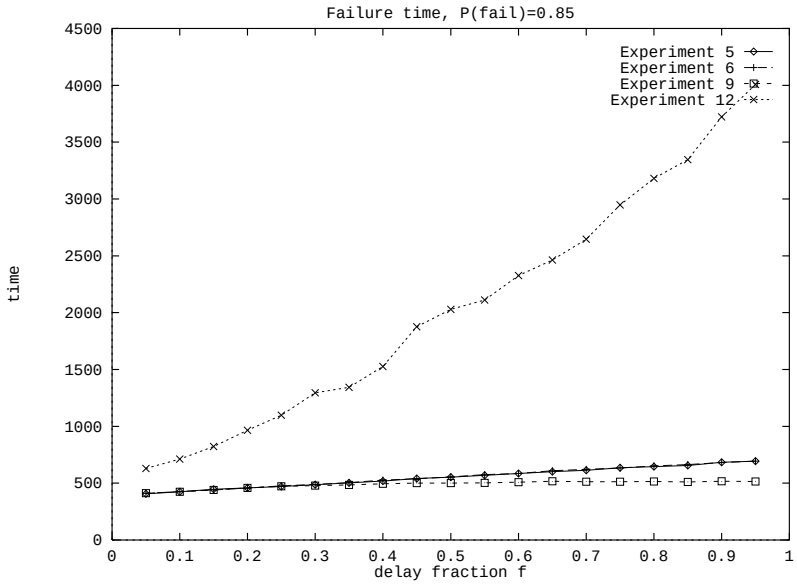


Figure 8: Success latency and messages for $p_f(r) =$ exponential with base 0.1



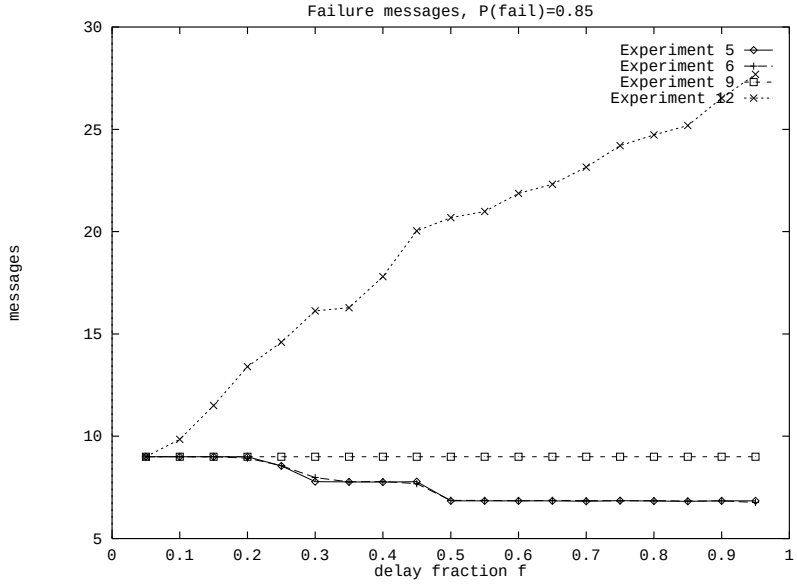
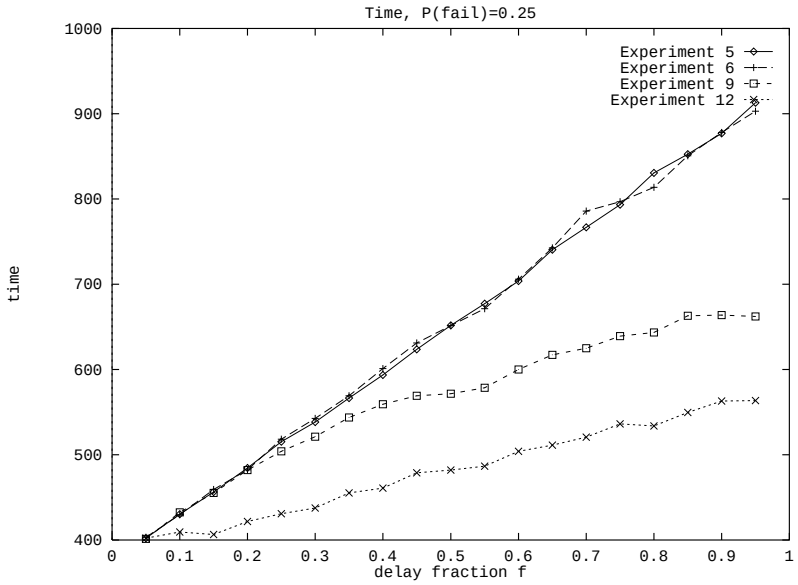


Figure 7: Failure latency and messages for fail(r) = normal distribution, $p_f = 0.85$



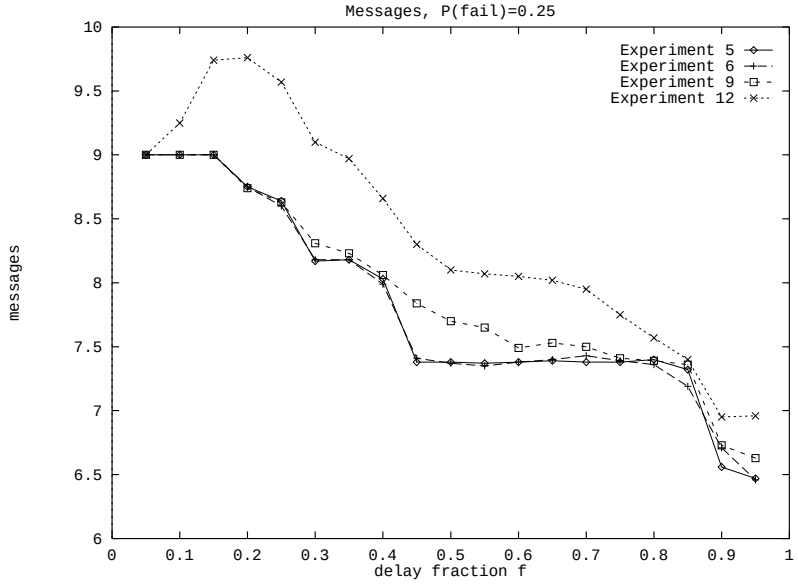


Figure 6: Success latency and messages for fail(r) = normal distribution, $p_f = 0.25$

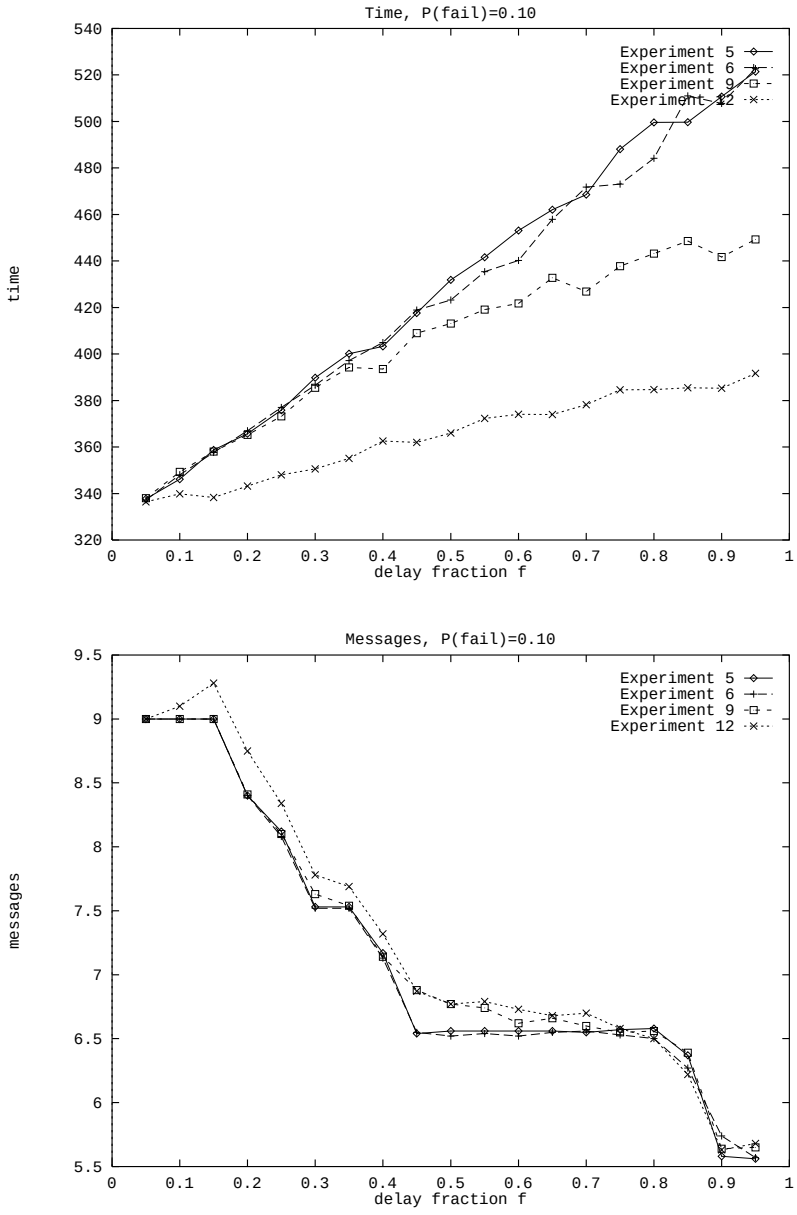


Figure 5: Success latency and messages for fail(r) = normal distribution, $p_f = 0.1$

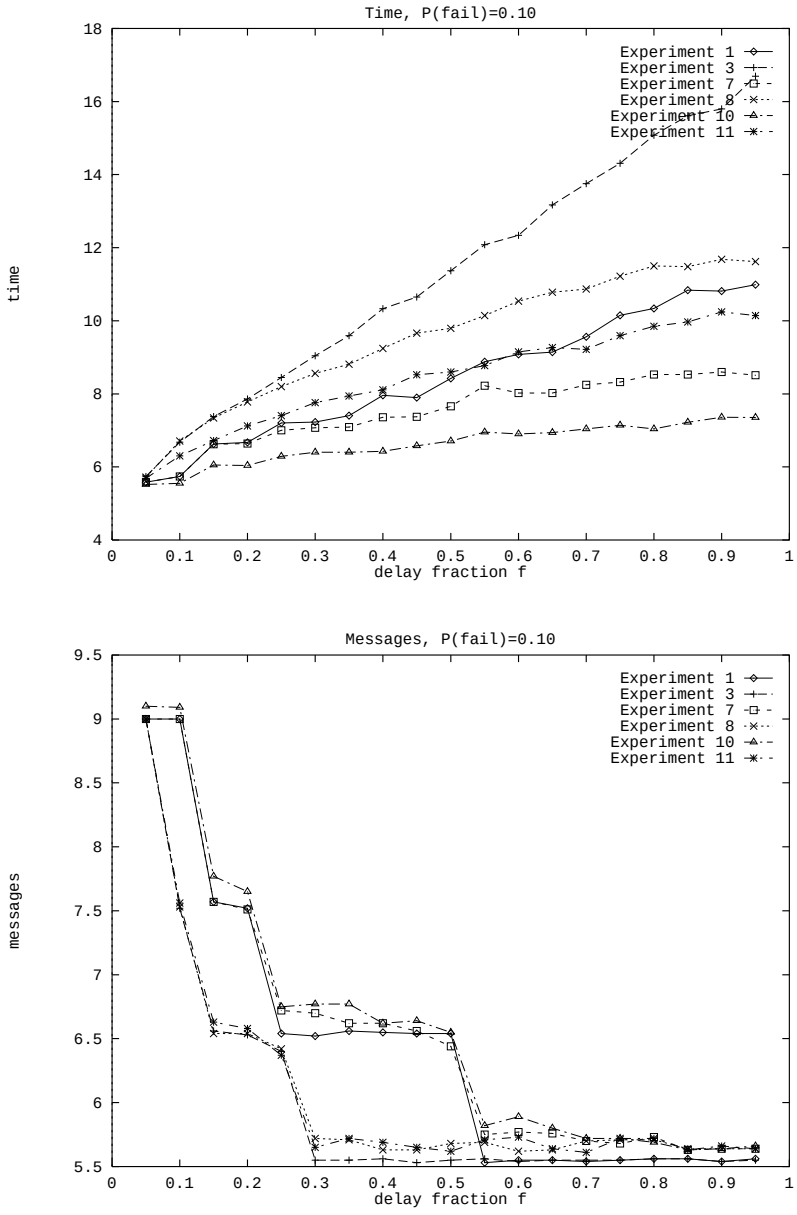


Figure 4: Success latency and messages for fail$(r) = cr$, $p_f = 0.1$

References

[Bernstein84] P. A. Bernstein and N. Goodman. An algorithm for concurrency control and recovery in replicated distributed databases. *ACM Transactions on Database Systems*, **9**(4):596–615, December 1984.

[Birrell84] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, **2**(1):39–59, February 1984.

[Davcev85] D. Davčev and W. A. Burkhard. Consistency and recovery control for replicated files. *Proceedings of 10th ACM Symposium on Operating Systems Principles* (Orcas Island, Washington). Published as *Operating Systems Review*, **19**(5):87–96, December 1985.

[Fishman78] G. S. Fishman. *Principles of Discrete Event Simulation.* Wiley and Sons, 1978.

[Jajodia87] S. Jajodia and D. Mutchler. Dynamic voting. *Proceedings of ACM SIGMOD 1987 Annual Conference*, pages 227–38. Association for Computing Machinery, May 1987.

derives an estimate of communication latency from the topology of the internetwork. The second mechanism uses observed access latencies from one access to predict the latencies for the next access. The first method establishes an initial estimate of the ordering, while the second refines the estimate, and adapts to changes in the network.

7 Conclusions

We have presented three algorithms for accessing replicated data in an internetwork, and analyzed their performance. Each of these algorithms is parameterized on the delay fraction f, which can be used to tune the algorithms to best match an application and an internetwork environment. All three algorithms order replicas by their expected latency of response. The `simple` algorithm initially queries enough nearby replicas to form a quorum if there are no failures, and additional queries are sent as the algorithm times out before obtaining a quorum. The `resched` algorithm improves the `simple` algorithm by sending additional queries either when the algorithm detects that a replica is unavailable or when a time-out is reached. The `retry` algorithm also queries a number of nearby replicas initially, but if one of those replicas is unavailable the `retry` algorithm will continue to resend queries to that replica while sending additional queries to more distant replicas.

In our performance simulations we have found that the the number of messages sent and the time spent in an access by these algorithms are inversely related. In particular, when the value of f is near one, the algorithms will favor messages over time, while values of f near zero will cause the algorithms to send more messages but require less time to complete an access. The parameter f can be used to tune the algorithm for different internetwork environments and different applications. We assume that both messages and time have a cost. If messages are considered to be expensive, perhaps due to the scale of the internetwork, large values of f can minimize the overall cost. If time is more important, smaller values of f are to be preferred. Further, the number of messages an algorithm can be expected to send is not a smooth function, but exhibits plateaus and sharp changes in value, particularly when failure is unlikely. This attribute of the message curve implies that only a few values of f need be considered when attempting to find a value which minimizes a cost function.

The three algorithms we presented exhibit different behaviors under different probabilities of a replica being unavailable. The `retry` algorithm is to be preferred when failure is unlikely, as it will require both fewer messages and approximately the same time. However, as the probability of failure increases the `resched` and `simple` algorithms perform better than `retry`, both by requiring fewer messages to successfully form a quorum and by requiring less time and fewer messages to detect when a quorum cannot be formed.

8 Acknowledgments

We are grateful to John Wilkes and the members of the Concurrent Systems Project at Hewlett-Packard Laboratories for their comments on this paper.

In figure 7 we observe the number of messages sent and amount of time required before each algorithm determines a quorum cannot be obtained. These curves were computed at $p_0 = 0.85$, but the observations we report here are valid for other failure probabilities as well. The most noticeable feature of these data is that the `retry` algorithm requires both more messages and more time to detect failure. This algorithm requires so much time because it continues trying to access replicas until either the nth replica replies, or until that replica is found to have failed. All during this time the algorithm will continually retry closer, failed replicas in case those replicas have become available. The `resched` and `simple` algorithms require less time, because these algorithms declare an access to have failed when fewer than q replicas are accessible, and no replicas are ever re-queried. In all our trials the `resched` algorithm sent an access to every replica before giving up, though the algorithm returned failure *before* the most distant replicas could reply. The `simple` algorithm sent the fewest messages of our three algorithms, sending only 60% as many messages as `resched` when $f \approx 1$. The times for both `resched` and `simple` were nearly constant, with `simple` requiring slightly more time as f increased. Once again experiments 5 and 6 showed that the variability of the access and fail times did not significantly affect results.

Finally, figures 8 and 9 show the results of similar experiments, where the probability of a replica being unavailable increased according to the relative distance of the replica. For these experiments we assumed that the probability of a replica being available was $(1-p_0)^r$ for replica r, with $p_0 = 0.1$. The data resulting from these experiments are nearly identical to those in figure 6, and we conclude that a non-uniform probability of failure has little affect on our conclusions.

6 Future Work

There are several assumptions and limitations in the model we have used for this simulation, and we intend to eliminate these deficiencies in further studies. The most significant limitation is the set of access times for each replica. We have assumed that the expected access time for replica r is the linear function $cr + b$ time units. We would like to model the effects of a non-linear distribution of access times, including placing several replicas on a local network (and making them accessible by broadcast) and an exponential distribution of access times. We have also assumed negligible time is spent in computation when processing an access. In reality an access may require data to be read from disk, which can take a significant amount of time. We intend to perform additional simulations, in which we will use measurements taken on the Internet for the access and failure latencies for each replica. Our simulations have assumed that the load on the network due to one query is negligible. While we have conducted experiments which suggest this is an accurate assumption, we intend to validate this assumption more accurately both by using more detailed simulation and by measurement of the Internet.

In addition to using more accurate distributions for latencies, we intend to improve our model of failure. The topology of the Internet provides many redundant paths along the backbone networks, while "leaf" sites are often connected by one gateway. When a gateway crashes it may make a large portion of the internetwork unavailable. We intend to improve our simulation by including all the components of an internetwork. We have conducted some preliminary studies of failure modes on the Internet; we intend to supplement these studies to better inform our simulation.

Our algorithms assume a known ordering on the expected access times for the replicas. We intend to study two different mechanisms for deriving an ordering on replicas. The first method

probability of failure as the systems hosting replicas, then we would expect to see an exponential increase in the probability that the connection to a replica is down as the number of intermediate gateways increases.

5 Results

Figure 4 shows the time spent and messages sent in processing a successful access, when the access(r) and fail(r) functions are single-valued, rather than distributions. These graphs were obtained with $p_0 = 0.1$. We observe a number of phenomena related to success latency. First, the `retry` algorithm succeeds faster than the `resched` algorithm, which is in turn faster than the `simple` algorithm. The `retry` algorithm uses only very slightly more messages than the other two algorithms. From these data we also observe that the number of messages required to complete an access drops, as expected, as the delay fraction f is varied from 0 to 1, while the time to completion increases linearly. We observe that the slope of the time line increases as the ratio of fail(r) to access(r) increases.

Figure 5 shows the time spent and messages sent in a successful access when the access and failure times are normally distributed, measured when $p_0 = 0.1$. Once again the `retry` algorithm requires substantially less time to complete than do the other algorithms, while requiring only slightly more messages. We observe that the number of messages decreases as f increases, and that the time to completion increases. We also observe that the results appear to be reasonably insensitive to the variability of the access and failure times, as the curves for experiments 5 and 6 are similar.

In figures 4 and 5, the number of messages decreases in a roughly stair-step fashion, and if there are no failures the number reaches a minimum value at $f \approx f_{\min}$, where

$$\lceil f_{\min} \cdot \text{fail}(q) \rceil > \frac{\text{fail}(q)}{\text{access}(q)}.$$

Thus as the ratio between the time required to detect failure and the expected time for a successful reply for the qth closest replica increases, the value of $f_{\min}$ decreases. As the number of failures in the system increases, $f_{\min}$ appears to increase as well. In our experiments, when $p_0 = 0$, the resulting values of $f_{\min}$ closely matched this formula. The messages curves for experiments 1, 7, and 10, when compared to the curves for experiments 3, 8, and 11 bear out this relationship. The predicted value of $f_{\min}$ for experiments 5, 6, 9, and 12 is approximately 0.86, which closely matches our data. Most notably, we find that all three algorithms reach plateaus at approximately the same values of f, excepting the increase in number of messages for the `retry` algorithm at low values of f.

Figure 6 reports the number of messages and amount of time required when $p_0 = 0.25$, substantially increasing the probability that nearby replicas will be unavailable. The `retry` algorithm still requires much less time to successfully form a quorum, but uses yet more messages to do so. When $p_0 = 0.1$, `retry` used approximately 2% more messages than `resched` or `simple`, but when $p_0 = 0.25$, `retry` used 9% more messages. As the probability of failure increases we find that the number of messages sent by the `retry` algorithm continues to grow. However, we also observe that `retry` obtains a quorum somewhat more often than the other algorithms, succeeding 99% of the time at $p_0 = 0.25$ while the `resched` and `simple` algorithms succeed in 95% of the trials.

Table 2: Experiment latencies

Experiment	Algorithm	access(r)	fail(r)	$p_f(r)$
1	simple	r	$2r$	uniform p_0
2	simple	r	$3r$	uniform p_0
3	simple	r	$4r$	uniform p_0
4	simple	uniform distribution on $[r,2r)$	$2r$	uniform p_0
5	simple	normal distribution $\mu = 50 + 50r$, $\sigma = 10/3.3$	normal distribution $\mu = 100 + 50r$, $\sigma = 10/3.3$	uniform p_0
6	simple	normal distribution $\mu = 50 + 50r$, $\sigma = 10$	normal distribution $\mu = 100 + 50r$,$\sigma = 10$	uniform p_0
7	resched	r	$2r$	uniform p_0
8	resched	r	$4r$	uniform p_0
9	resched	normal distribution $\mu = 50 + 50r$, $\sigma = 10/3.3$	normal distribution $\mu = 100 + 50r$, $\sigma = 10/3.3$	uniform p_0
10	retry	r	$2r$	uniform p_0
11	retry	r	$4r$	uniform p_0
12	retry	normal distribution $\mu = 50 + 50r$, $\sigma = 10/3.3$	normal distribution $\mu = 100 + 50r$, $\sigma = 10/3.3$	uniform p_0
13	resched	r	$2r$	$1 - (1 - p_0)^r$
14	resched	r	$4r$	$1 - (1 - p_0)^r$
15	resched	normal distribution $\mu = 50 + 50r$, $\sigma = 10/3.3$	normal distribution $\mu = 100 + 50r$, $\sigma = 10/3.3$	$1 - (1 - p_0)^r$
16	simple	r	$2r$	$1 - (1 - p_0)^r$
17	simple	normal distribution $\mu = 50 + 50r$, $\sigma = 10/3.3$	normal distribution $\mu = 100 + 50r$, $\sigma = 10/3.3$	$1 - (1 - p_0)^r$
18	retry	r	$2r$	$1 - (1 - p_0)^r$
19	retry	normal distribution $\mu = 50 + 50r$, $\sigma = 10/3.3$	normal distribution $\mu = 100 + 50r$, $\sigma = 10/3.3$	$1 - (1 - p_0)^r$

in experiment 5 was set so that 99% of all events would occur within 10 time units of the mean value μ. For experiment 6 the standard deviation was set to a larger value, so that 99% of all events would occur within 33 time units of the mean value. Taken together the two experiments show the effect of variability in latencies on our results.

Experiments 7, 8, and 9 were used to obtain performance measures for the `resched` algorithm. The results of these experiments can be compared to those of experiments 1, 2, and 5 respectively to determine the performance of the `resched` algorithm as compared to the `simple` algorithm. Experiments 10, 11, and 12 likewise were used to obtain performance data for the `retry` algorithm.

Experiments 13–19 are used to determine the effect of non-uniform probabilities of failure. In these experiments we assume that the probability that a replica is available decreases exponentially as the replica number index increases. This approximates the failure behavior of gateways and intermediary networks in an internetwork. If every gateway in the internetwork has the same

messages in the `simple` and `resched` algorithms. The `retry` algorithm used one timer to trigger additional messages, and one timer per replica to retry the replica after a message to that replica had failed. Each simulation was run 3000 times for each experiment. For each experiment, the simulation program collected the number of messages sent and the time spent before the access algorithm was able to obtain a quorum or declared failure. The program also derived 95% confidence intervals on these data.

In table 2 we summarize the experiments we conducted using these simulations. Each experiment gathered performance measures on one algorithm, reported in the "algorithm" column. Each algorithm was tested against several distributions of access(r) and fail(r), and against different distributions of $p_f(r)$. The distribution used in each experiment is also listed in table 2. In each experiment we tested an algorithm with the probability $p_f(r)$ of a replica failing at several different values. In some experiments the probability of a replica being unavailable was a function of the base failure probability p_0. For example, experiments 13–19 measure the effect of increasing the likelihood of failure as the "distance" of a replica increases.

All experiments shared certain parameters. In all experiments we assumed 9 replicas, with 5 replicas required to form a quorum. We selected 9 replicas to ensure that we would be able to test each algorithm with a relatively large number of replicas, and we selected 5 replicas as our quorum size as the smallest majority of 9.

There were four questions we wanted to examine in our experiments. The most important question was the relative performance of each of the three access algorithms. We also wanted to measure the sensitivity of each algorithm to its networking environment. We were interested in how each algorithm would perform as the variance of the access and fail latencies were varied, how a uniform versus an exponential probability of failure affected each algorithm, and the effect of varying the time required to detect failure.

In experiments 1, 2, and 3 we assume that the variances of the access and failure latencies for an access of a replica are zero; that is, that the values are exact, rather than a random distribution. For experiment 1 we assume that the failure latency is twice the access latency; for experiment 2, three times the access latency; and for experiment 3, four times the access latency. Using a single latency value produced very clear result graphs, especially at very low and very high failure probabilities. Taken together, these three experiments show the effect of different failure latencies on our performance figures. Experiments 1, 2, 7, 8, 10, and 11 show the relative performance of our three algorithms under the assumption of no variability in access and file times.

For experiment 4 we modified experiment 1, to assume that responses from replicas arrive according to a uniform distribution. We assumed that responses from replica r arrived no sooner than r time units after they were sent, and that responses would never take longer than the failure time-out of $2r$ time units, with a mean of $1.5r$ time units. This experiment has the highest variability of all the experiments. It also has a smaller ratio of failure latency to access latency than any of experiments 1, 2, or 3.

Experiments 5 and 6 were used to determine the effect of a normal distribution of both the access and the failure latencies on the `simple` algorithm. In these experiments replicas were ordered by the expected value of their access and failure latency distributions. We have measured communication times on the Internet, and found that actual times to send a message are approximately normally distributed. The normal distribution in these experiments exhibits less variability than does the uniform distribution used in experiment 4. The standard deviation σ of the access and fail latencies

```
// Retry -- send additional messages at a fraction of the longest
//          failure time for any outstanding message.  If a message
//          fails, periodically retry that replica.
retry(int q, site_list R, float f)
{
    int n = |R|;     // number of replicas
    int delay[|R|];  // time to wait for retry of replica i
    int succ = 0;    // number of successful replies
    int fail = 0;    // number of failed replies
    int next = 0;    // next replica to access

    for i = 1 to q {        // send off q queries
        access R(i);
        delay[i]=0;
    }
    schedule time-out(q) in (f*fail(q)) units;
    next = q+1;

    for each event {
        if event is reply(i) {
            succ = succ+1;
            if succ >= q
                return SUCCESS;
            else if i == n
                return FAILURE;
        } else if event is failed(i) {
            if i == n
                return FAILURE;
            } else {
                schedule retry(i) in delay[i] units;
            }
        } else if (event is time-out(i)) and (next <= r) {
            access R(next);
            delay[next]=0;
            schedule time-out(next) in (f*fail(next)) units;
            next = next+1;
        } else if event is retry(i) {
            access R(i);
            delay[i] = backoff(i,delay[i]);
        }
    }
}
```

Figure 3: Access algorithm with retry for failed queries

```
// Resched -- extra messages sent at the shorter of a fraction of the
//            longest failure time for any outstanding message, or
//            the time detection of an actual failure for a replica
//            with a shorter failure time.
resched(int q, site_list R, float f)
{
    int n = |R|;   // number of replicas
    int succ = 0;  // number of successful replies
    int fail = 0;  // number of failed replies
    int next = 0;  // next replica to access
    int extra = 0; // number of extra replicas queried

    for i = 1 to q         // send off q queries
        access R(i);
    schedule time-out(q) in (f*fail(q)) units;
    next = q+1;

    for each event {
        if event is reply(i) {
            succ = succ+1;
            if succ >= q
                return SUCCESS;
        } else if event is failed(i) {
            fail = fail+1;
            if n-fail < q
                return FAILURE;
            else if (next <= n) and (i > extra) {
                access R(next);
                reschedule time-out(next) in (f*fail(next)) units;
                next = next+1;
                extra = extra+1;
            }
        } else if (event is time-out(i)) and (next <= n) and (i > extra) {
            access R(next);
            schedule time-out(next) in (f*fail(next)) units;
            next = next+1;
            extra = extra+1;
        }
    }
}
```

Figure 2: Access algorithm with queries sent on failure

is sent out. Another time-out is requested, again as $f \cdot \text{fail}(i)$. The next time-out is always at the fraction f of the failure time of the most recently sent access.

We can adjust the algorithm by varying f. When $f = 0$, all queries are sent out at once, and when $f = 1$ the algorithm waits until all the initial q queries have responded before sending out additional queries. When f has some intermediate value, additional queries are sent out when the algorithm has waited the fraction f of the longest outstanding failure time-out and not yet obtained a quorum. For example, when $f = 0.5$ the algorithm sends additional queries when a quorum has not been reached by half of the time required to obtain failure notification from the longest-latency outstanding access. Values of f near zero will cause the algorithm to time out and send additional queries soon after the q initial queries are sent, while values of f near one will wait much longer to send additional queries, with the expectation that by waiting longer it is more likely that a quorum can be reached using the messages already sent.

The second algorithm, called `resched` (figure 2), presents an improvement on the simple version. The `simple` algorithm will always wait to send additional queries until a time-out has been reached, even if a replica is determined to have failed before that time. The `resched` algorithm improves this behavior by accessing additional replicas at either the shorter of the time when a replica is known to have failed, or when a time-out occurs. When $f = 0$, all replicas are queried at once, as with the `simple` algorithm. When $f = 1$, additional replicas are queried only when a failure is reported. When $f \approx 0.5$, additional replicas are queried either if a failure is reported, or if a time-out is reached and no additional access has already been sent due to a failure.

Our third algorithm, called `retry` (figure 3), continually retries queries to replicas which are believed to have failed, in the hope that the failure is due to a transient problem. The algorithm will continue to retry replicas until either a quorum has been gathered, or all replicas have been tried. The `retry` algorithm, like `simple`, will only send queries to additional replicas when a time-out is reached, so the success and failure latencies are bounded above by the times for the `simple` algorithm. However, if a nearby replica recovers before a distant replica replies, the `retry` algorithm may be able to declare success sooner than `simple`. This improvement in latency comes at the cost of additional messages as nearby replicas are retried. `Retry` will exhibit a larger failure latency than `simple`, since a failure is not declared until all replicas have been tried, while `simple` will declare a failure when sufficient replicas have failed that it is no longer possible to gather a quorum.

4 Experiments

To determine the actual performance of varying the time of sending extra queries, we constructed a set of discrete-event simulations [Fishman78]. These simulations implement the algorithms as we have presented them. All experiments were conducted using abstract time units, as we are concerned with the relative performance of different algorithms induced by different parameter values rather than absolute performance measures.

The simulations were written in **C**, using a set of locally-written simulation libraries. Each message to a replica was initiated with a *SendMessage* event, which caused either *DetectFailure* or *ReceiveReply* event at a later time, as determined by a sample of the fail(r) and access(r) latency distributions respectively. In addition the algorithm could schedule one or more timers, which produced a *Timeout* event when the timer expired. Timeouts triggered the sending of extra

```
// Simple -- extra messages sent at a fraction of the longest
//           failure time for any outstanding message
simple(int q, site_list R, float f)
{
    int n = |R|;   // number of replicas
    int succ = 0;  // number of successful replies
    int fail = 0;  // number of failed replies
    int next = 0;  // next replica to access

    for i = 1 to q         // send off q queries
        access R(i);
    schedule time-out in (f*fail(q)) units;
    next = q+1;

    for each event {
        if event is reply(i) {
            succ = succ+1;
            if succ >= q
                return SUCCESS;
        } else if event is failed(i) {
            fail = fail+1;
            if n-fail < q
                return FAILURE;
        } else if event is time-out {
            if next <= n {
                access R(next);
                schedule time-out in (f*fail(next)) units;
                next = next+1;
            }
        }
    }
}
```

Figure 1: Simple access algorithm

at once, since delaying any one access could slow down the response. Of course, this approach produces the maximum possible message traffic, since all replicas are queried even though not all replicas need to be queried to establish a quorum. We observe that the lowest network traffic can be achieved by sending out queries one at a time, and ceasing to send queries when we have either established a quorum or know that a quorum cannot be established. Since we must access at least q replicas, we can start by sending q queries and sending additional queries as failures are reported. All our algorithms are parameterized on $0 \leq f \leq 1$, which determines how soon additional queries will be sent. When f is near zero, the algorithms send extra messages sooner than when f is near one.

The first version of our access algorithm, which we call `simple`, is presented in figure 1. The algorithm initially sends as many queries as are required to obtain a quorum. When this initial set of queries has been sent, a time-out is requested at the fraction f of the longest time expected to detect failure for the initial set. Since we have assumed that replicas are ordered by expected failure latency, this time is E(fail(q)). As replies are returned from replicas, the algorithm counts them and exits successfully when a quorum has been reached. As failures are noted, they too are counted, and the algorithm gives up when enough replicas have failed that it is impossible to obtain a quorum. If the algorithm times out, the situation is still indeterminate, and one additional access

Table 1: Simulation variables

Variable	Meaning
n	number of replicas
r	the rth replica, $1 \leq r \leq n$
q	quorum size (number of replicas required for completion)
access(r)	latency of successful requests to replica r
fail(r)	latency of failed requests to replica r
$p_f(r)$	probability replica r has failed
p_0	base probability of failure for experiment
f	fraction of fail(r) for time-out

We use these orderings in our algorithms to select replicas which will provide the fastest response to a message. By assuming a monotonic ordering on expected access time, we can send messages to those replicas we expect will respond most rapidly. Assuming a similar monotonic ordering on failure times we can easily identify the longest time required to detect a failure when querying some set of replicas as the expected failure time of the replica with largest index. The algorithms we present in the next section could be modified to relax this monotonic ordering, though we have not done so in our experiments to date.

In systems which use an RPC protocol similar to the Birrell and Nelson protocol [Birrell84], an operation request is sent to a replica in a single message on an unreliable datagram channel, and the reply message is taken as the acknowledgment of the original request. If the sender does not receive a reply before a time-out occurs, the sender polls the replica to determine whether the replica is available or not. If we assume that normal messages and polls have the same transmission time, then the time to detect a failure is the time for a normal request message plus one or more polls, and our assumed monotonicity conditions are met. Since the actual time required to access a replica or to detect failure is a distribution, we require that replicas be ordered by *expected* access and fail times. This ordering is used in our algorithms to determine which replicas are likely to respond most quickly to a message.

An operation is said to be a *success* if a quorum can be obtained; an operation is said to *fail* if a quorum cannot be obtained because too many replicas have failed. We are interested in four measures of performance: *success latency,* the time required to successfully access the replicated data; the *number of messages* sent in a successful access; *failure latency,* the time required to determine that a replicated object cannot be accessed; and the *number of messages* required to determine failure.

3 Access Algorithm

Our algorithms for accessing replicas balance the latency of a request against the number of messages required to complete the access. To provide the lowest latency, an algorithm must obtain a quorum of replicas at the earliest possible time. This implies sending off queries to all replicas

2 Replication Model

In our performance measurements we measure the time and number of messages required to access a replicated data object. The replicated object is composed of a number of *replicas*, each of which stores a copy of the object being replicated. A *client* can access the replicas to read or write information in the object. Both the client and the replicas reside on *hosts*. All hosts are connected using an internetwork, which consists of several networks with *gateways* connecting the different networks. Sending a message between any two sites on the internetwork may take a variable amount of time depending on the load on the network at the time, and sending messages to different sites may take different amounts of time. The network can lose and reorder messages. We assume that hosts sending a message can use timeouts to detect with high probability that a message has not been received. Hosts cannot, however, distinguish whether a message has been lost due to network failure or due to host failure.

A *replication protocol* is used to control the accesses. The replication protocol specifies what kinds of messages must be sent, and to which replicas, for each kind of access. For any kind of access a *quorum* of replicas must be established; depending on the replication protocol, this might be one replica, all replicas, or some fraction of replicas.

A replica is either available or unavailable. Replicas can be unavailable due to host system failure or network failure, or a replica may appear to be unavailable due to network congestion. We do not attempt to distinguish between these different sources of failure. If a client is unable to gather enough replicas to form a quorum, then an access is said to fail; otherwise it is said to succeed.

We have made several simplifying assumptions in our analysis. Rather than model an internetwork in detail, we have simply assigned a distribution of latencies for successful accesses and failed accesses for each replica. This distribution models all network delays, including transmission time and queuing delay at forwarding sites. We have conducted a series of experiments on the Internet, which lead us to conclude that actual communication times exhibit a complex distribution, depending on the load of the client host and on the topology of the network. In many of our experiments we have approximated this distribution by a normal distribution. We do not consider the effects of contention for network bandwidth on these times. We believe that individual accesses do not significantly affect the load on a network, because an individual access transmits little data, and lasts a very short time. By making these assumptions we were able to use a significantly simpler and faster simulation to obtain performance measurements.

In table 1 we summarize the variables used in our study. We assume that there are n replicas of the data, labeled 1 through n. For any access to complete successfully, a quorum of q replicas must respond to the access request message. If fewer than q replicas respond, the access fails. We associate two functions with each replica r: access(r) is the latency of the operation for replica r if it is available, and fail(r) is the latency of the operation if replica r has failed or is unavailable due to network partitioning. A replica is treated as failed if it is unreachable for any reason; this includes corrupted data, failure of the site holding the replica, and failure of any network gateways between the process performing the access and the replica. We model failure by assigning each replica a probability of failure $p_f(r)$.

We assume that the replicas are ordered by expected access time, so that $\mathrm{E}(\mathrm{access}(r)) \leq \mathrm{E}(\mathrm{access}(r+1))$ for $1 \leq r < n$. We also assume that $\mathrm{E}(\mathrm{access}(r)) < \mathrm{E}(\mathrm{fail}(r))$, and that expected failure times are monotonically increasing so that $\mathrm{E}(\mathrm{fail}(r)) \leq \mathrm{E}(\mathrm{fail}(r+1))$ for $1 \leq r < n$.

connections across a continent. Most LANs use 10–100 megabit/second networks, while many long-distance networks use 56 kilobit/second to 4 megabit/second links. The long-distance links also exhibit much higher communication latency than local network segments. Some local-area networks allow broadcast, while internetworks do not. Broadcast messages on a LAN allow replication protocols to send requests to all replicas in one message, while a message must be sent to each replica in an internetwork, increasing the message traffic required for replication.

An internetwork is shared among more systems than is a local-area network, and the links which connect LANs are often slower than those of the LANs, so traffic must be considered more expensive in an internetwork than in a LAN. Few local-area networks have more than a hundred systems on a single network segment, while internetworks are used to connect hundreds of thousands of systems together. This difference in scale between local- and wide-area networks means that the load on any particular resource which is shared among all systems in an internetwork will be much higher, and that the demands on the bandwidth of the networks will be greater.

Our algorithms for replica access address these differences. In a LAN, access to replicated data can be accomplished using a single multicast message to all replicas. In an internetwork this is infeasible, and simulating a broadcast generates too much network traffic to be feasible in large-scale internetworks. Our algorithms send messages to replicas in a more controlled fashion, and can be tuned to minimize either network traffic or time spent on the access, while taking advantage of the quorum size required by the replication protocol. The algorithms are sensitive to the "distance" of replicas and will tend to communicate with nearby replicas rather than distant ones, providing lower access latencies and limiting the portion of the internetwork affected by an access. One of the algorithms also addresses the problems associated with transient failures in the internetwork.

Since our algorithms are tunable, an access can be parameterized to minimize a cost function of message traffic and latency. One application might place a high cost on latency, and so use a tuning parameter which will provide low latency, though at the cost of extra network traffic. Another application, on the other hand, might not require low latency and place a low cost on latency. In this case the tuning parameter could be selected to minimize the cost of network traffic.

In this article we are concerned with replication *in general,* and not with any one particular replication protocol. All replication protocols maintain consistency between a set of replicas of data. For each operation on the data, some fraction of the replicas are required to participate. For example, the Available Copy protocol [Bernstein84] requires all replicas to participate in a write operation, but any one current replica is sufficient for a read operation. Other protocols, such as Majority Consensus Voting [Davcev85, Jajodia87], require some *quorum* of the replicas to participate. For many operations it is preferable to involve as many replicas as possible, but in some cases a read operation gains no advantage by reading from more replicas than the quorum size. Our algorithms can coexist with other access algorithms, so that a process can improve performance by accessing fewer replicas when that is profitable, or accessing all replicas when necessary.

The remainder of the paper is organized as follows: in §2 we present a generalized model of replication for internetworks. In §3 we describe our methods for accessing such replicated objects. We then describe in §4 a set of simulation experiments we conducted to determine the performance of our techniques, and in §5 the results of those experiments. Finally, we report our future plans in §6 and our conclusions in §7.

Accessing Replicated Data in an Internetwork

Richard Golding *
Darrell D.E. Long
Computer and Information Sciences
University of California
Santa Cruz, CA 95064

August 23, 1990

Abstract

When accessing a replicated data object across an internetwork, the time to access different replicas is non-uniform. Further, the probability that a particular replica is inaccessible is much higher in an internetwork than in a local-area network because of partitions and the many intermediate hosts and networks that can fail. We report three replica-accessing algorithms which can be tuned to minimize either the time spent on the access, or the number of messages sent. We have obtained performance results for these algorithms by simulation. We find an inverse relationship between the time spent processing an access and the number of messages required to complete the access.

1 Introduction

Our goal is to identify efficient techniques for accessing a data object replicated on an internetwork. We have found that replication on an internetwork presents different problems than replication on a local-area network, due to the differences in the structure and uses of each kind of network. These differences in networking between LANs and internetworks makes the techniques used for replication in a local-area network inappropriate for internetwork use.

Internetworks exhibit higher partial failure rates than local-area networks, and partitions are common. An internetwork consists of a set of local networks, connected by gateways. On the Internet, an internetwork which includes many university and industrial local networks throughout the world, many local network segments are connected to the rest of the internetwork by a single gateway. For example, we have found that on the Internet, we are often temporarily unable to connect from the west coast to sites on the east coast. Many of these perceived failures are transient, caused by network congestion, remote host unavailability, or gateway failure.

There are significant differences between the network systems used for local-area networks and those in wide-area internetworks. All the replicas on a local-area network have very similar access times, usually less than ten of milliseconds. On the other hand, replicas on an internetwork have non-uniform access times, and access times are greater, often several hundred milliseconds for

*Richard Golding was supported by Hewlett-Packard Laboratories, Concurrent Systems Project.