[Postel80a] J. Postel. *User Datagram Protocol*, Technical report RFC–768 (28 August 1980). USC Information Sciences Institute.

[Postel80b] J. Postel. *Transmission Control Protocol*, Technical report RFC–761 (January 1980). USC Information Sciences Institute.

[Postel81] J. Postel. *Internet control message protocol*, Technical report RFC–792 (September 1981). USC Information Sciences Institute.

[Pu90] C. Pu, F. Korz, and R. C. Lehman. A methodology for measuring applications over the Internet. Technical report CUCS–044–90 (28 September 1990). Department of Computer Science, Columbia University.

[Sun88] Sun Microsystems. *RPC: Remote Procedure Call, Protocol version 2*, Technical report RFC–1057 (June 1988). USC Information Sciences Institute.

[Thomas79] R. H. Thomas. A majority consensus approach to concurrency control. *ACM Transactions on Database Systems*, **4**:180–209 (1979).

also like to thank Jim Hayes at Apple Computer for his help in measuring network performance. We are also grateful to the referees for their many helpful comments on this paper.

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).

[Carroll89] J. L. Carroll and D. D. E. Long. The effect of failure and repair distributions on consistency protocols for replicated data objects. *Proceedings of 22nd Annual Simulation Symposium* (Tampa, Florida), pages 47–60 (March 1989). IEEE Computer Society.

[Comer88] D. Comer. *Internetworking with TCP/IP: principles, protocols, and architecture* (1988). Prentice Hall, Englewood Cliffs, NJ.

[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).

[Demers88] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. *Operating Systems Review*, **22**(1):8–32 (January 1988).

[Gifford79] D. K. Gifford. Weighted voting for replicated data. *Proceedings of 7th ACM Symposium on Operating Systems Principles* (Pacific Grove, California), pages 150–62 (December 1979). Association for Computing Machinery.

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

[Long88] D. D. E. Long. *The Management of Replication in a Distributed System.* Ph.D. thesis (1988). Department of Computer Science and Engineering, University of California, San Diego.

[Long90] D. D. E. Long, J. L. Carroll, and C. J. Park. A study of the reliability of internet sites. Technical report UCSC–CRL–90–46 (1990). Computer and Information Sciences, University of California, Santa Cruz.

[Metcalfe76] R. M. Metcalfe and D. R. Boggs. Ethernet: distributed packet switching for local computer networks. *Communications of the ACM*, **19**(7):395–404 (July 1976).

[Paris86] J.-F. Pâris. Voting with witnesses: a consistency scheme for replicated files. *6th International Conference on Distributed Computer Systems*, pages 606–12 (1986). IEEECS.

will improve the performance of some algorithms, since failures will likely be detected sooner than they are in our current simulations. Shorter failure time-outs may, however, increase the number of messages sent by algorithms which retry messages. Second, the `retry` and `count` algorithms require a function to determine the delay between retries to a replica. We arbitrarily selected a geometric progression. Other series, including arithmetic and exponential series, are worthy of consideration and may affect performance. Lastly, we arbitrarily chose a retry limit of 5 for the `count` algorithm, which performed well, but we feel that exploring different retry limit values is worthwhile.

All our access algorithms depend on knowing the expected communication latency when communicating with each replica. In our simulations we computed these values *a priori*, which is not reasonable in a real system, since network behavior changes over time. We are examining different mechanisms for computing an expected latency from the data available on the Internet. We are evaluating the use of weighted averages of the communication time with each replica. These mechanisms should allow the algorithms to adapt quickly and gracefully to changes in network topology and to the addition and removal of replicas.

8 Conclusions

We have presented three new algorithms for accessing replicated data. These algorithms form a low-level transport which can be used by replication protocols which need to establish some quorum of a number of replicas to perform an operation. The three algorithms, which we call `reschedule`, `retry`, and `count`, all send messages to nearby replicas before attempting to contact more distant replicas.

In evaluating the performance of these algorithms, we have measured the communication performance of the Internet. Our most significant finding was that a significant fraction of failed messages in long-distance communication are likely due to transient problems, and that we can gain a significant degree of availability by retrying messages only a few times.

There are three criteria which can be used to select between our algorithms, and which can be used to tune them as needed for an application: *availability* is the fraction of operations which successfully obtain a quorum of replicas; *communication latency* is the amount of time spent obtaining the quorum; and *number of messages* determines the amount of load placed on the network.

The `count` algorithm provides the highest availability, especially as the probability of message failure increases. The naive approach of sending messages to all replicas always obtains the lowest latency. Each of our algorithms can be tuned to do this, by setting the tuning parameter p to zero. `Count` generally provides the lowest latency of our three new algorithms. However, it sends more messages than `retry`, particularly when message failure is likely. For values of $p > 0.2$ and with low probability of failure, the number of messages sent is only slightly greater than the minimum required to establish a quorum. This suggests that the range $0 \leq p \leq 0.2$ is a useful range of parameters, in which increasing the number of messages sent decreases the access latency.

9 Acknowledgments

We would like to thank John Wilkes and the members of the Concurrent Systems Project at Hewlett-Packard Laboratories for their comments on an earlier version of this paper. We would

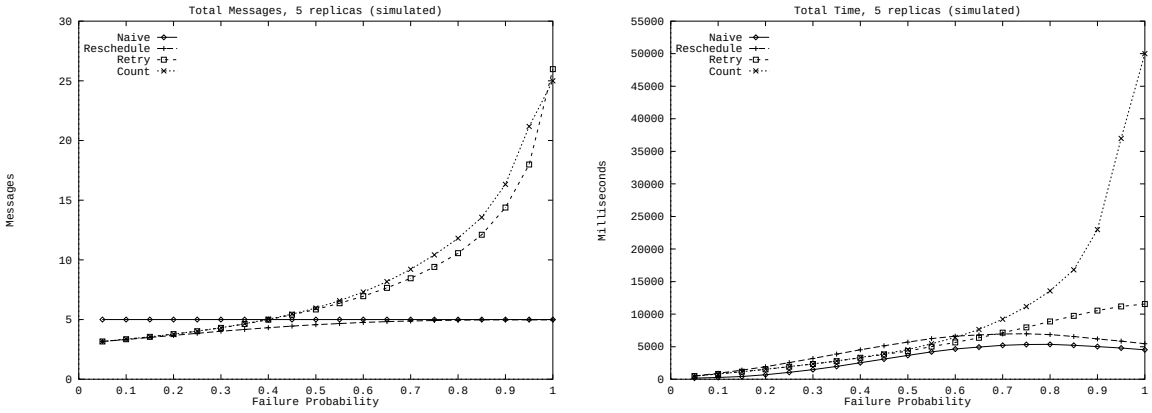

Figure 10: Total messages, and total time, varying f, $p = 0.5$

of message failure suggest that the relative latency and message performances of the algorithms are all similar when the message failure probability is $f \leq .35$. However, under pathological conditions the algorithms behave differently. `Count` can send many messages and take quite a bit of time—50 seconds in our simulations—when no replicas are available. `Retry` is perhaps a more reasonable choice under pathological conditions, succeeding less often than `count` but taking between half and one-fifth as much time. If availability is not of great importance, `reschedule` and `naive` both perform much better than the other two algorithms under high-failure conditions, since they do not retry messages for extended periods of time.

7 Future Work

There are a number of limitations and assumptions in the models and performance evaluation we have performed. We intend to explore these limitations in the near future. The most significant limitation is the number and geographic distribution of hosts used to collect communication latency traces, and the lengths of the traces collected. We intend to repeat these experiments using a much larger sample of hosts on the Internet. In addition, we intend to validate the simulation results by constructing a test application which will be run on several sites around the Internet. We also intend to collect traces over much longer periods of time. Doing so, however, requires placing more load on the Internet for a longer period of time, and thus constitutes more of a disturbance for more people than did our current set of measurements.

The model of message failure used in the second and third sets of simulations only modeled independent single-message communication failures occurring at uniformly distributed intervals. We feel this is an acceptable limitation, given that we considered host failure to be a rare event and given that we were only interested in performance estimates. We intend to improve our simulation to model both host failure and multiple-message communication failure, which will in turn require additional measurements of the Internet to better determine distributions for each of these kinds of failure.

There are three parameters in our algorithms which have not been explored. First, we believe that a more carefully-chosen method of selecting the failure time-out for detecting message failure

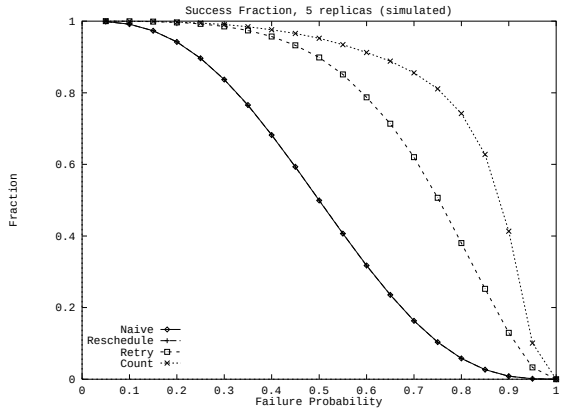

Figure 9: Success fraction, varying f, $p = 0.5$

In our third set of simulation experiments we were interested in the relative behavior of each algorithm as we varied the probability of message failure. We once again used derived distributions rather than measured traces. We chose to fix the delay parameter p at 0.5, chosen because it was close to neither extreme. The lowest value of f sampled was 0.05, which corresponds roughly to the probability of failure in our first two experiments.[2] We did not sample at $f = 0$, because the performance of each algorithm is completely predictable when no failures occur.

The results of this experiment are summarized in figures 9 and 10. As expected, figure 9 shows that `count` can be expected to successfully gather a quorum more often than the other algorithms, and that `retry` will succeed less often than `count`. Both these algorithms succeed more often than `reschedule` and `naive`, which only try each replica once. The data for `naive` match availability figures for Majority Consensus Voting reported in previous work [Paris86], which used Markov analysis to measure the availability of replicated data given reliable communication channels. In that study hosts were only checked once for availability, just as with the `naive` and `reschedule` protocols in our experiments.

Figure 10 shows the number of messages and communication latency required for all operations, whether or not the operation was successful, as the probability of message failure is varied from 0.05 (almost no failures) to 1.0 (all messages fail). As always, `naive` sends one message to each replica regardless of conditions. The number of messages sent by `reschedule` approaches the number of replicas as the probability of failure increases, since it becomes more likely that the algorithm will have to send a message to all replicas. `Retry` sends more messages than `reschedule`, since it will retry messages which fail. This becomes increasingly important as the probability of failure increases. `Count` sends slightly more messages than `retry`, and peaks at 25 messages (5 replicas, limit of 5 retries per replica) when failure is certain at $f = 1.0$.

We can draw a number of conclusions from the results, shown in figure 10, keeping in mind the limitations of the assumptions we have made. These results suggest that `count` provides the highest availability of replicated data of the algorithms we are considering. Once again, this is as expected, since this algorithm will try the most times to contact each replica. The data for low probabilities

[2] Though not exactly, since the third experiment uses a uniform probability of failure f for all replicas, while the first two experiments use different probabilities for different replicas, as obtained from measurements of the actual failure probability for each host.

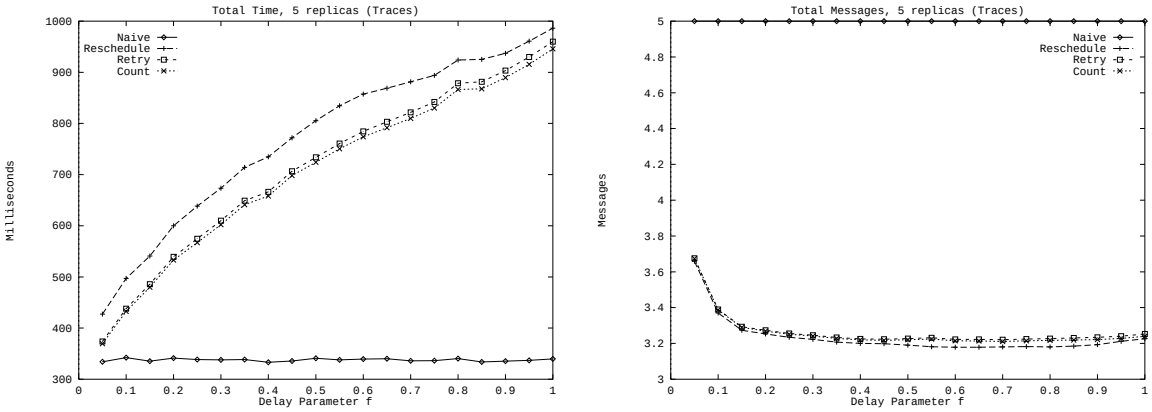

Figure 8: Communication latency and messages for all operations.

highest latency of the three. This is the reverse of their positions for successful operations.

We conclude from the performance measures that one can choose between the algorithms, and tune the algorithms, depending on whether the probability of success, operation latency, or message count are more important. If success is the overriding concern, then `count` should be used. Otherwise, if the probability of message failure is low, then `naive` provides the fastest response, though it sends the greatest number of messages. The other algorithms use fewer messages, at the cost of somewhat greater latency. It appears that the tuning parameter p is best set somewhere in the range 0.1–0.2, which gives nearly the lowest possible number of messages these algorithms use, while requiring less than twice the latency of `naive`.

6.2 Effect of Failure Probability

Our second simulation experiment was intended to validate the use of exponential distributions for communication latency, rather than measured latency traces, so that we could simulate failure probabilities we could not reproduce on the Internet. We derived the distributions as described in §5, and ran simulations for each of the values of p sampled in the first experiment.

While the results of the second set of experiments were not identical to those of the first, they were sufficiently close to those of our trace-based simulations to warrant our confidence that we could proceed with the third set of experiments. The most significant abstraction we made in this set of experiments was to assume that message failures were independent events, even though we have shown that this is not the case in the real Internet. By assuming independent message failure, we increased the probability that each algorithm would obtain a quorum. Furthermore, this assumption made it less likely that the `retry` and `count` algorithms would have to retry many times before successfully sending a message, which decreased both the number of messages and amount of time required for these algorithms to complete an operation. Nonetheless, we found that the results all showed the same *relative* performance among the algorithms we evaluated, and the results for all operations were within about 20% of the expected values for latency and messages. Previous work based [Carroll89] has shown that simulation of various replication protocols was relatively insensitive to the distribution of failures, so we were willing to accept these simulations as a preliminary look at the relative behavior of our algorithms.

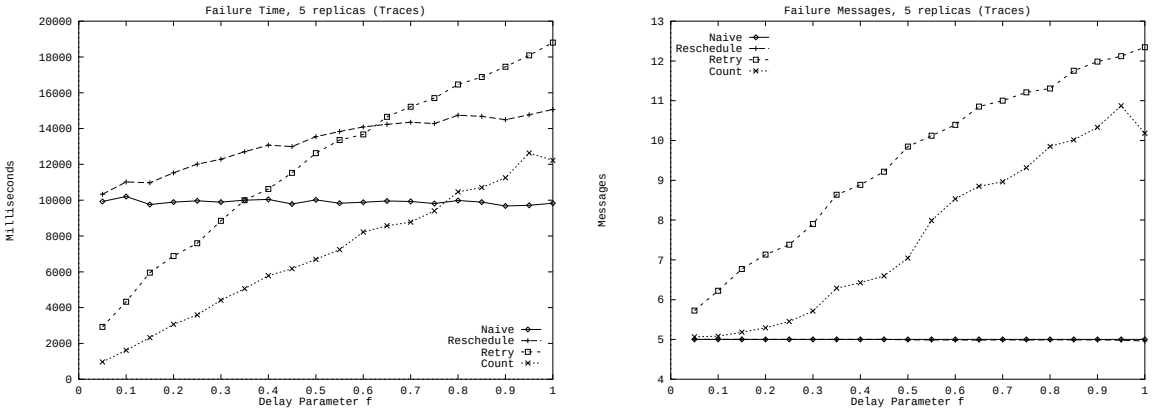

Figure 7: Communication latency and messages for failed operations.

other three algorithms exhibit similar communication latencies, with `count` taking very slightly longer than `retry`, which in turn takes very slightly longer than `reschedule`. `Reschedule` takes less time than the other two because of the rare cases where the `retry` and `count` algorithms must send more than one message to distant replicas to gather a quorum. It also uses the least messages in establishing a quorum for the same reason. `Retry` and `count` each use almost identical numbers of messages, slightly more than the number of messages sent by `reschedule`. We believe that the low probability of failure means that few retries actually occur, so the differences between the two algorithms are not apparent. The number of messages drops to a nearly steady value of about 3.2 for delay parameter $p \geq 0.2$. This value is the number of messages required to communicate with a quorum q of replicas, plus a fraction of a message on the average to handle the uncommon case where one of the q nearest replicas is unavailable.

The performance of the four algorithms is quite different when a quorum cannot be established, as is shown in figure 7. The baseline measure is `naive`, which requires about 10 seconds to determine that a quorum cannot be formed, which is more than an order of magnitude longer than was generally required for success. `Reschedule` requires more time than `naive`, since it must detect just as many failed messages as `naive`, but it may have delayed sending some of those messages. The time required to declare failure increases roughly linearly as p increases. On the other hand, `retry` and `count` require *fewer* messages than `naive` for low values of p. `Naive` must always send $n = 5$ messages, but surprisingly, `reschedule`, despite its ability to declare failure before receiving a reply from all replicas, sends only very slightly fewer than n messages on failure. We believe that this is due to the long values selected for $\mathrm{E}[fail(r)]$, which is used to detect failure, and that a shorter or more intelligent value for the failure latency would improve the performance of `reschedule`. `Retry` and `count` both require more than n messages to establish failure, since both algorithms may try to send multiple messages to a host if the first message fails. `Count` tends to send significantly fewer messages than `retry`, as well as requiring less time.

Figure 8 shows the overall performance of each algorithm. Since the probability of obtaining a quorum is quite high—in excess of 99%—the values for successful operations predominate in these graphs. However, it is worth noting that even with a high probability of success, the low failure latency of `count` makes it the fastest of our three new algorithms, and that `reschedule` has the

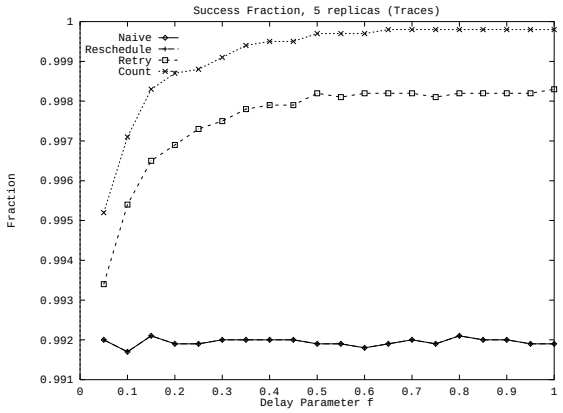

Figure 5: Success fraction, varying p.

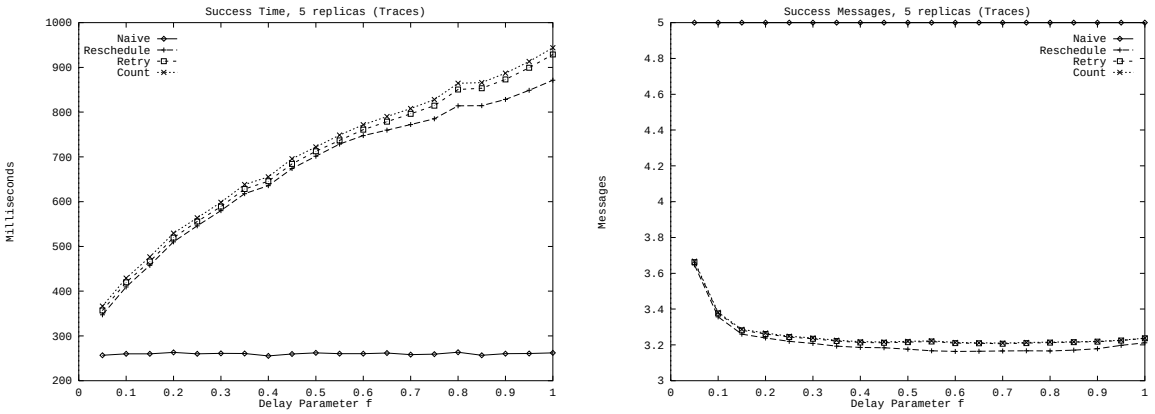

Figure 6: Communication latency and messages for successful operations.

access(r). We found that the exponential fit well for most of the hosts, but failed quite badly for some, particularly the sites in Finland and France. We believe that this failure is due to the nature of the transoceanic connections through which these messages were routed. We were encouraged by previous work on the effect of different distributions on the validity of this kind of simulation [Carroll89] and so decided to proceed with exponentials.

The third set of experiments examined the performance of our algorithms under different failure conditions. Our measurements suggest that the probability of failure is low under normal circumstances, so failure probabilities less than about 0.4 are of interest. However, when a host becomes partitioned from the rest of the network, or there is a pathological condition in the Internet, the probability of a message failing to reach its destination become essentially 1. This allows us to evaluate the performance of our algorithms under worst-case conditions.

In this third set of experiments we fixed the delay parameter p at 0.5, and the probability of failure for each replica was set to some value f. The performance of each access algorithm was measured as f varied from 0.05 (that is, 95% probability of a message being successful) to 1.0 (complete message failure). Since it was not possible to effect particular failure rates on our live internetwork to obtain communication traces, we instead used the exponential fits for each host which were derived for the second set of experiments.

6 Simulation Results

Our three simulation experiments allowed us to examine the behavior of each of the four replica access algorithms (`naive`, `reschedule`, `retry`, and `count`) under different conditions. The performance measures we gathered were the probability of successfully gathering a quorum, the number of messages and the latency required for successful operations, the number of messages and the latency for failed operations, and the total messages and latency for all operations.

6.1 Trace-based Simulations

Our first experiment used the measured samples of network behavior to drive the simulation. The results of this experiment for five replicas are summarized in the graphs in figures 5–8. The results for three and nine replicas are similar.

Figure 5 shows the fraction of all access operations which were successful in gathering a quorum of replicas. The `naive` and `reschedule` algorithms each exhibited an approximately constant success fraction, at about 99.2% of all accesses. Since these two algorithms each attempt to send at most one message to each replica, the delay fraction has no effect on the probability of success. The `retry` algorithm, however, retries nearby replicas more times when the delay parameter p is larger, since there is more time for retries. `Retry` succeeds in approximately 99.8% of all cases when $p \geq 0.5$, while `count` performs even better. In all, `count` is most likely to succeed, with likelihood in excess of 99.9%; `retry` succeeds somewhat less often, but substantially more often than `reschedule` and `naive`.

In all, `count` is most likely to succeed, with likelihood in excess of 99.9%; `retry` succeeds somewhat less often, but substantially more often than `reschedule` and `naive`.

Figure 6 shows the communication latency and number of messages sent for successful operations. We find that `naive` is the fastest of the four algorithms, but it always sends 5 messages. The

intended to determine the relative performance of the access algorithms under widely varying probabilities of message failure. Since these conditions could not be created on our live internetwork, we used synthetic performance distributions instead.

The first experiment used a trace-driven discrete event simulation. The simulation program was coded in **C**, using locally-written simulation libraries. The program was parameterized on the number of replicas n, the quorum size q, the delay parameter p, and the set of host data sets to be traced, and reported the communication latency and number of messages required for both successful operations and failed operations. Each run of the simulator performed 5,000 iterations, where each iteration consisted of choosing n hosts at random from 24 hosts sampled, essentially simulating a random placement of n replicas throughout the Internet. Using our 24 hosts, there are 42,504 possible selections of five hosts, which gives a reasonable probability of each of the 5,000 iterations being independent. Each algorithm was run once for each of the 144 data sets sampled for each of the hosts, yielding approximately 720,000 samples of the performance of each algorithm. The results have a 95% confidence interval with a width of less than 5% for the reported values, and typically ranging from 1% to 3%. The performance of each algorithm was sampled at values of $p = 0.05, 0.1, 0.15, \ldots, 1.0$. We did not include $p = 0$, since all algorithms are equivalent at that value of the tuning parameter.

The simulation of 144 operations for a randomly-selected set of hosts proceeded as follows. First, the n hosts were selected. These n hosts were then ordered by $\mathrm{E}[access(r)]$, which was computed as the data sets were read in. Then one run of each algorithm was performed for each of 144 data sets. When the simulation of an algorithm sent the ith message to replica r, the ith sample for host r (of the 50 samples in a data set) was used to determine whether the message was received, and if so, how long the communication took. Detection of failed messages using timers was simulated by introducing a message failure event, rather than explicitly simulating the timers. The timer duration for host r was set to be the time of the longest-latency reply from that host. This number was determined when the host's data sets were read in. Most implementations of reliable protocols such as TCP use different methods for selecting the failure time-out. A common technique is to base the time-out on a moving average latency of the last several messages [Comer88].

The second simulation experiments duplicated the first, sampling the same performance measures. The performance of each replica access algorithm was sampled at values of p ranging from 0.05 to 1.0 in steps of 0.05, for 3, 5, and 9 replicas. The only difference between the two sets of experiments was that in each simulated run in the second set, the probability of message failure and the latency probability distribution were used to determine the behavior of each message. The sole purpose of this experiment was to validate the accuracy of using derived distributions rather than traces to drive the simulation.

To construct the simulations for the second set of experiments, we constructed distributions for the communication latency and message failure for each host. For message failure, we assumed each message was independent, and had a uniform probability of being received, equal to the overall availability measured in our samples (see table 1). While we have shown that message failure is not independent, we elected to make this simplifying assumption in order to obtain quick results. We intend to repeat these simulations again using more accurate failure distributions. For communication latency, we fit exponential curves to the observed communication latency distributions, such as is shown in figure 4. To fit the exponential, we found the minimum latency $x_{\min}$, and used the estimator $\hat{\lambda} = \mathrm{E}[X - x_{\min}]^{-1}$ to define an exponential probability density function for

Table 2: Fraction of failed messages by size of run.

Length of run	Failure fraction (%) All failures	Communication	Independent
1	50.04	67.81	87.09
2	6.87	9.31	11.63
3	1.33	1.80	1.17
11	1.99	2.70	—
12	3.47	4.70	—
13	1.52	2.07	—
17	0.77	1.04	—
50	26.22	—	—

These values are shown in the fourth column of table 2. Those values less than 10^{-8} are shown as dashes. If message failures were independent, they would generally exhibit many more single-message failures than we observe. The difference between the observed behavior and independent behavior leads us to conclude that message failure is in fact not an independent event, which comes as no surprise.

Our conclusion is that there are two behaviors for message failure: short, transient failures due to temporary network conditions, and longer failures due to host or network failure. These data confirm our suspicion that retrying messages can significantly improve performance or availability. We also believe that retrying more than two or three times likely of little value, since most (61%) of all failures of longer than two messages appear to be due to long-term failure.

We make use of the data collected in our experiment in several ways in determining the performance of our access algorithms. For one set of simulations, we used the poll samples directly in a trace-driven simulation of each algorithm. In addition, we used the message failure and communication latency measurements to derive artificial distributions of $f(r)$ and $access(r)$ for each host, and constructed simulations from these distributions. We discuss this further in the next section.

5 Simulation Model

We studied the performance of our replica-accessing algorithms using discrete-event simulations. In this section we will detail the structure of these simulations.

We conducted three sets of simulation experiments. Each set of experiments consisted of simulating the performance of each of our algorithms when applying the Majority Consensus Voting protocol [Gifford79,Thomas79] for three, five, and nine replicas. The first experiments used the samples of communication latency from our measurements of the Internet. These experiments were intended to provide relative performance information for each access algorithm, given the actual performance of the Internet. The second set of experiments used distributions derived from our measurements to repeat the simulations performed in the first experiments. The intent of these experiments was to validate that artificial distributions gave accurate results when compared with measured Internet performance. The third and final set of experiments also used derived performance distributions, except that we varied the probability of message failure. This experiment was

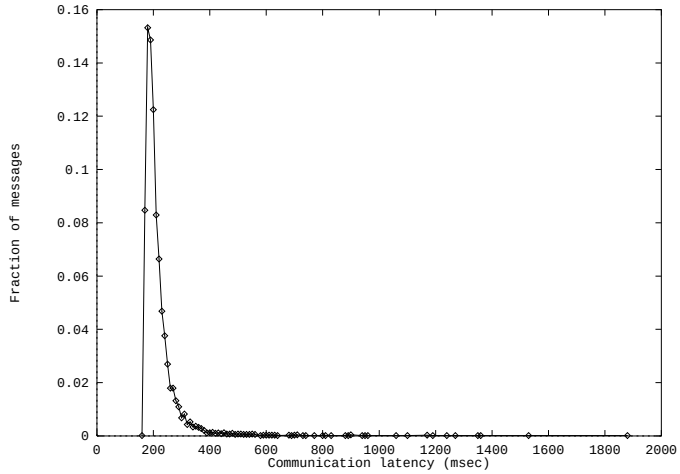

Figure 4: Communication latency for host `lcs.mit.edu` from host `maple.ucsc.edu`.

we sampled. Combining this information with our data on the reliability of hosts, we conclude that communication will succeed most of the time when a host is functioning.

The second focus of our measurements was to determine the time required to communicate with hosts. The single most important measure, the average response latency, is also reported in table 1. These values were obtained on a Sun 4/20 (`maple.ucsc.edu`), which has a clock resolution of 10 milliseconds. The host `spica.ucsc.edu` is on the same network segment as the host from which samples were taken, so the value for `spica` is not particularly reliable. We found that almost all hosts exhibited a curve similar to that shown in figure 4, which shows a histogram of the fraction of messages which fell into each 10-millisecond range, starting from zero, when communicating with the host `lcs.mit.edu`. Most hosts were similar to this host, showing a very few short-latency messages, with a sudden peak dropping rapidly back to zero.

Finally, we examined the data sets to determine how long communication failures lasted. We classified failed polls by the length of the run of failed messages of which they were a member. The first column in table 2 lists those run lengths which contributed to 1% or more of the number of failed messages. The second column reports the percentage of failed messages which were in runs of each length. We found that in more than half the cases where a message failed, the message was part of a run of only one or two failed messages. The only other significant run length was 50, the size of one data set. This large fraction was due to one host being down for some hours. The third column in table 2 lists the percentage of message failures which were not in runs of length 50. We believe that this number approximates the percentage of each run length which is due solely to communication failure, such as congestion, loss of connectivity, or routing loops.

We next compared this distribution to that which would be obtained if all communication failures were independent. This can be modeled as a Bernoulli trial. Given an average message failure probability of $f = 93.32\%$, we computed the probability that a failed message would be part of a run of length n as

$$p(n) = \frac{n(1-f)^n f}{\sum_{i=1}^{\infty} i(1-f)^i f}.$$

Table 1: Hosts selected for traces.

Host	Location	Ave. Response latency (msec)	Availability (%)
acrux.is.s.u-tokyo.ac.jp	Tokyo, Japan	263.68	96.46
andreas.wr.usgs.gov	Menlo Park, CA	26.64	79.90
apple.com	Cupertino, CA	24.50	95.96
beowulf.ucsd.edu	San Diego, CA	57.68	93.33
cs.helsinki.fi	Finland	1525.78	90.42
cs.stanford.edu	Palo Alto, CA	18.50	97.79
fermat.hpl.hp.com	Palo Alto, CA	39.88	97.92
gvax.cs.cornell.edu	Ithaca, NY	162.35	95.28
inria.inria.fr	France	1142.99	84.63
june.cs.washington.edu	Seattle, WA	52.56	97.11
lcs.mit.edu	Cambridge, MA	219.13	89.08
mtecv1.mty.itesm.mx	Mexico	1641.70	91.33
prep.ai.mit.edu	Cambridge, MA	215.97	89.47
sdsu.edu	San Diego, CA	404.54	92.85
slice.ooc.uva.nl	Netherlands	1340.40	88.97
spica.ucsc.edu	Santa Cruz, CA	0.59	100.00
swbatl.sbc.com	Atlanta, GA	298.57	97.06
top.cs.vu.nl	Netherlands	1312.32	90.42
ucbvax.berkeley.edu	Berkeley, CA	24.96	96.43
ucsd.edu	San Diego, CA	51.86	91.15
unicorn.cc.wwu.edu	Bellingham, WA	107.88	96.44
vivaldi.helios.nd.edu	Notre Dame, IN	228.96	96.57
zia.aoc.nrao.edu	Virginia	353.09	97.79

time that hosts have been up, provided that the pattern of up-times is governed by an exponential distribution. The data were gathered by polling hosts using Sun RPC [Sun88] to query `rpc.statd`, the ICMP echo protocol [Postel81] to test availability, and by polling domain servers to obtain host-specific information. Estimates of average MTTF for various Sun 4 systems ranged between 12 and 17 days. These same systems were found to have availabilities in the range of 93% to 97%. The MTTR estimate for Sun 4 models ordinarily used as workstations was approximately 1.2 days, while those models ordinarily used as servers was approximately 0.5 days.

The second study was conducted to obtain data specifically for evaluating the performance of our replica access algorithms. Our measurement methodology was inspired by the techniques used by Pu *et al.* [Pu90]. For this study we selected 24 hosts on the Internet, and polled each of them using the ICMP echo protocol. Each poll consisted of a single datagram message sent from the host `maple.ucsc.edu` to the destination, and one reply datagram. One set of polls was collected for each host every 20 minutes over a 48-hour period, on a Wednesday and Thursday of a normal work week. Each set of polls consisted of 50 ICMP echo requests issued at one-second intervals. This resulted in 7,200 samples for each host. The results of these studies are summarized in table 1.

We first examined the availability of each host, which is reported in table 1. We found that most hosts would respond to a message more than 90% of the time. The one significant exception (`andreas.wr.usgs.gov`) represents a host which was continuously unavailable for 7 of the 48 hours

incremented, and if sufficient have been obtained, the access is declared a success and the algorithm returns. When a message is found to have failed, presumably because a timer in lower layers of software expires, the algorithm schedules a retry for that replica. The first retry occurs immediately, but later retries can be delayed, as determined by the function `backoff`. In our simulations we made each retry delay twice as long as the previous, but we emphasize that this is an arbitrary choice and should be explored further. The delay helps to avoid sending vast numbers of messages to a nearby replica which has failed. The idea of progressively delaying messages was inspired by the collision-handling techniques used in the Ethernet [Metcalfe76].

The `retry` algorithm terminates with failure when either a reply or a communication failure is detected for replica n. At this time, all replicas should have been tried at least once, given our assumption that $\mathrm{E}[\mathit{fail}(r)] \leq \mathrm{E}[\mathit{fail}(r+1)]$. If this assumption does not hold, the algorithm can be modified so that a bit is maintained for each replica, and set to *true* when the first reply or failure is observed for the replica. In that case, the terminating condition would be that all bits are *true,* in place of the two tests for $i = n$.

The `count` algorithm is similar to `retry`, except that a counter l is maintained for each replica and the algorithm stops retrying a replica when it has been retried l times. The terminating condition for the `count` algorithm is thus that all replicas have been tried l times. We expect this algorithm to improve on `retry` in a number of ways. First, by trying each replica a fixed number of times, it should obtain a quorum more often than `retry`, since more distant replicas will be tried more times. We expect that this bound would cause the two algorithms to exhibit significantly different behaviors for communication latency and number of messages when the probability of message failure was high. Second, retrying a fixed number of times evens out the number of times messages are sent to each replica, and our message failure measurements suggest that retrying more than a small number of times is probably of little value, since few communication failures lasted more than two or three messages in our samples. In our simulations we arbitrarily chose a limit $l = 5$. In future experiments we intend to investigate the effect of different limits on algorithm performance.

The `retry` and `count` algorithms are both parameterized on the same tuning value p as the `reschedule` algorithm. In addition, both algorithms use a `backoff` function to determine how to delay retries to a replica, and the `count` algorithm uses a retry limit value. While we have examined how each algorithm behaves as p varies, we have not attempted to examine the effects of different `backoff` functions and different retry limits.

4 Internet Measurements

We have made two separate sets of measurements of the Internet which are relevant to the performance evaluation of the replica access algorithms. The first set of measurements are for a long-term study of the reliability of Internet sites, which are detailed elsewhere [Long90] and briefly summarized here. The second set of measurements were taken specifically for evaluating these algorithms, and are detailed here.

The study of the reliability of Internet sites was conducted by collecting up-time and availability data from several thousand hosts. Data were collected from as many hosts as was practical and then used to derive estimates of availability, mean time-to-failure (MTTF), and mean time-to-repair (MTTR). MTTF is not directly available from hosts, but it can be estimated using the length of

```
// 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 p)
{
    int n = |R|;    // number of replicas
    int delay[|R|]; // time wait for retry of replica i
    int succ = 0;   // number of successful replies
    int next = 0;   // next replica to access

    for i = 1 to q {
        send to R(i);
        delay[i] = 0;
    }
    schedule time-out(q) in (p*E[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   // all replicas have been tried, and
                return FAILURE;   // a quorum has not been obtained
        } else if event is failed(i) {
            if i == n             // again, all replicas have been tried
                return FAILURE;
            else
                schedule retry(i) in delay[i] units;
        } else if (event is time-out(i)) and (next <= r) }
            send to R(next);
            delay[next] = 0;
            schedule time-out(next) in (p*E[fail(next)]) units;
            next = next + 1;
        } else if event is retry(i) {
            send to R(i);
            delay[i] = backoff(i,delay[i]);
        }
    }
}
```

Figure 3: Access algorithm with retry for failed queries.

```
// reschedule -- extra messages sent at the shorter of a fraction of the
//          longest failure time for any outstanding message, or
//          the time of the detection of an actual failure for a replica
//          with a shorter failure time.
reschedule(int q, site_list R, float p)
{
    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 to R(i);
    schedule time-out(q) in (p*E[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) {
                send to R(next);
                reschedule time-out(next) in (p*E[fail(next)]) units;
                next = next + 1;
                extra = extra + 1;
            }
        } else if (event is time-out(i)) and (next <= n) and (i > extra) {
            send to R(next);
            schedule time-out(next) in (p*E[fail(next)]) units;
            next = next + 1;
            extra = extra + 1;
        }
    }
}
```

Figure 2: Access algorithm with extra queries sent on failure.

```
// Naive -- send to all replicas
naive(int q, site_list R)
{
    int n = |R|;
    int i;

    for (i=1; i<=n; i++)
        send to R(i);
    schedule time-out in max(fail(i),i=1..n) units;

    for each event {
        if event is reply(i) {
            q = q-1;
            if q == 0
                return SUCCESS;
        } else if event is time-out {
            return FAILURE;
        }
    }
}
```

Figure 1: Naive access algorithm.

sent to all replicas right away, since the time-out for sending the next message is always set to zero. When p is set to one, `reschedule` only sends additional messages when communication failures are detected. When p is set to one-half, additional replicas are queried either if a failure is reported, or if a time-out of one-half the longest expected communication failure time occurs and no additional access has already been sent due to a failure.

As noted earlier, neither `naive` nor `reschedule` accommodate transient failures. Since our experimental results suggest that more than three-fourths of all message failures are transient,[1] our next two algorithms accommodate transient failure by retrying messages to replicas after detecting a communication failure. These two algorithms, which we call `retry` and `count`, are similar to `reschedule` except that they will continue to retry failed messages, in the hope that the failure was due to some transient problem and the next message will be delivered successfully. They differ in the conditions that they use for determining when to stop retrying. `Retry` continues to retry messages until either a quorum has been obtained or until all replicas have been tried at least once. `Count`, on the other hand, retries each replica at most a fixed number of times.

We expect the retry algorithms to improve both the success latency and the probability that a quorum will be obtained, though at the cost of sending more messages, and possibly at the cost of having longer failure latencies. We therefore expect that the retry algorithms will be most appropriate in situations where short transient failures predominate longer failures, such as are caused by host failure.

Figure 3 shows the algorithm for `retry`. Initially, messages are sent to the q closest replicas, where q is the minimum quorum. When a reply is received, the count of successful replies is

[1] 77.12% of all failed messages were part of a run one or two messages long in our samples. See table 2 in §4.

quorum of replicas at the earliest possible time. This implies sending queries to all replicas at once, since delaying any one message could slow down the response. Of course, this approach produces high message traffic, since all replicas are queried even though not all replicas need to be queried to establish a quorum. This is the approach taken in a naive simulation of broadcast. We observe that the lowest network traffic can be achieved by sending queries one at a time, and ceasing to send queries when we have either established a quorum or have observed sufficient failures to be sure 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. This is the basis for all of our algorithms except `naive`.

As we have noted, the two extremes of sending all messages at once or sending as few messages as possible are not always appropriate for all applications. Each of our algorithms are parameterized by $0 \leq p \leq 1$, which determines how long to delay sending messages. This mechanism allows an application to specify an intermediate position, where more messages than are sent than are strictly necessary with some improvement in the operation latency.

We will compare four algorithms. The first, which we call `naive`, is a straightforward simulation of broadcast, and is shown in figure 1. It operates by sending a message to all replicas. Replies from replicas are counted, and when a quorum has been obtained the algorithm returns, indicating success. The algorithm schedules a time-out for the longest expected reply time. If the time-out occurs before a quorum is obtained, the algorithm assumes that the replicas which have not yet replied are unavailable, and declares the access a failure. There are two problems with this algorithm. The first is that it uses more messages than are strictly necessary, though in doing so it requires the minimum possible time to either obtain a quorum or decide that one is unobtainable. The second is that the algorithm does not account for transient communication failures, thus providing lower availability.

`Reschedule` is the first of our new algorithms, and addresses the first problem with `naive`. This algorithm sends fewer messages than `naive`, though at the expense of requiring more time. It does not attempt to solve the problem of ignoring transient communication failures.

The `reschedule` algorithm sends messages to nearby replicas before sending to more distant replicas. We can create an algorithm which sends the minimal number of messages by initially sending messages to the q replicas with the least expected access time $\mathrm{E}[access(r)]$, and sending messages to additional replicas as the earlier messages are observed to fail. However, this approach will require much longer than `naive` to complete an access in the presence of failures. To ameliorate this problem, the `reschedule` algorithm does not delay sending additional messages until failure is declared. Since message failure is detected by a time-out, additional messages can be sent at some fraction p of the failure time-out. In this way p can be used to tune the `reschedule` algorithm to account for different conditions. Since there are some situations where we may be able to detect communication failure in a shorter period than the designated time-out—perhaps because a negative acknowledgment message has been sent—we also account for early failure detection in our algorithm.

The complete `reschedule` algorithm is shown in figure 2. For clarity, we do not include the code to set a timer for each message sent, but instead assume that it is included as part of sending a message. We assume that when a message time-out occurs, a failure event is reported to the algorithm.

When the tuning parameter p is set to zero, this algorithm is identical to `naive`: messages are

of the host holding the replica, and failure of any network gateways between the host performing the access and the host holding the replica. We model failure by assigning each replica a boolean function $avail(r)$. In some of our simulations $avail(r)$ is defined in terms of a failure probability distribution $f(r)$. Rather than model an internetwork in detail, we have based our analyses on empirical measurements of the Internet, as discussed in §4.

We are interested in six measures of performance: the number of messages required for successful and failed operations; the communication latency of successful and failed operations (*success latency* and *failure latency,* respectively), and the overall number of messages and overall communication latency. We consider our algorithms successful if they require fewer messages or lower latency than a naive simulation of broadcast.

We assume that replicas are ordered by expected access time, so that $\mathrm{E}[access(r)] \leq \mathrm{E}[access(r+1)]$ for $1 \leq r < n$. We also assume that $\mathrm{E}[access(r)] < \mathrm{E}[fail(r)]$, so that messages which are successfully delivered to their destinations are generally not reported as having failed because they took longer than some time-out value. Further, we also assume that $\mathrm{E}[fail(r)] \leq \mathrm{E}[fail(r+1)]$ for $1 \leq r < n$. By assuming a monotonic ordering on expected access time, we can send messages to those replicas we expect will respond most rapidly. These assumptions are not necessary for correct operation of these algorithms, but are needed for optimal performance. The assumption of monotonicity of expected failure time in particular is not important, but simplifies the presentation of our algorithms.

As we have mentioned, the access algorithms are providing a transport mechanism which replication protocols can use to communicate with replicas. Our algorithms fill a niche similar to RPC communication protocols, in that they provide a specialized communication service with particular properties (in this case, communication with q other hosts). In systems which use an RPC protocol similar to the Birrell and Nelson protocol [Birrell84] an operation request is sent to a host in a single message on an unreliable datagram channel, and the reply message is taken as acknowledgment of the original request. If the sender does not receive a reply before a time-out occurs, the sender polls the receiver to determine whether the receiving host 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.

Previous investigations into the performance of replication algorithms [Paris86,Long88] have generally assumed reliable communication channels. Our work extends this, by modeling an unreliable network which can both lose and reorder messages. An unreliable network forces us to examine the question of how to determine when a replica is unavailable. Since we can only detect when a message has failed, not when a host itself has failed, our algorithms must base their determination of replica availability solely on whether they receive a reply from the replica or not. Since communication channels can drop messages, algorithms which only send one message to a replica will report many failures which could have been avoided by sending the same message a second time. We are interested in the performance of algorithms which try to contact a replica more than once, as compared to algorithms which only send one message.

3 Access Algorithms

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, and algorithm must obtain a

implement a lower-level transport mechanism for multiple-site access, much as TCP [Postel80b] provides a reliable byte-stream connection between two sites.

The remainder of this paper is organized as follows. In §2 we present a generalized model of replication in internetworks. In §3 we describe our methods for accessing such replicated objects. We then describe in §4 the set of measurements we have taken of the Internet, and follow this in §5 and §6 with a set of simulations based on these measurements. Finally, we report our future plans in §7 and our conclusions in §8.

2 Replication Model

In this section we will define our model of replication, and establish notation for later sections. We will also discuss how this model compares to other similar systems, and mention the simplifying assumptions we have made.

A 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 local-area networks with *gateways* and *point-to-point links* connecting them. Hosts communicate by sending unreliable datagrams, using a protocol such as UDP [Postel80a]. Sending a message between any two hosts on the internetwork requires a variable amount of time, the *communication latency*. The communication latency depends on the load on the network at the time and the available routes between hosts. By assuming that the internetwork is *unreliable*, we assume it will lose messages, and will deliver them out of order. Hosts sending a message can use time-outs and acknowledgments 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 or host failure.

Replicas are either available or unavailable. Replicas can be unavailable due to host failure (such as a system crash or controlled shutdown), replication software failure, failure of the network (gateway or link failure), or controlled failure (such as removal of a replica which is no longer needed). Replicas may also appear to be unavailable due to congestion or partial failure, such as when a gateway becomes overloaded. We do not attempt to distinguish between these different sources of failure.

A *replication protocol* is used to control the accesses. The replication protocol specifies which messages must be sent, and to which replicas, for each kind of access. For each kind of access the protocol also determines a *quorum* of replicas which must be involved in the operation. Depending on the protocol an operation, this might be one replica, all replicas, or some fraction, and may vary between accesses as replicas fail or become available. 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.*

In our study, we assume there are n replicas of the data, numbered 1 through n. For any access to complete successfully, a quorum of q replicas must respond to an access request message. If fewer than q replicas respond, the access fails. We associated two functions with each replica r: $access(r)$ is the time-varying communication latency of the operation for replica r, and $fail(r)$ is the length of the time-out used to determine when a message has failed. The latency distributions include all communications delays, including transmission time and queueing delay at forwarding sites. A replica is treated as failed if it is unreachable for any reason; this includes corrupted data, failure

point of connection increases the possibility of partitioning the local segment from the rest of the internetwork. Internetworks thus exhibit higher partial failure rates than local-area networks, and partitions are both possible and common. Many of the failures are transient, caused by network congestion, remote host unavailability, or gateway failure.

Replication on an internetwork presents different problems than replication on a local-area network (LAN), due to the differences in the structure and uses of each kind of network. All the replicas on a local-area network have very similar access times, usually less than ten milliseconds. In contrast, access times for replicas on an internetwork are non-uniform, and are greater, often several hundred milliseconds for connections across a continent. Most LANs use 10–100 megabit/second networks, while many long-distance networks use 56 kilobit/second to 45 megabit/second links. The long-distance links also exhibit much higher communication latency than local network segments, especially when satellites are used for transoceanic communication. 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 much 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 more systems may share resources, implying that the potential load on resources which are widely shared in an internetwork will be higher than the load on resources in a LAN.

Our algorithms for replica access address the differences between LANs and internetworks which make the techniques used for replication in a local-area network inappropriate for internetwork use. The algorithms do not require broadcast, nor do they work by simulating broadcast, but instead send messages to replicas in a more controlled fashion. The algorithms can be tuned to minimize either network traffic or access latency, while taking advantage of the quorum requirement of the replication protocol. The algorithms are sensitive to the communication latency 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. Two of the algorithms also address the problems associated with transient failures by resending messages to replicas, which produces a significantly higher availability than can be obtained without retry.

These algorithms are not specific to any one replication protocol. Replication protocols maintain some form of consistency between a set of replicas of data. Many protocols require that each operation on the data be performed by some fraction of the replicas, though this number may vary depending on the kind of operation. For example, the Available Copy protocol [Bernstein84] requires all available 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 [Gifford79,Thomas79] or Dynamic 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 operations gain no advantage from involving more than a minimum number of replicas. Epidemic replication schemes [Demers88], which do not guarantee complete consistency between replicas, can benefit from seeding some fraction of the replicas with an update, before using epidemic techniques to propagate the change to the remaining replicas. Each of these protocols—available copies, voting, and epidemic replication—can be implemented using our access algorithms. The access algorithms

Accessing Replicated Data in a Large-Scale Distributed System

Richard Golding
Darrell D. E. Long
Concurrent Systems Laboratory
Baskin Center for Computer Engineering & Information Sciences
University of California, Santa Cruz

January 10, 1991

Abstract

Replicating a data object improves the availability of the data, and can improve access latency by locating copies of the object near to their use. When accessing replicated objects 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 (LAN) 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 access latency or the number of messages sent. These algorithms assume only an unreliable datagram mechanism for communicating with replicas. Our work extends previous investigations into the performance of replication algorithms by assuming unreliable communication.

We have investigated the performance of these algorithms by measuring the communication behavior of the Internet, and by building discrete-event simulations based on our measurements. We find that almost all message failures are either transient or due to long-term host failure, so that retrying messages a few times adds only a small amount to the overall message traffic while improving both access latency as long as the probability of message failure is small. Moreover, the algorithms which retry messages on failure provide significantly improved availability over those which do not.

1 Introduction

Our goal is to identify efficient techniques for accessing a data object replicated on an internetwork. Replicated data are used to increase availability, to decrease the chances that data will be lost, and to improve performance by locating replicas near where data will be used.

An internetwork consists of a set of local networks, connected by point-to-point links and gateways or routers. Hosts can communicate quickly with other hosts on the local segment, while communications with more distant hosts must pass through several gateways. As the number of systems involved in transmitting a message increases, the latency of communication and the probability that some component will fail both increase. 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. Having a single