[Rice88] J. A. Rice. *Mathematical statistics and data analysis*. Wadsworth and Brooks, Pacific Grove, California, 1988.

[Schroeder84] M. D. Schroeder, A. D. Birrell, and R. M. Needham. Experience with Grapevine: the growth of a distributed system. *ACM Transactions on Computer Systems*, **2**(1):3–23, February 1984.

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

[Tanenbaum81] A. S. Tanenbaum. *Computer Networks*. Prentice-Hall, 1981.

[Terry85] D. B. Terry. *Distributed Name Servers: Naming and Caching in Large Distributed Computing Environments*. PhD thesis, published as Technical report CSL–85–1. Xerox Palo Alto Research Center, CA, February 1985.

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

[Trivedi82] K. Trivedi. *Probability and statistics with reliability, queuing, and computer science applications*. Prentice-Hall, Englewood Cliffs, NJ, 1982.

[Wilkes91] J. Wilkes. The refdbms bibliography database user guide and reference manual. Technical report HPL–CSP–91–11. Hewlett-Packard Laboratories, 20 May 1991.

[Mattern88] F. Mattern. Virtual time and global states of distributed systems. *Proceedings of International Workshop on Parallel Algorithms* (Chateau de Bonas, France, October 1988), pages 215–26, M. Cosnard, Y. Robert, P. Quinton, and M. Raynal, editors. Elsivier Science Publishers, North-Holland, 1989.

[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.

[Oppen81] D. C. Oppen and Y. K. Dahl. The Clearinghouse: a decentralized agent for locating named objects in a distributed environment. Technical report OPD–T8103. Xerox Office Products Division, Palo Alto, Ca, 1981.

[Pâris86] J.-F. Pâris. Voting with witnesses: a consistency scheme for replicated files. *6th International Conference on Distributed Computer Systems*, pages 606–12. Institute of Electrical and Electronics Engineers Computer Society, 1986.

[Pâris88] J.-F. Pâris and D. D. E. Long. Efficient dynamic voting algorithms. *4th International Conference on Data Engineering* (Los Angeles), pages 268–75. Institute of Electrical and Electronics Engineers, February 1988.

[Pâris90a] J.-F. Pâris. Escrow accounts, 28 November 1990. Personal communication.

[Pâris90b] J.-F. Pâris and D. D. E. Long. Protecting replicated objects against media failures. Technical report UCSC-CRL-90-35. Computer Science Department, University of California, Santa Cruz, July 1990.

[Patashnik88] O. Patashnik. Bibtexing, 8 February 1988. Overview document distributed with Bibtex.

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

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

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

[Pu86a] C. Pu. *Replication and nested transactions in the Eden distributed system*. PhD thesis, published as technical report TR–86–08–02. University of Washington, 7 August 1986.

[Pu86b] C. Pu, J. D. Noe, and A. Proudfoot. Regeneration of replicated objects: a technique and its Eden implementation. *Proceedings of 2nd International Conference on Data Engineering*, pages 175–87, February 1986. IEEE. Computer Society Order Number 622.

[Pu90] C. Pu, F. Korz, and R. C. Lehman. A measurement methodology for wide area internets. Technical report CUCS–044–90. Department of Computer Science, Columbia University, 28 September 1990 (Revised March 1991).

[Pu91a] C. Pu and A. Leff. Epsilon-serializability. Technical report CUCS–054–90. Department of Computer Science, Columbia University, 15 January 1991.

[Pu91b] C. Pu and A. Leff. Replica control in distributed systems: an asynchronous approach. Technical report CUCS–053–090. Department of Computer Science, Columbia University, 8 January 1991.

[Ladin90] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Lazy replication: exploiting the semantics of distributed services. Technical report MIT/LCS/TR–484. Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, July 1990.

[Ladin91] R. Ladin, B. Liskov, and L. Shrira. Lazy replication: exploiting the semantics of distributed services. *Position paper for 4th ACM-SIGOPS European Workshop* (Bologna, 3–5 September 1990). Published as *Operating Systems Review*, **25**(1):49–55, January 1991.

[Lamport78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, **21**(7):558–65, 1978.

[Lamport85] L. Lamport. *LaTeX: a Document Preparation System*. Addison-Wesley Publishing Company, Reading, MA, 1985.

[Lantz84] K. A. Lantz and W. I. Nowicki. Structured graphics for distributed systems. *ACM Transactions on Graphics*, **3**(1):23–51, January 1984.

[Lesk78] M. E. Lesk. Some applications of inverted indexes on the UNIX system. Computing Science technical report 69. Bell Laboratories, June 1978.

[Long88] D. D. E. Long and J.-F. Pâris. A realistic evaluation of optimistic dynamic voting. *Proceedings of 7th IEEE Symposium on Reliability in Distributed Software and Database Systems* (Columbus, OH), pages 129–37, October 1988.

[Long89a] D. D. E. Long, J. L. Carroll, and K. Stewart. Estimating the reliability of regeneration-based replica control protocols. *IEEE Transactions on Computers*, **38**(12):1691–702, December 1989.

[Long89b] D. D. E. Long and J.-F. Pâris. Regeneration protocols for replicated objects. *5th International Conference on Data Engineering* (Los Angeles), pages 538–45. Institute of Electrical and Electronics Engineers, February 1989.

[Long90a] D. D. E. Long. Analysis of replication control protocols. Technical report UCSC–CRL–90–13. Board of Studies in Computer and Information Sciences, University of California Santa Cruz, 19 April 1990.

[Long90b] D. D. E. Long and J.-F. Pâris. Voting with regenerable volatile witnesses. Technical report UCSC–CRL–90–09. Computer and Information Sciences, University of California, Santa Cruz, April 1990.

[Long91] D. D. E. Long, J. L. Carroll, and C. J. Park. A study of the reliability of Internet sites. *Proceedings of 10th IEEE Symposium on Reliability in Distributed Software and Database Systems* (Pisa, Italy). Institute of Electrical and Electronics Engineers, September 1991, to appear.

[Mann89] T. Mann, A. Hisgen, and G. Swart. An algorithm for data replication. Report #46. Digital Equipment Corporation Systems Research Center, Palo Alto, CA, June 1989.

[Martin87] B. Martin, C. Bergan, and B. Russ. PARPC: a system for parallel procedure calls. *Proceedings of 1987 International Conference on Parallel Processing*, 1987.

[Demers89] A. Demers, M. Gealy, D. Greene, C. Hauser, W. Irish, J. Larson, S. Manning, S. Shenker, H. Sturgis, D. Swinehart, D. Terry, and D. Woods. Epidemic algorithms for replicated database maintenance. Technical report CSL–89–1. Xerox Palo Alto Research Center, CA, January 1989.

[El-Abbadi86] A. El-Abbadi and S. Toueg. Availability in partitioned replicated databases. *Proceedings of 5th SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 240–51, 1986.

[Ellis90] M. A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, Mass. and London, 1990.

[Garcia-Molina85] H. Garcia-Molina and D. Barbara. How to assign votes in a distributed system. *Journal of the ACM*, **32**(4):841–55, October 1985.

[Garcia-Molina88] H. Garcia-Molina and B. Kogan. An implementation of reliable broadcast using an unreliable multicast facility. *Proceedings of 7th Symposium on Reliable Distributed Systems* (Ohio State University, Columbus, OH), pages 101–11. IEEE Computer Society Press, catalog number CH2612-0, 10–12 October 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. Association for Computing Machinery, December 1979.

[Golding91] R. A. Golding. Distributed epidemic algorithms for replicated tuple spaces. Technical report. Hewlett-Packard Laboratories, in preparation.

[Gopal90] A. Gopal, R. Strong, S. Toueg, and F. Cristian. Early-deliver atomic broadcast (extended abstract). *Proceedings of 9th ACM Symposium on Principles of Distributed Computing*, pages 297–309, August 1990.

[Hisgen90] A. Hisgen, A. Birrell, C. Jerian, T. Mann, M. Schroeder, and G. Swart. Granularity and semantic level of replication in the Echo distributed file system. *Proceedings of Workshop on the Management of Replicated Data* (Houston, Texas, 8–9 November 1990), pages 2–4, L.-F. Cabrera and J.-F. Pâris, editors. IEEE Computer Society Press, November 1990.

[Jacobson90] V. Jacobson. Traceroute man page, ca.1990. Accompanies source code.

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

[Jajodia88] S. Jajodia and D. Mutchler. Integrating static and dynamic voting protocols to enhance file availability. *Proceedings of Fourth International Conference on Data Engineering* (Los Angeles), pages 144–53. Institute of Electrical and Electronics Engineers, 1988.

[Kumar90] A. Kumar. An analysis of borrowing policies for escrow transactions in a replicated data environment. *Proceedings of 6th International Conference on Data Engineering* (Los Angeles, California), pages 446–54. Institute of Electrical and Electronics Engineers Computer Society Press, February 1990.

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

[Boggs83] D. R. Boggs. Internet broadcasting. Technical report CSL–83–3. Xerox Palo Alto Research Center, CA, October 1983.

[Boggs88] D. R. Boggs, J. C. Mogul, and C. A. Kent. Measured capacity of an Ethernet: myths and reality. Technical report 88/4. Digital Equipment Corporation Western Research Laboratory, Palo Alto, CA, September 1988.

[Cabrera84] L. F. Cabrera, E. Hunter, M. Karels, and D. Mosher. A user-process oriented performance study of Ethernet networking under Berkeley UNIX 4.2BSD. Technical report UCB/CSD 84/217 (PROGRES report 84.19). University of California at Berkeley, December 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. IEEE Computer Society, March 1989.

[Carter89] J. B. Carter and W. Zwaenepoel. Optimistic implementation of bulk data transfer protocols. *1989 ACM SIGMETRICS and Performance '89 International Conference on Measurement and Modeling of Computer Systems* (Berkeley, CA), volume 17, number 1, pages 61–9, May 1989.

[Cheriton84] D. R. Cheriton and W. Zwaenepoel. One-to-many interprocess communication in the V-system. Technical report STAN–CS–84–1011. Computer Systems Laboratory, Department of Computer Science, Stanford University, August 1984.

[Cheriton89] D. R. Cheriton. Sirpent: a high-performance internetworking approach. Technical report STAN–CS–89–1273. Computer Science Department, Stanford University, July 1989.

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

[Comer91] D. E. Comer and D. L. Stevens. *Internetworking with TCP/IP: design, implementation, and internals*, volume II. Prentice-Hall, 1991. Implementation-oriented discussion of the IP family of protocols.

[Cooper84] E. C. Cooper. Circus: a replicated procedure call facility. *Proceedings of 4th Symposium on Reliability in Distributed Software and Database Systems*, pages 11–24, October 1984.

[Cristian86] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic broadcast: from simple message diffusion to Byzantine agreement. RJ 5244. IBM Almaden Research Center, 30 July 1986.

[Davčev85] 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.

References

[Alexander87] J. C. Alexander. *Tib: a TeX bibliographic preprocessor*. Department of Mathematics, University of Maryland, 1987. Version 2.1.

[Alon87] N. Alon, A. Barak, and U. Manber. On disseminating information reliably without broadcasting. *Proceedings of 7th International Conference on Distributed Computing Systems* (Berlin, 21–25 September, 1987), pages 74–81, R. Popescu-Zeletin, G. Le Lann, and K. H. Kim, editors. IEEE Computer Society Press, 1987.

[Barbará86] D. Barbará, H. Garcia-Molina, and A. Spauster. Policies for dynamic vote reassignment. *Proceedings of 6th International Conference on Distributed Computing Systems* (Cambridge, Mass), pages 37–44. IEEE Computer Society Press, Catalog number 86CH22293-9, May 1986.

[Barbará89] D. Barbará, H. Garcia-Molina, and A. Spauster. Increasing availability under mutual exclusion constraints with dynamic vote reasignment. *ACM Transactions on Computer Systems*, **7**(4):394–426, November 1989.

[Barbará90] D. Barbará and H. Garcia-Molina. The case for controlled inconsistency in replicated data (position paper). *Proceedings of the Workshop on the Management of Replicated Data* (Houston, Texas), pages 35–8, L.-F. Cabrera and J.-F. Pâris, editors, November 1990.

[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.

[Bernstein87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, Massachusetts, 1987.

[Birman87] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, **5**(1):47–76, February 1987.

[Birman90] K. Birman, A. Schiper, and P. Stephenson. Fast causal multicast. Technical report TR–1105. Department of Computer Science, Cornell University, 13 April 1990.

[Birman91] K. P. Birman, R. Cooper, and B. Gleeson. Programming with process groups: group and multicast semantics. Technical report TR–91–1185. Department of Computer Science, Cornell University, 29 January 1991.

[Birrell82] A. D. Birrell, R. Levin, R. M. Needham, and M. D. Schroeder. Grapevine: an exercise in distributed computing. *Communications of the ACM*, **25**(4):260–74, April 1982.

for *a priori* prediction, and am investigating the correlatability of latency and failure at different hosts sharing routing paths.

has observed a few sequential message failures to a particular host it is likely that host is unavailable.

Based on these traces, I have analyzed the performance of quorum multicast protocols. They provide a robust mechanism for communicating with replicas, and can require fewer messages and less time than naive multicasts. The **count** protocol appears to be a good choice in low-failure situations, while **reschedule** is most appropriate when messages are at a premium or when message failure is likely. **Count** is a good choice when operation success is important, since its persistence causes it to succeed more often than the other protocols.

8.1 Future work

In any work this size there are a number of loose ends that can be tidied up. I have not investigated the operation throughput that can be achieved using quorum multicast protocols. This is of little concern when a distributed service is used but lightly, while a service that is used more heavily must contend with congestion and overload. I intend to extend the performance evaluation to include throughput and a more accurate failure model in the near future.

While simulation and measurement methods produce detailed performance evaluations, analytic methods are often useful for fast, qualitative looks at the effect of one parameter or another. The quorum multicast protocols can be modeled as embedded Markov systems. The model's state transition graph is acyclic, which allows one to apply very simple methods to obtain exact results. I am investigating this model with the intent of measuring the distributions of operation latency.

It would seem certain that the measurements of Internet performance will yield more results. I am investigating the actual correlation between the latency of one message and the next. It is also necessary to establish metrics of quality for latency predictors so that the moving-average method of prediction can be evaluated. I am also interested in methods

Chapter 8

Summary

In this thesis I have presented a family of *quorum multicast* communication protocols that can be used to implement replicated data. These protocols depend on an unreliable datagram service like that provided by the Internet. My three new protocols are called **reschedule, retry,** and **count.**

Quorum multicast protocols provide multicast to a subset of a group of sites. If the subset of sites is selected by shortest expected communication time, the protocols will communicate with the closest available sites and fall back to more distant sites when the nearby ones fail. The size of the subset is specified as the reply count, which is a parameter. By varying the reply count, quorum multicasts can be used as a fault-tolerant one-to-one communication mechanism that contacts 'spare' replicas on failure, to a one-to-many multicast. These protocols provide failure indications to higher level software. If a host has failed, or the network connecting the hosts has failed, these protocols are guaranteed to report a failure. If a host is available, it is unlikely (but not impossible) that the protocol will report a failure.

I recorded traces of Internet communication behavior as part of the performance evaluation. The analysis of these traces indicates that the communication latency distribution for most sites is reasonably predictable. The analysis also shows that failed messages generally occur in short runs of a few failures, and that these failures are not independent. These observations lead me to the conclusions that communication behavior of the next message can be predicted from the behavior of recent messages, and that when a protocol than the others.

7.4 Summary

The measurement experiment was largely a success. It validated most of the results obtained by simulation. The primary differences arose because the hosts showed fewer failed messages than those in the traces that drove the simulations. An error in the client invalidated the results for failed operations for the **count** protocol, but successful operations showed the expected behavior. Overall, the measurements confirm the conclusion that quorum multicast protocols can provide significant performance advantages for wide-area applications, and show that there is a trade-off among latency, traffic, and operation success.

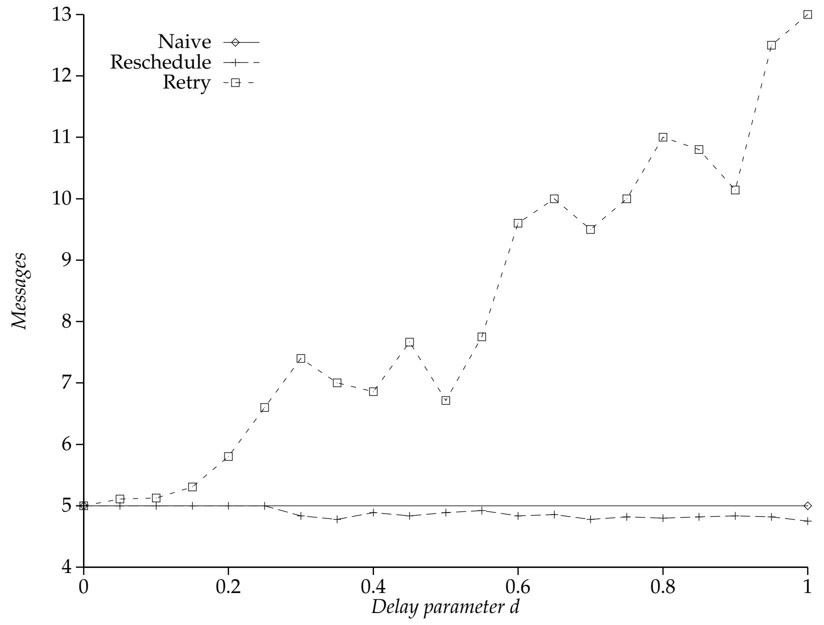

FIGURE 7.6: Measured message count for failed operations. Group 2 measured from midgard.ucsc.edu.

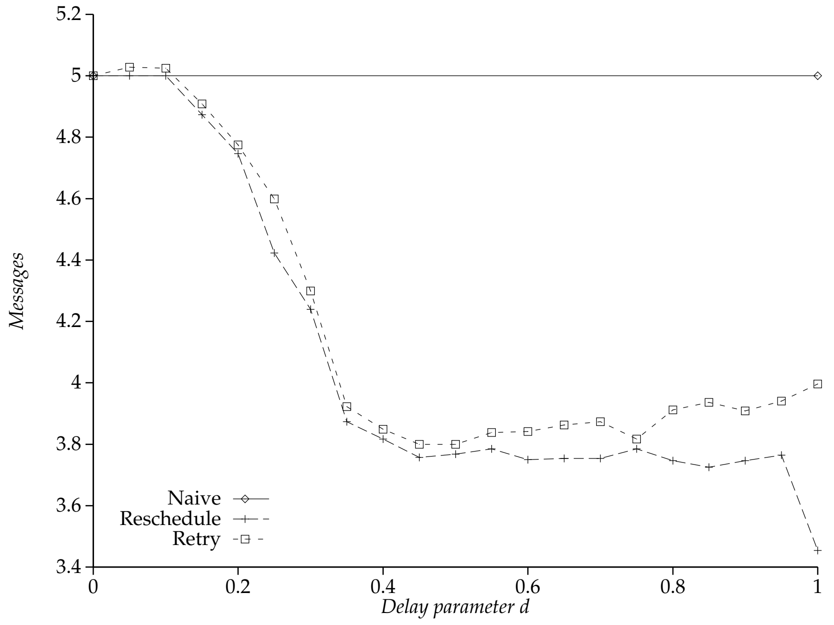

FIGURE 7.7: Measured message count for all operations. Group 2 measured from midgard.ucsc.edu.

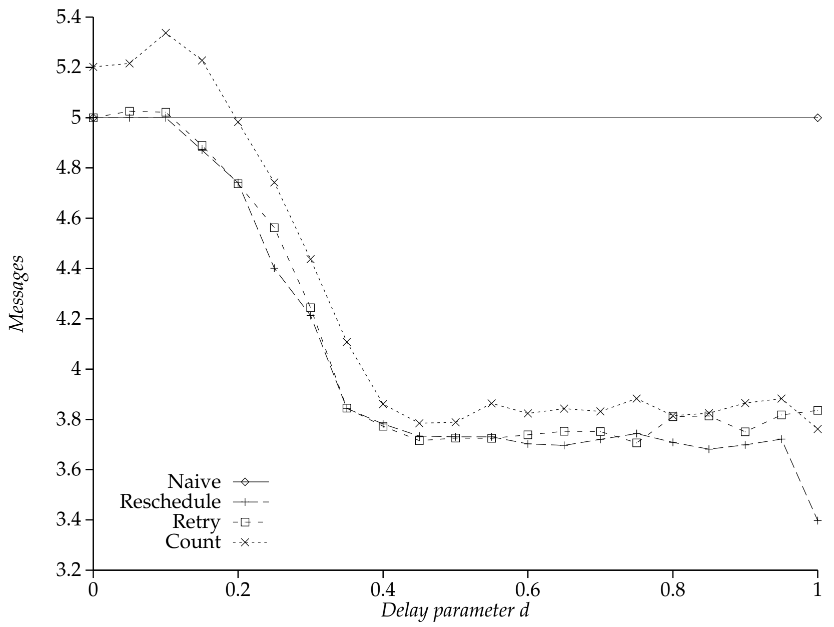

FIGURE 7.5: Measured message count for successful operations. Group 2 measured from midgard.ucsc.edu.

$d = 0.4$ and flattening out thereafter. The 95% confidence intervals on these results are uniformly less than 5%, and generally much smaller. Since these hosts exhibited fewer message failures than did those in the 125-host traces, this experiment should show little difference between the **reschedule**, **retry**, and **count** protocols.

The measured results for failed operations are again similar to simulation results, as seen in Figure 7.6 (compare Figure 6.6). **Naive** and **reschedule** each use five messages as always. **Retry** uses an increasing number as d increases, but the count is much lower than in simulation results because there were fewer message failures. The results for **count** are omitted due to the error in the client.

Finally, the overall message count results in Figure 7.7 are similar to those obtained by simulation (compare Figure 6.7). As always the results are dominated by successful operations. The low number of messages **retry** used in failed operations makes it competitive with the other protocols, unlike the simulation results where it used far more messages

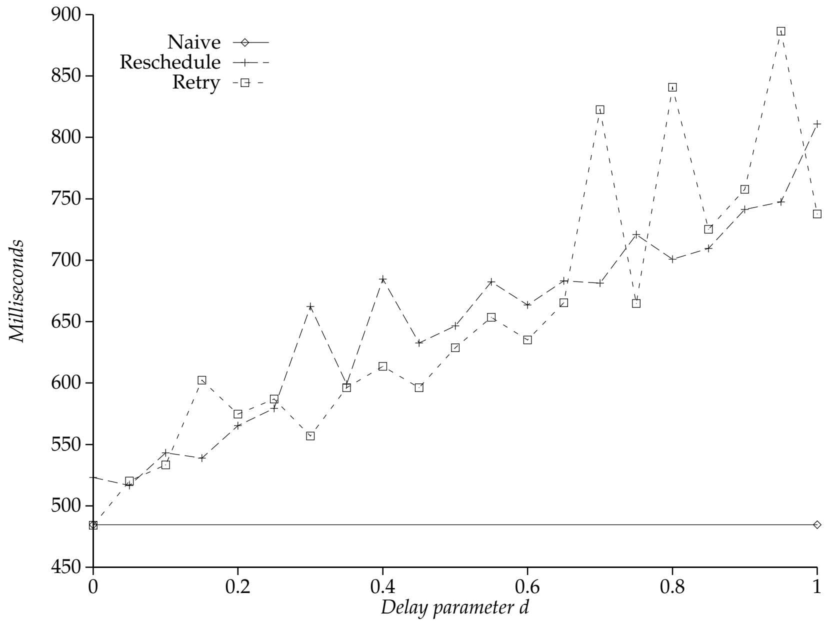

FIGURE 7.4: Measured communication latency for all operations. Group 2 measured from midgard.ucsc.edu.

---

d, the measured results show that the two have quite similar latencies. The sample size is small enough that this result is inconclusive. The results for **retry** are consistent with simulation results. The spikes at high values of d are due to the small sample size, where the data exhibit extremely wide confidence intervals. The results for **count** are omitted due to the client error.

As seen in Figure 7.4, despite the differences with failed operations, the overall results are consistent with simulation results (compare Figure 6.4) since the overall success rate was in excess of 95%. The **reschedule** and **retry** protocols all exhibited an increasing latency as d ranges from zero to one. **Count** is omitted from this graph.

The message counts for operations that met their reply count are reported in Figure 7.5 (compare Figure 6.5). The three new quorum multicast protocols all show similar behavior, unlike the simulation results where **reschedule** used significantly fewer messages than the others. However, the curves have about the same shape, decreasing from $d = 0.0$ to

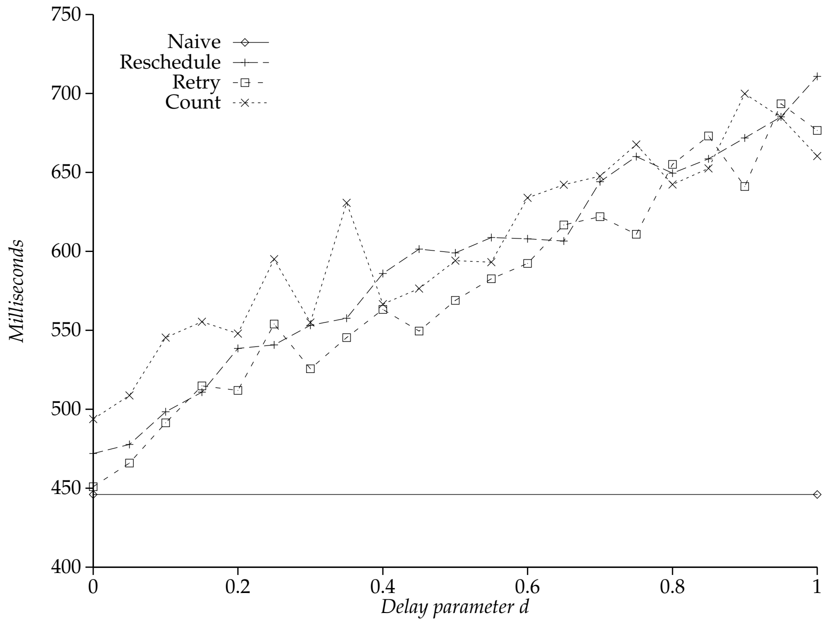

FIGURE 7.2: Measured communication latency for successful operations. Group 2 measured from midgard.ucsc.edu.

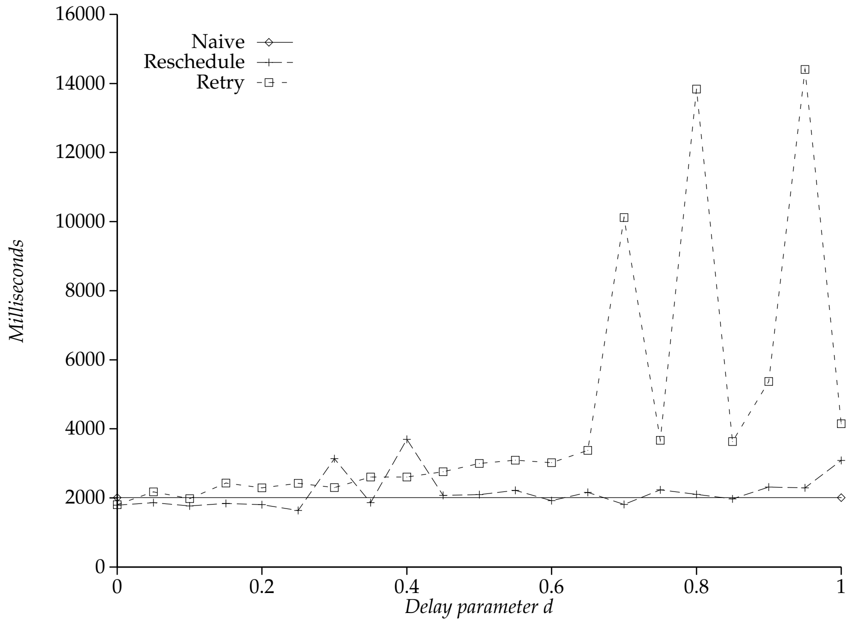

FIGURE 7.3: Measured communication latency for failed operations. Group 2 measured from midgard.ucsc.edu.

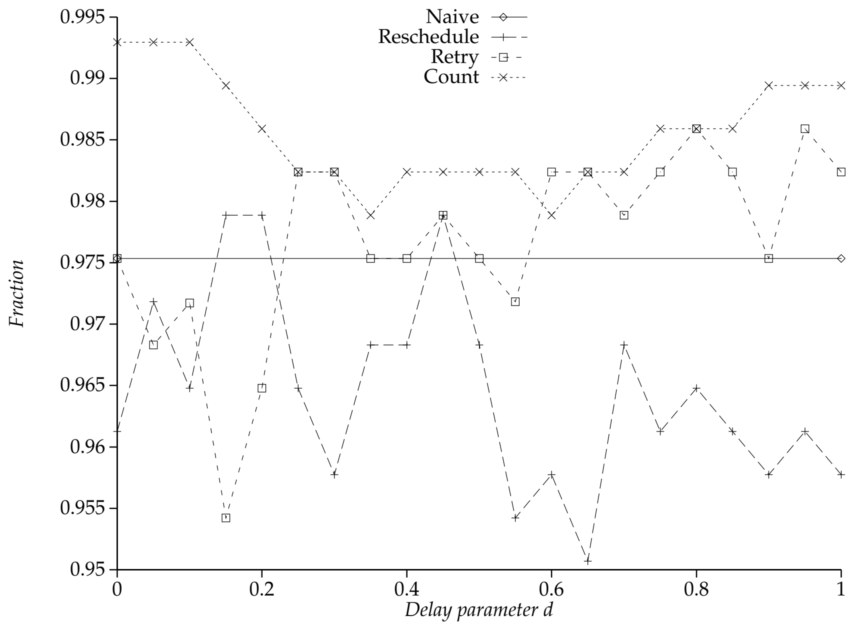

FIGURE 7.1: Measured success fraction. Group 2 measured from midgard.ucsc.edu.

Figure 7.1 shows the success fraction for each protocol (compare Figure 6.1). The data show that all four protocols met the reply count more than 95% of the time. **Count** succeeded more often than the other protocols for almost all values of d, with **retry** generally succeeding more often than **naive** and **reschedule**. These results are similar to the simulation results.

Figure 7.2 shows the mean latency required for successful operations (compare Figure 6.2). Again the results appear to be similar to those obtained by simulation. **Naive** is generally the fastest of the protocols, and the latency of the other three protocols increases approximately linearly in the delay parameter d. **Count** appears in general to require slightly more time than the other protocols, but the difference is somewhat smaller than the half-width of the confidence interval so this measurement is inconclusive.

The results for failure latency, shown in Figure 7.3, are somewhat different from those obtained by simulation (compare Figure 6.3). While simulation indicated that **reschedule** would take more time than **naive** to declare failure, and that this time would increase with

TABLE 7.2: Summary of host performance.

	From midgard.ucsc.edu		From slice.ooc.uva.nl		From beowulf.ucsd.edu	
	Message success	Average latency (ms)	Message success	Average latency (ms)	Message success	Average latency (ms)
asc.cise.nsf.gov	85%	280	84%	263		
aupair.cs.athabascau.ca	70%	1723	87%	2699		
baldulf.cs.purdue.edu	92%	103	98%	310		
beowulf.ucsd.edu	88%	67			96%	11
bugs.sics.se	87%	449	93%	233		
cricket.ece.arizona.edu	95%	141			95%	163
dslab2.cs.uit.no	88%	507			89%	559
elettra.dm.unibo.it	82%	314	97%	81		
fysap.fys.ruu.nl	92%	332	97%	37		
helpdesk.rus.uni-stuttgart.de	82%	2246			79%	2207
hornet.cs.vu.nl	93%	7204			94%	404
iggy.gw.vitalink.com	91%	334	94%	367	89%	337
issun3.stc.nl	88%	557	73%	571		
longtarin.irisa.fr	94%	437	92%	213		
manray.berkeley.edu	95%	38			93%	81
midgard.ucsc.edu	97%	17	97%	362		
nachiketa.ms.uky.edu	15%	183			7%	684
odin.ethz.ch	94%	309	97%	107		
plains.ndsu.nodak.edu	91%	372			91%	362
relay.cs.toronto.edu	90%	219	97%	248		
scott.prime.com	66%	288	94%	1101		
skydesign.arc.nasa.gov	22%	47			24%	181
sol.acs.unt.edu	94%	116			91%	82
sole.cs.ucla.edu	3%	128			0%	—
spud.img.rit.edu	5%	301	89%	318		
tik.vtt.fi	53%	758	94%	391		
tnofel.fel.tno.nl	85%	603			85%	662
tyrannosaurus.scrc. symbolics.com	41%	172	85%	372		
ubitrex.mb.ca	93%	459	97%	770		
yragael.inria.fr	92%	310	91%	211		
zeno.gso.uri.edu	59%	243	95%	359		

parameter d in the same way. The measured 95% confidence intervals for latency are generally around 10 to 15%, and generally less than 5% for messages. However, there are exceptions, which will be noted with the appropriate graphs. Further, given the sample size even tight confidence intervals may not be meaningful since their coverage could be low.

The graphs presented here show the results obtained using midgard.ucsc.edu to poll the second group. This group consisted of hosts from all over North America and Europe, and four of the hosts were generally available. The average communication latency ranged from 38 ms to 603 ms.

For these groups, one server host was selected from each of the Netherlands, the rest of Europe, eastern US and Canada, western US, and the Bay area. The sixth group consisted of sites in California, and the seventh of sites in the eastern half of North America. These two groups modeled the behavior of regionally-replicated data.

The client program polled these groups from three client hosts: midgard.ucsc.edu, a host at UC Santa Cruz; beowulf.ucsd.edu, a host at UC San Diego; and slice.ooc.uva.nl, a host at the Universiteit van Amsterdam. Every group was measured from midgard and either slice or beowulf. Each of these sites is connected to the Internet by a different regional network. Measurements from the two sites in California are generally similar, suggesting that the results are not significantly biased by the BARRnet (to which UC Santa Cruz is connected).

Table 7.2 lists the overall message success rate and average latency for all 31 server hosts. Some of them belonged to more than one group and so were polled from all three client hosts.

7.3 Measurement results

In this section I present the results of the measurement experiment, and compare them to those obtained by simulation. The two sets of results are not identical since different hosts were used for the measurements, as can be seen by comparing Table 7.2 with Table 5.1. Further, the simulation results are the averages of the performances of several thousand combinations of hosts, while each measurement in this chapter used a single fixed selection of hosts. Finally, the number of samples obtained for each measurement are much smaller in this chapter: about 400 per protocol compared with half a million per protocol in the simulations (504 data points, one thousand iterations minimum).

Agreement between these results and simulation results is therefore not quantitative, but qualitative. The measured results confirm the simulation results if the relative performances of each protocol are similar, and if each measurement varies with the delay

TABLE 7.1: Systems participating in measurement experiment.

Host	Type	Organization
asc.cise.nsf.gov	Sun 4	Raleigh Romine, NSF
aupair.cs.athabascau.ca	Sun 4	Lyndon Nerenberg, Athabasca Univ.
babar.acs.oakland.edu	VAX 3100	Thomas Hacker, Academic Computing Services, Oakland Univ.
baldulf.cs.purdue.edu	Sun 4	Shawn Ostermann, Internetworking Research Group, Purdue Univ.
beowulf.ucsd.edu	Sun 4	U. C. San Diego
bugs.sics.se	Sun 4	Stephen Pink, Swedish Institute of Computer Science
cricket.ece.arizona.edu	Sun 4	(anonymous)
django.colorado.edu	DEC 3100	Al Marti, Univ. Colorado Boulder
dslab2.cs.uit.no	HP 9000/300	Tage Stabell-Kulø, Dept. of CS, Univ. of Tromsø
elettra.dm.unibo.it	Sun 4	Özalp Babaoğlu, Dept. of Mathematics, Univ. of Bologna
estella.daimi.aau.dk	HP 9000/300	Michael Glad, CS Dept., Aarhus Univ.
fysap.fys.ruu.nl	HP 9000/300	Peter Mutsaers, Physics Dept., Rijksuniversiteit Utrecht
helpdesk.rus.uni-stuttgart.de	SGI PI	Kurt Jaeger, Comp. Center, Univ. of Stuttgart
hornet.cs.vu.nl	Sun 4	Kees J. Bot, Dept. of Math. and CS, Vrije Universiteit, Amsterdam
ibminet.awdpa.ibm.com	IBM RT/PC	Steve DeJarnett, IBM Personal System Programming, Palo Alto
iggy.gw.vitalink.com	Sun 4	Erik J. Murrey, Vitalink Communications Corp.
issun3.stc.nl	Sun 4	Jon Wilkes, SHAPE Technical Centre, The Hague
longtarin.irisa.fr	Sun 4	Patrick Sanchez, INRIA-IRISA (Rennes) Campus de Beaulieu
manray.berkeley.edu	Sun 4	George Neville-Neil, Mammoth Project, UC Berkeley
midgard.ucsc.edu	Sun 4	Concurrent Systems Lab, UC Santa Cruz
nachiketa.ms.uky.edu	Sun (4?)	Raj Yavatkar, Dept. of CS, Univ. of Kentucky
nro.cs.athabascau.ca	Sun 3	Lyndon Nerenberg, Athabasca Univ.
odin.ethz.ch	Sun 4	Rudolf Baumann, Institut für Molekularbiologie und Biophysik, ETH Hönggerberg
plains.ndsu.nodak.edu	Solbourne 5	Blayne Puklich, North Dakota State Univ.
relay.cs.toronto.edu	Sun 4	Ken Lalonde, Univ. of Toronto
scott.prime.com	Sun 3	Graeme Williams, Constellation Software Inc.
skydesign.arc.nasa.gov	Sun 3	RIACS, NASA Ames Research Center
sol.acs.unt.edu	Solbourne 5E	Billy Barron, Univ. of North Texas
sole.cs.ucla.edu	HP 9000/300	Robert J. Collins, CS Dept., UC Los Angeles
spud.img.rit.edu	Sun 4	Lance Ware, Center for Imaging Science, Rochester Inst. of Tech.
tik.vtt.fi	HP 9000/800	Tor Lillqvist, Technical Research Centre of Finland
tnofel.fel.tno.nl	Sun 4	Rene van den Assem, TNO Physics and Electronics Lab
tyrannosaurus.scrc.symbolics.com	Sun 3	Randolph K. Zeitvogel, Symbolics MACSYMA Division
ubitrex.mb.ca	Sun 4	Danny Boulet, Ubitrex Corp.
uop.uop.edu	Sun 3	Nick Sayer, CS Dept., Univ. of the Pacific
yragael.inria.fr	Sun 4	Philippe Mussi, INRIA
zeno.gso.uri.edu	Macintosh II	James Gallagher, Grad. School of Oceanography, Univ. of Rhode Island

7. relay.cs.toronto.edu, scott.prime.com, spud.img.rit.edu, tyrannosaurus.scrc.symbolics.com, zeno.gso.uri.edu

The first five of the seven groups were spread evenly over the sample area, allowing measurement of the performance of quorum multicasts to very widely distributed replicas.

client logged all its operations to a file, from which I verified that the other results were correct, and that the values reported for successful **count** multicasts were correct.

7.2 Experiment design

The hosts for this experiment were selected by the people running them, rather than randomly as in the measurements in Chapter 5. This self-selection may introduce a different bias to these measurements than was observed in previous measurements. Table 7.1 lists the 37 sites that participated in this experiment.

These sites were distributed all over North America and Europe. I divided them into seven groups: five sites in the San Francisco Bay area; six in the rest of the western United States; nine in the eastern United States; four in Canada; four in the Netherlands; five in the rest of northern Europe; and four in southern Europe. No responses were obtained from other parts of the world – the lack of sites in Japan is notable.

I initially polled all hosts several times to ensure all the daemons were working. Some daemons did not respond to these initial polls, and attempts to identify and solve the problem were unsuccessful. I excluded these sites from the remainder of the experiment. There did not appear to be a geographic bias to the sites that were excluded.

The 31 functioning sites were arranged into seven groups of five "server hosts" each (with some overlap). Each group was polled using all four quorum multicast protocols, with a reply count of three. The groups were:

1. issun3.stc.nl, longtarin.irisa.fr, ubitrex.mb.ca, asc.cise.nsf.gov, midgard.ucsc.edu
2. tnofel.fel.tno.nl, dslab2.cs.uit.no, cricket.ece.arizona.edu, nachiketa.ms.uky.edu, manray.berkeley.edu
3. fysap.fys.ruu.nl, odin.ethz.ch, aupair.cs.athabascau.ca, baldulf.cs.purdue.edu, iggy.gw.vitalink.com
4. hornet.cs.vu.nl, helpdesk.rus.uni-stuttgart.de, sol.acs.unt.edu, plains.ndsu.nodak.edu, skydesign.arc.nasa.gov
5. tik.vtt.fi, bugs.sics.se, elettra.dm.unibo.it, yragael.inria.fr, issun3.stc.nl
6. manray.berkeley.edu, iggy.gw.vitalink.com, skydesign.arc.nasa.gov, sole.cs.ucla.edu, beowulf.ucsd.edu

7.1 Daemon and client structure

The experiment consisted of a server daemon that ran continuously at several sites on the Internet, and a client program that ran every half hour at three sites. The clients sent packets containing 64 bits of data: a 32-bit sequence number and a 32-bit delay value, while the daemons replied with a 192-bit packet containing the sequence number, delay value, and two 64-bit timestamps.

The server daemon was coded in portable C, and used the BSD socket mechanism to passively listen for UDP packets on port 4296. When the daemon received a packet, it built a reply packet and recorded an arrival timestamp. It then waited a number of seconds as specified by the delay value, then recorded a departure timestamp and sent the reply packet. The delay mechanism was intended to simulate the effects of long-duration computations in the server, but was not used in this experiment.

The client was coded in C++, and used the BSD socket mechanism to send UDP messages. It also used the system SIGALRM timer to implement message failure and delay timeouts. The program read a series of operation specifications, and started one operation every 10 seconds. These operation specifications indicated an experiment number, the protocol to be used, the reply count, and a list of server IDs.

The client also used a file of information on each server, including the server's ID, the name of its host, its UDP port, and moving averages of message latency and failure probability. The client updated an in-memory copy of these records every time a message was received, and wrote them back to disk after all operations were processed. This file was initialized with pessimistic values: all hosts were assigned an expected latency of 5 seconds and a message failure probability of 95%. The client detected message failure using a timer set to the 95th percentile expectation, as described in §5.5. The values quickly converged to more normal values as the experiment progressed.

There was an error in the client that invalidates one of the results it measured. It appears I implemented the termination condition for the **count** protocol incorrectly, and so the message count and latency for failed operations for that protocol are incorrect. The

Chapter 7

Measured performance evaluation

In the last chapter I used simulation to back up the claim that quorum multicast protocols have advantages for implementing replication protocols. These simulations could not always model all of the relevant effects of a system, such as the effects of loading a client system or gateways, and timer accuracy was unrealistically high.

In this chapter I measure an application running on the Internet to substantiate the simulation results. This application was structured as a client communicating with servers. The client ran on a few hosts, and sent UDP packets to the servers. The server was a simple daemon that listened for UDP packets on a particular port, and echoed them back to their origin. I published the source code for the daemon on Usenet, and 37 people ran copies on their systems. I then ran a test client on three systems, two in the United States and one in the Netherlands, to collect latency, message count, and success measurements as I did with the simulations.

These measurements will confirm the simulations results if the performance of each protocol relative to the others is similar. The measured performance will be different than the simulated performance, since the two used different sets of hosts. However, the shape of each curve should be similar, allowing for differences in communication failure probability and latency.

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 protocols under high-failure conditions, since they do not retry messages for extended periods of time.

6.6 Summary

I used discrete-event simulation as the first method to evaluate the performance of quorum multicast protocols. Using traces of message behavior from the measurements of the Internet, I found that those protocols that persistently retry messages upon failure get successful operations more often. The latency of an operation increases roughly linearly as the delay parameter d increase from 0 to 1, with the **count** protocol generally the fastest and **reschedule** the slowest. The number of messages sent per operation decreases as d ranges from 0 to about 0.4 in the 125-host traces, and from 0.4 to 1 the number of messages is roughly stable. The **reschedule** protocol uses the least messages, and **retry** the most.

I also examined the effects of using a longer message failure timeout. My conclusion is that larger timeout periods increase latency significantly, while not changing message behavior. The protocols' performance appears to be sensitive to the distribution of failures. Finally, I found that the **count** and **retry** protocols have troublesome behaviors when the probability of message failure is high. The other protocols require fewer messages and lower latency at the expense of succeeding less often.

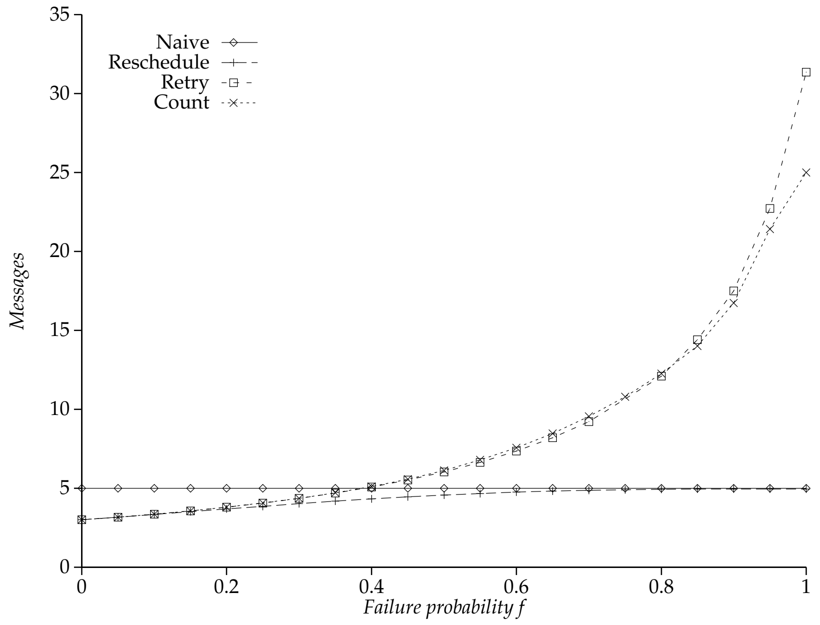

FIGURE 6.12: Total messages, varying f, $d = 0.5$.

particularly when the probability of message failure f approaches unity. The **count** protocol is limited to sending at most 25 message (5 replicas, 5 messages per replica), while **retry** can send an unbounded number of messages in the worst case. When the simulator used latency distributions derived from the 24-host data set, **retry** used slightly fewer messages and – by coincidence – peaked at the same value as **count**. The disparity between the latency of distant and nearby hosts was smaller in the 24-host data set, which caused **retry** to send fewer messages.

There are a number of conclusions to be drawn from these results, keeping in mind the limitations of the assumptions I have made. The results suggest that **count** provides the highest availability of replicated data of the protocols we are considering. Once again, this is as expected, since this protocol will try the most times to contact each replica. The data for low probabilities of message failure suggest that the relative latency and message performances of the protocols are all similar when the message failure probability is $f \leq 0.2$. However, under pathological conditions the protocols behave differently. **Count** can send many messages and take quite a bit of time – more than 100 seconds in our

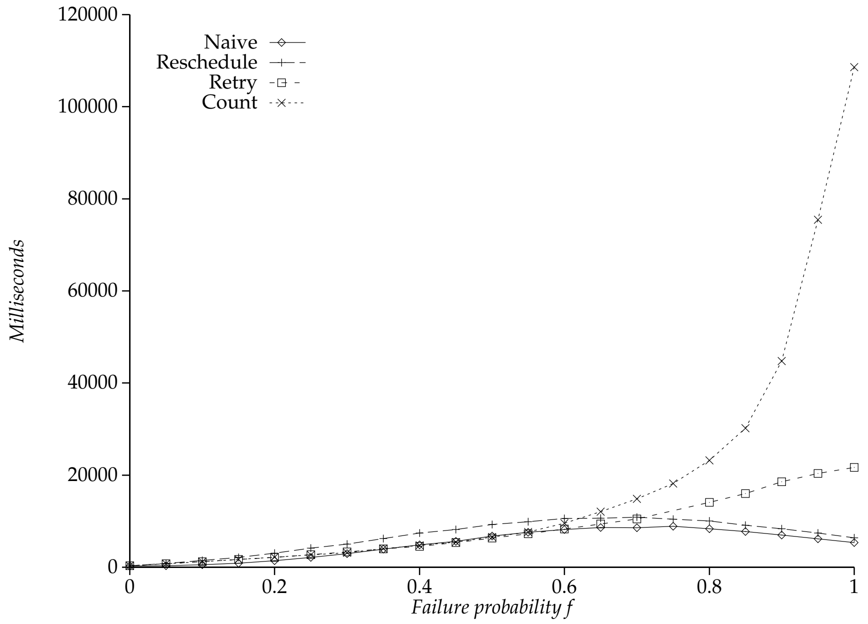

FIGURE 6.11: Total time, varying f, $d = 0.5$.

these experiments.

Figure 6.11 shows the communication latency required for all operations as f is varied. **Naive** requires the least time, as expected. At low failure probabilities, **reschedule** requires more time than the other protocols, but at high failure probabilities it spends no time retrying failed messages and so can complete – presumably with failure – in little more time than **naive**. At high failure probabilities, the **retry** protocol requires one-fifth the latency of the **count** protocol, since it sends only one message to the most distant host.

Figure 6.12 shows the overall number of messages sent by each protocol, using distributions derived from the 125-host data set. 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 protocol will have to send a message to all replicas. **Retry** sends more messages than **reschedule**, since it will retry messages that fail. This becomes increasingly important as the probability of failure increases. **Count** sends slightly fewer messages than **retry**,

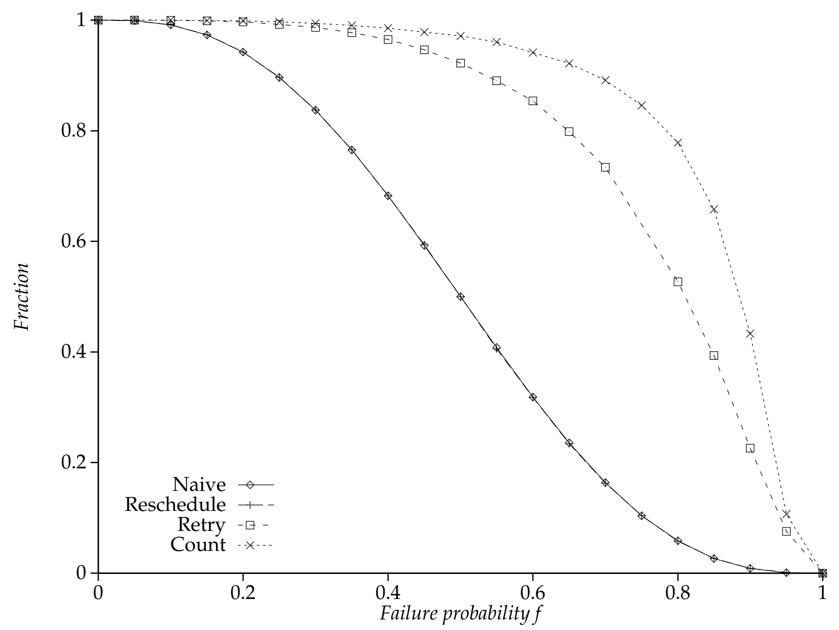

FIGURE 6.10: Success fraction, varying f, $d = 0.5$.

experiment determined message latency. Message failure was once again assumed independent, and occurred with a fixed probability f. The performance of all four protocols was sampled at values of f ranging from 0 (complete success) through 1 (complete message failure). Values of f in the range 0.2 to 0.3 are similar to the behavior of messages in the 125-host sample.

The results of this experiment are summarized in Figures 6.10, 6.11, and 6.12. As expected, figure 6.10 shows that **count** can be expected to successfully gather a reply count of responses more often than the other protocols, and that **retry** will succeed less often than **count**. Both these protocols 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 by Pâris [Pâris86]. That study 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 connections these messages were routed through. Nonetheless, I was encouraged by previous work on the effect of different distributions on the validity of this kind of simulation [Carroll89] and decided to proceed with exponentials.

While the results of the third set of experiments were not identical to those of the first, they were sufficiently close to those of the trace-based simulations to warrant confidence that I could proceed with the fourth set of experiments. The most significant abstraction I made in this set of experiments was to assume that message failures were independent events. In the last chapter I presented experimental results showing that they are not, in the Internet. Independent message failure increases the probability that each protocol would meet its reply count. This assumption makes it less likely that the **retry** and **count** protocols have to retry many times before successfully sending a message. This decreases both the number of messages and amount of time required for these protocols to complete an operation. Nevertheless, I found that the results all showed the same *relative* performance among the protocols, and the results for all operations were within about 20% of the expected values for latency and messages. These simulations thus give a preliminary look at the relative behavior of quorum multicast protocols.

6.5 Effect of failure probability

The fourth set of experiments examined the performance of all four protocols under different failure conditions. The Internet 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 becomes essentially one. This simulation allowed me to evaluate the quorum multicast performance under these worst-case conditions.

In this fourth set of experiments I fixed the delay parameter d at 0.5, because it was close to neither extreme. The synthetic communication latency distributions for the last

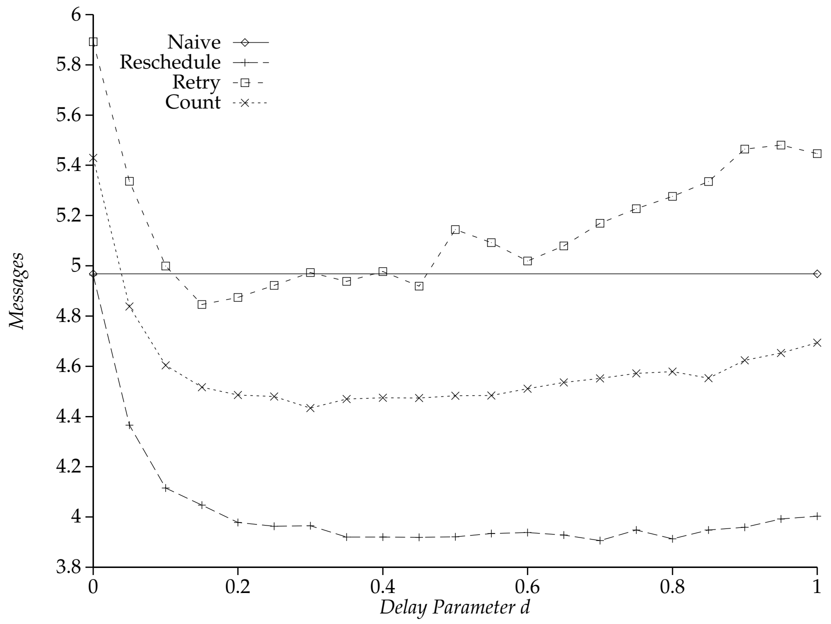

FIGURE 6.9: Messages for all operations, using maximum timeout.

communication traces.

Synthetic distributions are necessary for simulating network conditions that cannot be reproduced on a real internetwork. For example, in the next section I consider the effects of varying probabilities of message failure. I could not arbitrarily and reliably disrupt Internet communication to obtain these data, so I used synthetic distributions instead.

I derived a distribution for the communication latency and message failure for each host from the 125-host traces. For message failure, I assumed each message was independent, and had a uniform probability of being received, equal to the overall message success measured in the traces. For communication latency, I fit exponential curves to the observed communication latency distributions by finding the minimum latency x_{min}, and using the maximum likelihood estimator $\hat{\lambda} = \mathrm{E}[X - x_{\text{min}}]^{-1}$ to define an exponential probability density function for the communication latency a_r. I found that the exponential fit well for most of the hosts, but failed quite badly for some, particularly the sites in Finland, Norway, and France. I believe that this failure is related to the nature of the transoceanic

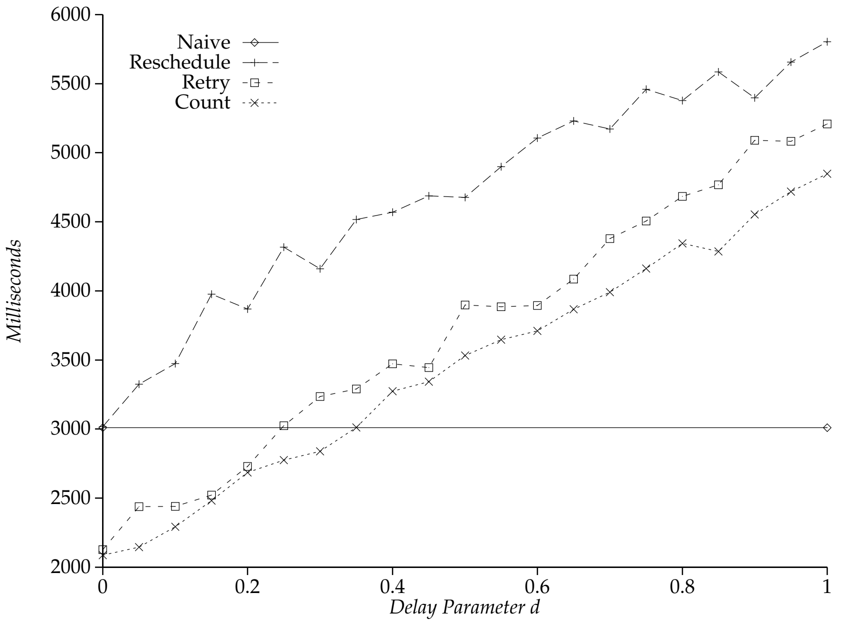

FIGURE 6.8: Communication latency for all operations, using maximum timeout.

a minimum as d increases from zero, while the moving-average curves drop more slowly. The **retry** protocol exhibits the greatest difference, using only a fraction of a message more than **naive**. This occurs because the difference between the maximum latencies of the closest and farthest replicas is much less than the difference between their moving averages. Since there is less of a disparity between nearby and distant replicas, the **retry** protocol has less time to retry messages to nearby replicas and so behaves better.

Generally, longer timeouts appear detrimental to quorum multicast protocols, and the moving average technique is generally more appropriate.

6.4 Effect of synthetic distributions

I conducted a third set of experiments to validate the accuracy of using derived distributions, rather than traces, to drive protocol simulations. The only difference between these experiments and previous ones is that synthetic distributions were substituted for predominate. However, the large number of messages sent by **retry** using the 125-host data set make that the least attractive quorum multicast protocols as far as messages are concerned. This is a very different result from that obtained with the 24-host data set, where the three new protocols exhibited only slight differences overall.

6.2.4 Choosing between the protocols

One can choose between the protocols, and tune the protocols, 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 protocols use fewer messages, at the cost of somewhat greater latency. If the probability of message failure is somewhat higher, **count** provides the lowest latency while **reschedule** requires the fewest messages.

6.3 Effect of timeout period

The second set of simulation experiments allowed me to determine how sensitive the protocols are to the failure timeout period. In this experiment the timeout period for each host was set to the latency of the slowest message to that host. This assures than no messages would be rejected because they arrived too late. However, this usually produced a timeout period between two and twenty times as long as that produced by the moving-average technique (§5.5).

Figures 6.8 and 6.9 show the overall latency and messages, respectively, required by each protocol for the 125-host data sets. Overall latencies are about twice as long as the latencies observed using moving averages (see Figure 6.4). Further, the differences between the **retry** and **count** protocols are much less pronounced in this simulation. The message count for **naive** is five, as always, but the other protocols use fewer messages overall than with moving average timeouts. The **reschedule** and **count** protocols both drop rapidly to

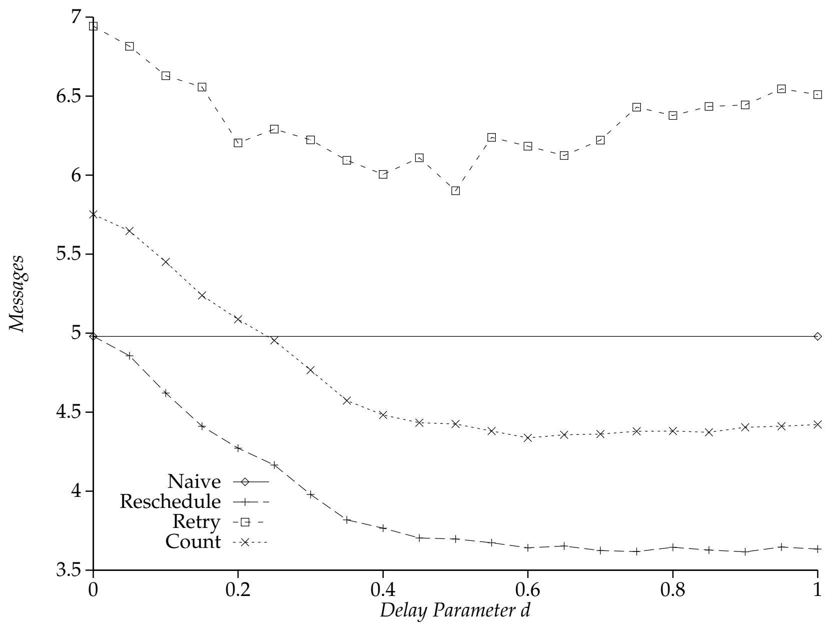

FIGURE 6.7: Messages for all operations.

the two is dramatic. Using the 125-host set of traces, the **retry** protocol sends between three and six times as many messages as the other protocols, while **count** usually sends only one additional message. The difference is due to the extra control that **count** exercises over sending messages – no replica will be tried more than a certain number of times, while **retry** may try nearby replicas a great many times, as was discussed in §4.5. **Retry** exhibits this behavior when the most distant replica has an average communication latency much longer than that of nearby replicas. If a nearby replica has failed, the protocol will have time to retry it many times while waiting for a response (or timeout) from the most distant replica.

When the simulations were run using the 24-host data sets, the behavior was somewhat different. In that data set, which exhibited fewer message failures, the **count** and **retry** protocols required *fewer* messages than **naive**.

Figure 6.7 shows the overall number of messages sent by each protocol. Once again, since the probability of meeting the reply count is high, the values for successful operations

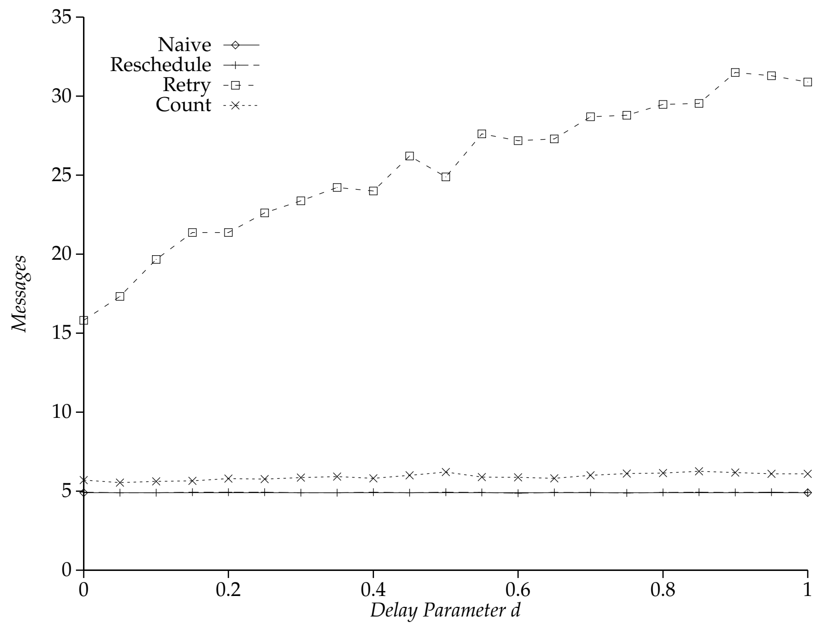

FIGURE 6.6: Messages for failed operations.

d. Since **retry** must try every host once, it will require at least as many messages as **naive**. If messages to nearby hosts fail, they may be retried thus adding to the average message count.

The **count** protocol uses more messages than **naive** for low values of d, but requires about 10% fewer messages for values of $d \geq 0.5$. When d is set to a low value, **count** behaves much like **retry**, sending messages to all replicas and occasionally resending when a message fails. When d is set to a higher value, the protocol behaves like **reschedule**, except that it resends (on the average) about one message because of failure.

The four protocols perform quite differently when they are unable to obtain a reply count of responses. Figure 6.6 shows the number of messages required. **Naive** is used as the baseline measure, requiring exactly five messages no matter what. Reschedule also requires exactly five messages, since it will generally send to all replicas before it can determine that the operation has failed. The **retry** and **count** algorithms will generally send more than five messages before they can declare failure, but the difference between

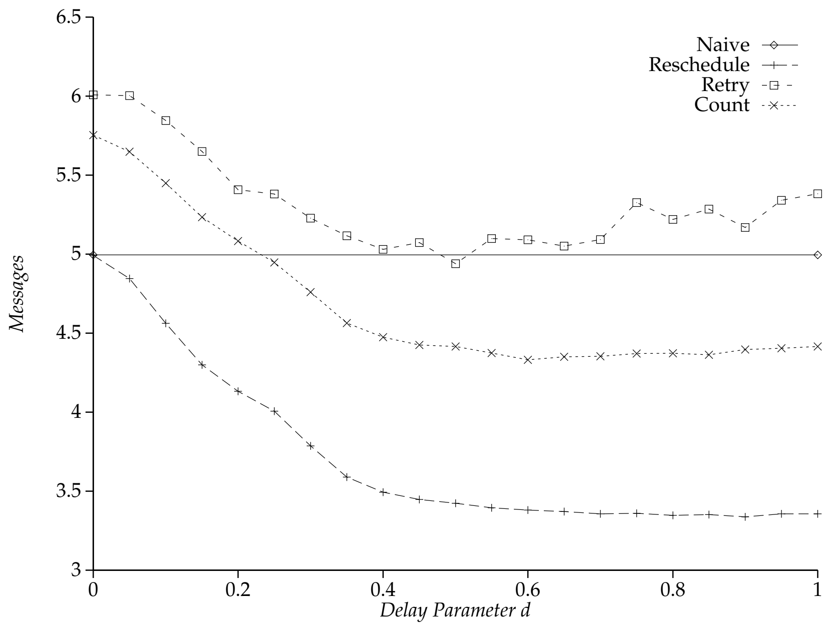

FIGURE 6.5: Messages for successful operations.

6.2.3 Messages

Figure 6.5 shows the number of messages sent for successful operations. **Naive** provides a baseline of 5 messages, since it always sends one message to each host. The **reschedule** algorithm sends fewer messages than **naive**, except at $d = 0$ when the two algorithms are identical. This savings happens because **reschedule** avoids sending messages to distant replicas. When $d \geq 0.5$, the number of messages drops to a steady value of less than 3.5. The message count drops as d increases because small values of d make it likely that an extra message will be sent to a distant replica before some nearby replica can respond. When the delay timeout period – d times the message failured timeout period T_r – exceeds the expected communication latency, this effect should be negligible. Since $d = 1$ corresponds to the message failure timeout period T_r, and this period is set to approximately three times the expected communication latency, the message count decreases to a minimum at a value of d slightly greater than one-third. As the variance of communication latencies decreases, the closer to $d = 1/3$ this point will be.

The **retry** protocol uses at least as many messages as any other protocol at all values of

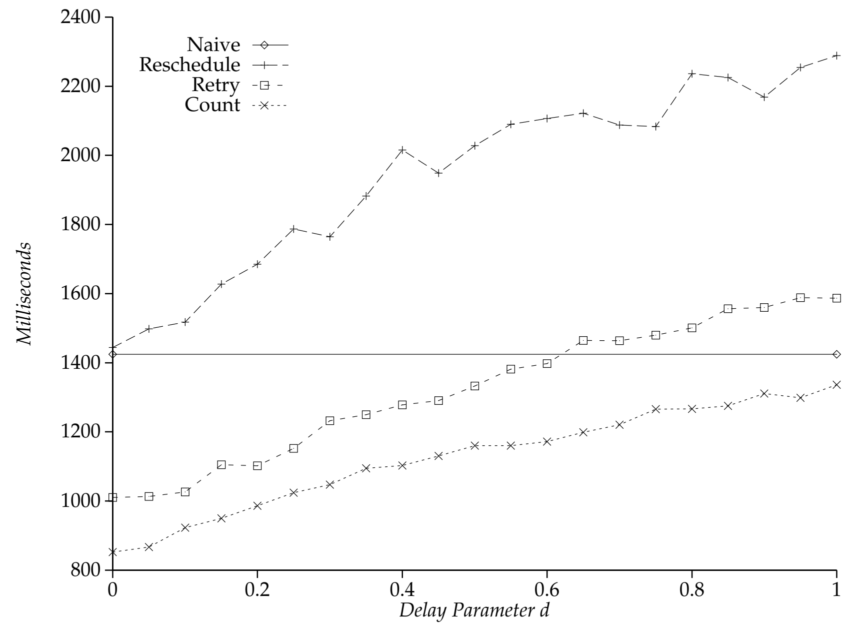

FIGURE 6.4: Communication latency for all operations.

replicas before declaring failure, delaying messages to distant replicas slows down failed operations approximately linearly.

The **count** protocol performs much better than any of the other three protocols when a reply count of responses cannot be obtained. **Count** avoids the problem of having to communicate with the most distant replica, since it can stop when sufficient nearby replicas have failed.

Figure 6.4 shows the overall latency for each protocol. Since the probability of meeting the reply count is quite high, 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 the three quorum multicast protocols, consistently faster even than **naive**. **Reschedule** has the highest latency of the three for all values of d. **Retry** is better than **naive** or **reschedule** for values of d less than about 0.6. This is the reverse of their positions for successful operations. The latency of the quorum multicast algorithms increases approximately linearly as d increases.

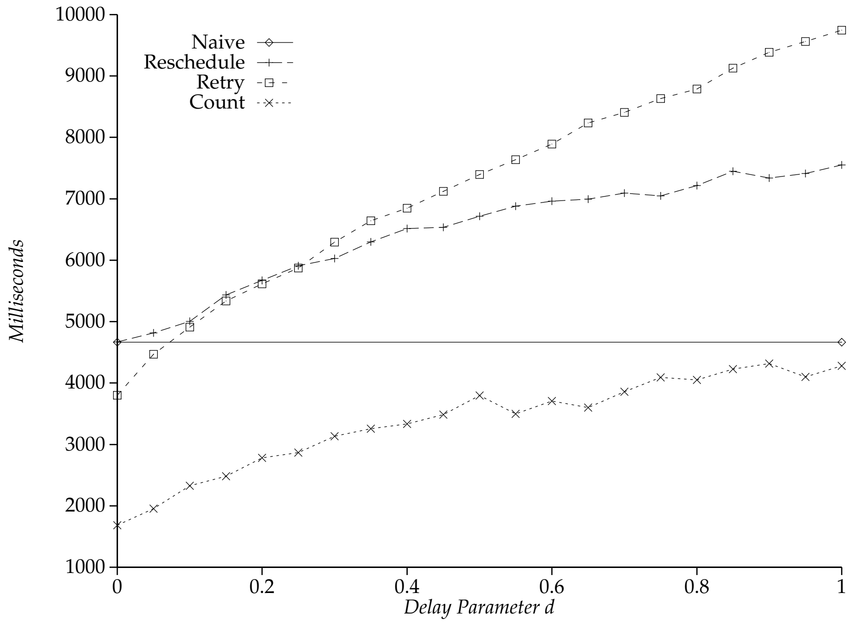

FIGURE 6.3: Communication latency for failed operations.

---

protocol may try communicating with a distant replica several times. Since the probability of message failure is not negligible, this effect translates into longer average latencies. In the 24-host traces, with their lower failure probability, the **reschedule** and **count** protocols performed almost identically.

The performance of the four protocols is quite different when the reply count cannot be met, as is shown in figure 6.3. All four protocols require several seconds to declare failure. While this is quite a long time, failures constitute only a few percent of all operations and the latency is not onerous. **Naive** is the baseline measure, requiring about 4.8 seconds to determine that a reply count cannot be obtained – almost 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 d increases. The **retry** protocol is slightly faster than **naive** or **reschedule** at small values of d, but requires more time than either for larger delay settings. Since **retry** must try all

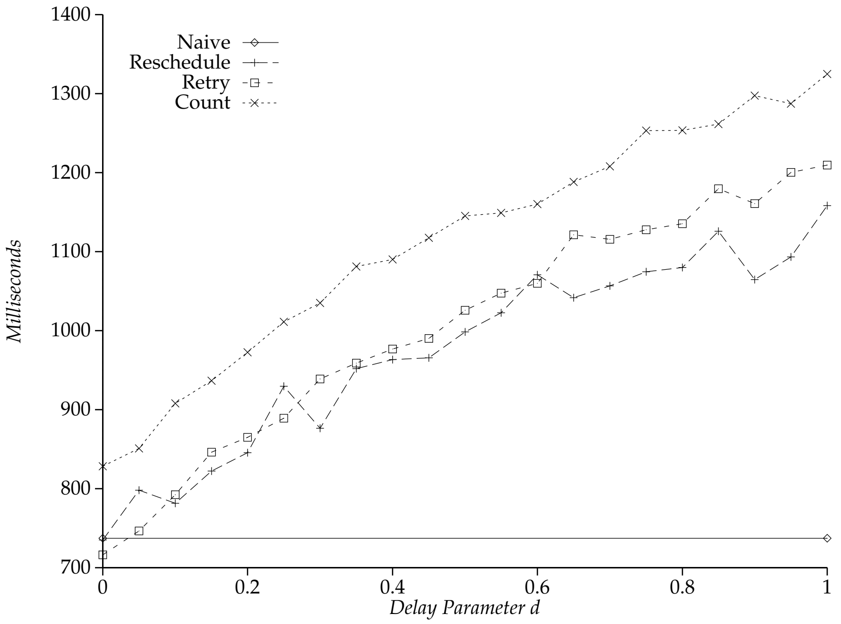

FIGURE 6.2: Communication latency for successful operations.

The results using the smaller set of traces were similar, but produced somewhat higher success fractions for all algorithms. **Naive** and **reschedule** both showed a likelihood of 99.2%; **retry** about 99.8%; and **count** succeeded in excess of 99.9% of the operations attempted. The overall message failure probability in the 24-host set of traces was lower than in the 125-host set, so this difference was expected.

6.2.2 Latency

Figure 6.2 shows the communication latency for successful operations in the 125-host traces. **Naive** is generally the fastest of the four protocols, since it always sends messages to every replica immediately. The other three protocols exhibit similar communication latencies, with **count** taking somewhat 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** protocols must send more than one message to distant replicas to obtain the reply count. **Retry** takes less time than **count** because the latter

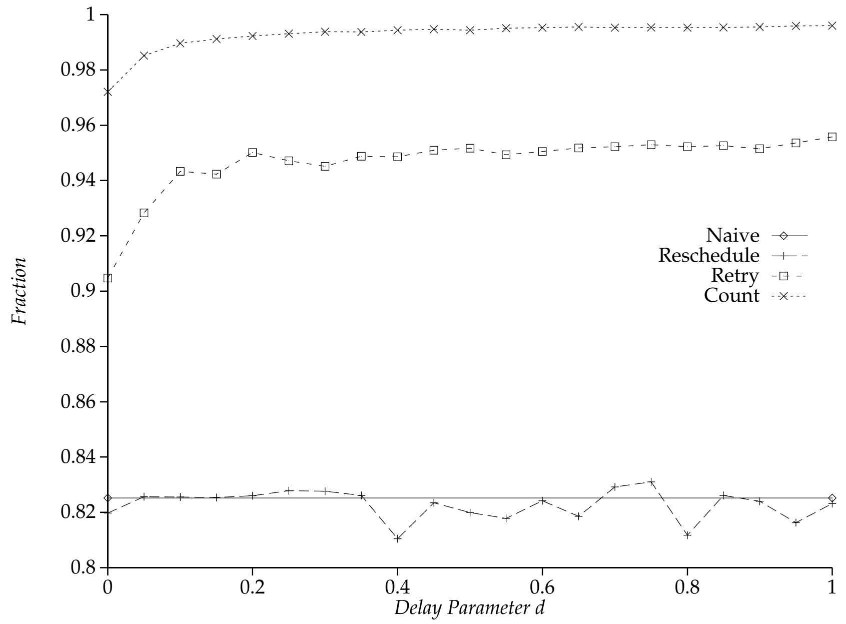

FIGURE 6.1: Success fraction, varying d.

The simulation results for five replicas are summarized in the graphs in figures 6.1–6.7. The results for three and nine replicas are similar. These results were obtained using the large (125-host) set of communication traces.

6.2.1 Operation success

Figure 6.1 shows the fraction of all multicast operations that were successful in meeting the reply count. The **naive** and **reschedule** protocols each exhibited an approximately constant success fraction, at about 82% of all operations. Since these two protocols each attempt to send at most one message to a replica, the delay fraction has no effect on the probability of success. The **retry** protocol, however, retries nearby replicas more times when the delay parameter d is larger, since this allows more time for retries. **Retry** succeeds in approximately 95% of all cases when $d \geq 0.5$ and more than 94% for $d \geq 0.1$, while **count** performs even better. In all, **count** is most likely to succeed, with likelihood in excess of 99%; **retry** succeeds somewhat less often, but substantially more often than **reschedule** and **naive**. This shows that persistent protocols are worthwhile.

imum response time observed for the host. This experiment shows the effectiveness of adapting to changes in network conditions by predicting timeout and expected response time. The third set of experiments substituted synthetic latency and failure distributions for the communication traces. These distributions were derived from the communication traces, and the experiment showed the effect of different distributions on protocol performance. The fourth and final set of experiments also used synthetic distributions, but varied the overall probability of failure so I could determine how the protocols would respond in pathological high-failure situations.

The simulations have limits that must be understood before accepting the performance results they provide. First, they are based upon communication traces obtained with the ping tool, rather than from actual client-to-replica communication using a process-level unreliable datagram mechanism. These traces cannot capture any effects that are specific to quorum multicast protocols, such as congestion at the client or nearby gateways. Timer precision is a further difference between the simulations and a real implementation. The simulations provide timers accurate to a small fraction of a millisecond while many real systems provide much coarser granularity. In the next chapter I will present measurements of an actual implementation that confirm that these limits do not invalidate the simulation results.

6.2 Trace-based simulation results

The first set of simulation experiments allowed me to examine the behavior of each of the four multicast protocols (**naive**, **reschedule**, **retry**, and **count**) under different conditions. In this section I will examine the probability of successfully obtaining the reply count, as well as the number of messages and the latency required for successful, failed, and overall operations. These experiments used measured traces of network behavior to drive the simulation. A moving average of communication latency and failure probability determined the order hosts would be polled and the timeout period used to detect message failure.

Given a randomly-selected set of hosts, the simulation of one operation for each of the 504 sets of samples proceeded as follows. First, the n hosts were selected. These n hosts were then ordered by their predicted response time. Some experiments used the sample mean a_r, which was precomputed as samples were read in. Other experiments used an expectation based on moving averages of communication latency and failure, as we defined in Equation 5.3 (§5.5). Next, one run of each protocol was performed for each of 504 data sets. When the simulation of a protocol sent the ith message to replica r, the ith sample for host r (of the 30 samples in a data set) was used to determine whether the message was received, and if so, how long the communication took.

The iteration was repeated at least 1000 times, until the width of the 95% confidence interval for all values being recorded was less than 5%, or until the simulator had performed 10 000 iterations. The actual widths of the confidence intervals were often much smaller than 5%, particularly for the number of messages sent. Using 125 hosts, there are more than 234 million combinations of five hosts, so there is a reasonable probability that each run was independent of other runs. By selecting n hosts at random, I simulated the random placement of n replicas throughout the Internet.

Message failure events simulated detection of failed messages using timeouts, rather than explicitly simulating the timers. The timeout period for a message to host r was set in some experiments to be the time of the longest-latency reply from that host, which was again precomputed while the trace data were being read. Other experiments used a timeout period based on the moving average of communication latency, as defined in Equation 5.1.

The four sets of experiments allowed me to explore the performance of each protocol under different conditions and assumptions. The first set of experiments used communication traces from our Internet measurements to determine message latency and failure. These experiments used a moving average of latency and failure to set the message failure timeout and to order replicas from closest to farthest. The second set of experiments also used the communication traces, but set the message failure timeout for a host to the max-

I will first discuss in §6.1 the details of the simulators. The next several sections then discuss the simulated results. §6.2 presents overall performance measurements of the quorum multicast protocols, using simulations based on communication traces. §6.3 explores how different timeout periods effect performance, while §6.4 considers the effect of different distributions of communication latency and failure. Finally, §6.5 investigates the effect of various failure probabilities.

6.1 Simulation model

All of the four simulation experiments used a common discrete-event simulator. The simulator program was coded in the **C++** programming language [Ellis90], using locally-written simulation libraries. The functions to determine message failure and latency were the only changes to the program required for the different simulation experiments.

Each run of the simulator produced data points for one particular setting of the number of replicas n, the reply count q, and the delay parameter d. The simulator reported the number of completed operations, the mean communication latency, and the mean number of messages sent, for both successful and unsuccessful operations. Since there were several hundred data points to be determined, I used more than twenty Sparcstation SLC computers in the Baskin Center at UC Santa Cruz to compute the points in parallel. Each simulator run required between ten and forty minutes, so an entire set of simulation runs could be completed in a few hours, covering all four protocols for values of d from 0 to 1 in increments of 0.05 with three different reply counts.

Each simulator run consisted of repeatedly choosing n host traces at random from the hosts in the sample sets, and simulating one operation for each of the sets of ping samples in the trace. I ran the simulator on both sets of traces. The first, smaller, set used the 24-host traces that contained sets of 50 pings taken over a period of 48 hours, yielding 144 sets of samples per host. The second set used the 125-host traces that were taken over a period of 7 days, giving 504 sets of 30 ping samples per host.

Chapter 6

Simulated performance evaluation

In Chapter 4 I presented a family of communication protocols, which I claim can improve the performance of applications that use replicated data. In this chapter and the next I will back up this claim using simulation and measurement techniques.

In this chapter I use discrete-event simulations to analyze performance. Simulations have the advantage of being simple to code and quick to execute. Unfortunately they do not always provide a correct answer to performance questions, so I will back up the simulations with direct measurement in the next chapter.

I conducted four sets of simulation experiments. Each set of experiments consisted of simulating the performance of each multicast protocol for three, five, and nine replicas. The first two experiments used the samples of communication latency from the traces of the Internet. These experiments were intended to estimate relative performance information for each protocol, given actual network performance. The third set of experiments used distributions derived from the traces to repeat the simulations performed in the first two experiments. The intent of these experiments was to validate that artificial distributions gave accurate results when compared with measured Internet performance. The fourth and final set of experiments also used derived performance distributions, except that the probability of message failure varied. This experiment was intended to determine the relative performance of the protocols under widely varying probabilities of message failure. I used synthetic performance distributions since these conditions could not be created on the live Internet.

5.6 Summary

I recorded two sets of communication traces on the Internet, one using 24 hosts, the other using 125. These traces sampled the time required for the ping utility to send an ICMP echo packet to each host. Each trace contained several thousand samples per host.

I analyzed these traces to measure communication latency and success rates. Latency distributions are complex – they are obviously not exponential, for example – but appear to have a consistent shape. Message success is more complex. Message failures are definitely not independent events: far too few messages fail singly. More than 70% of all messages fail singly or in a run of two or three failures. If three messages in a row have failed, it is very likely that all the messages in a data set failed, and the host is probably unavailable.

Moving averages of recent latency and failures provide a convenient and reasonably accurate mechanism for predicting near-term future behavior. Moving averages track changes in latency well, and can be used to derive a timeout period for detecting message failure.

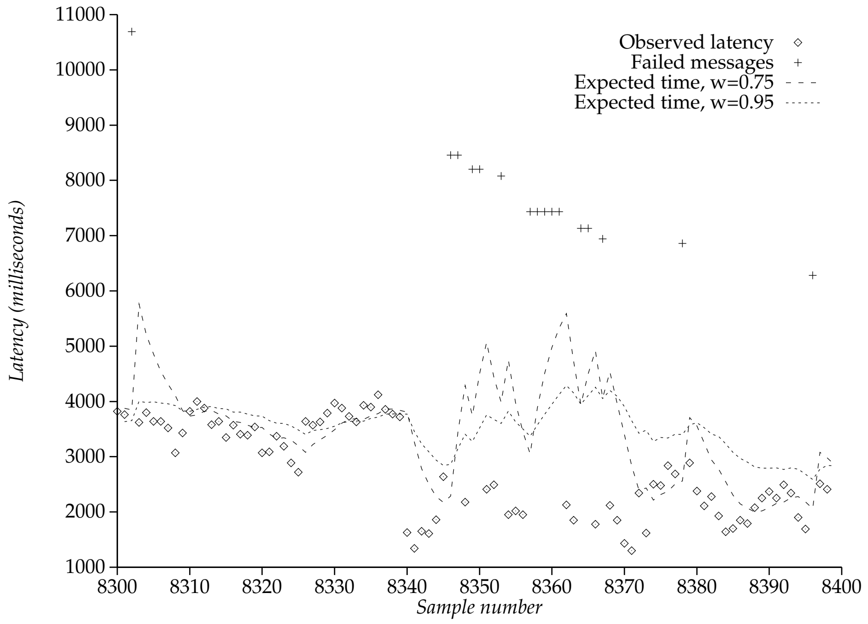

FIGURE 5.19: Expected communication latency for brake.ii.uib.no

The overall latency expectation is useful when a communication protocol is to selectively communicate with the replicas most likely to respond quickly. Given that the communication latency, probability of failure, and timeout period can all be estimated, the overall expected time o_t is the sum of the expected latency and timeout period, weighted by failure probability:

$$o_t = \overline{f_t} a_{95,t} + (1 - \overline{f_t}) \overline{a_t}. \tag{5.3}$$

Figure 5.19 shows a sample of this overall expectation for one of the sample hosts. The latencies shown in this figure for failed messages are the timeout periods for $w = 0.95$. The way the expectation responds to changing conditions is evident in the samples between 8330 and 8350, where the expectation changes from tracking actual latency in a low-failure period to averaging actual latency and time-out period in a high-failure period.

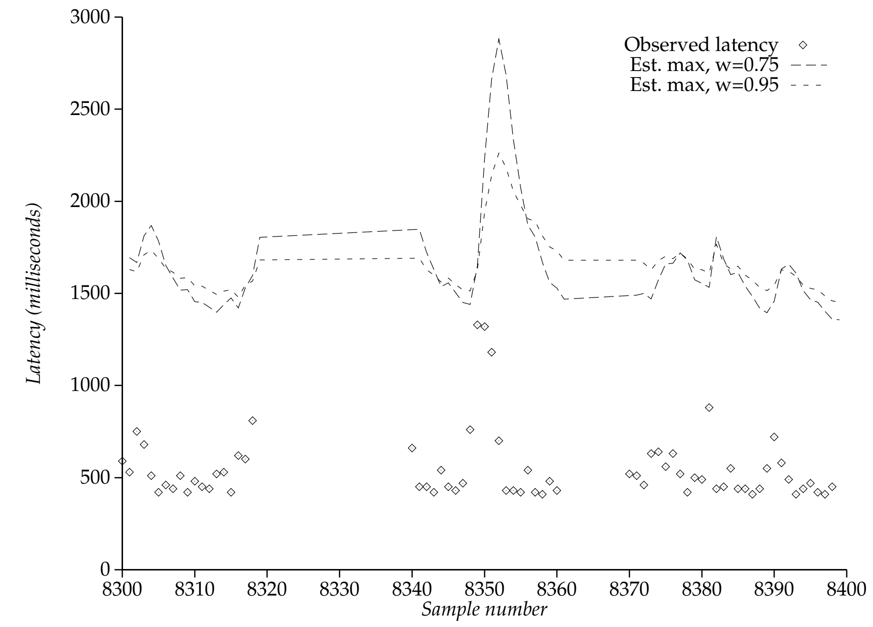

FIGURE 5.17: Sample estimated timeout periods for cana.sci.kun.nl. Overall maximum 19 010 milliseconds.

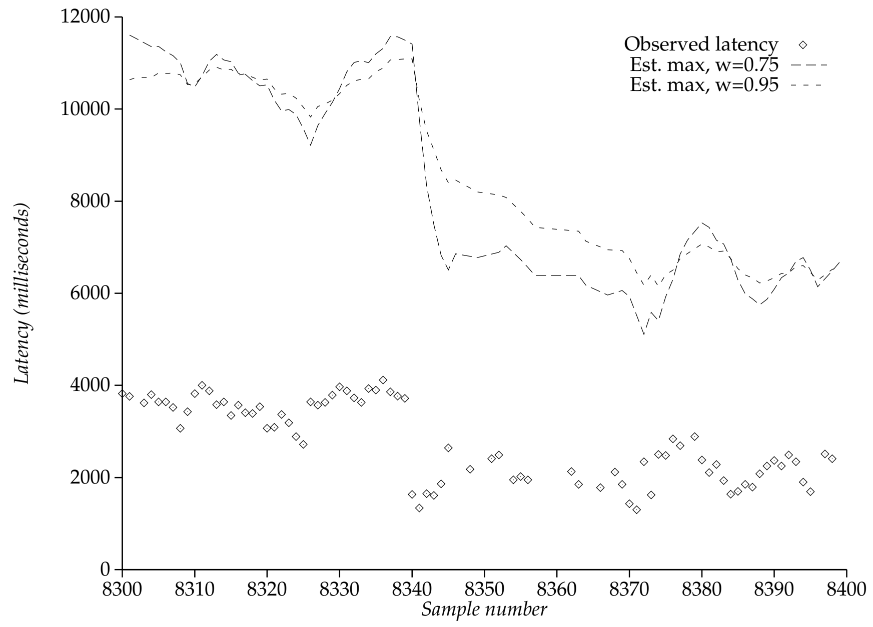

FIGURE 5.18: Sample estimated timeout periods for brake.ii.uib.no. Overall maximum 23 160 milliseconds.

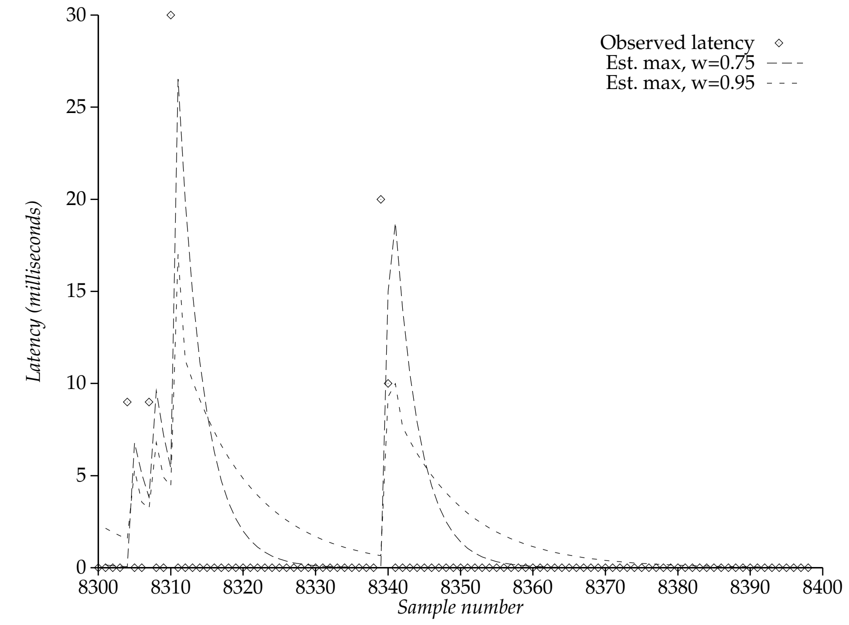

FIGURE 5.15: Sample estimated timeout periods for sequoia.ucsc.edu. Overall maximum 570 milliseconds.

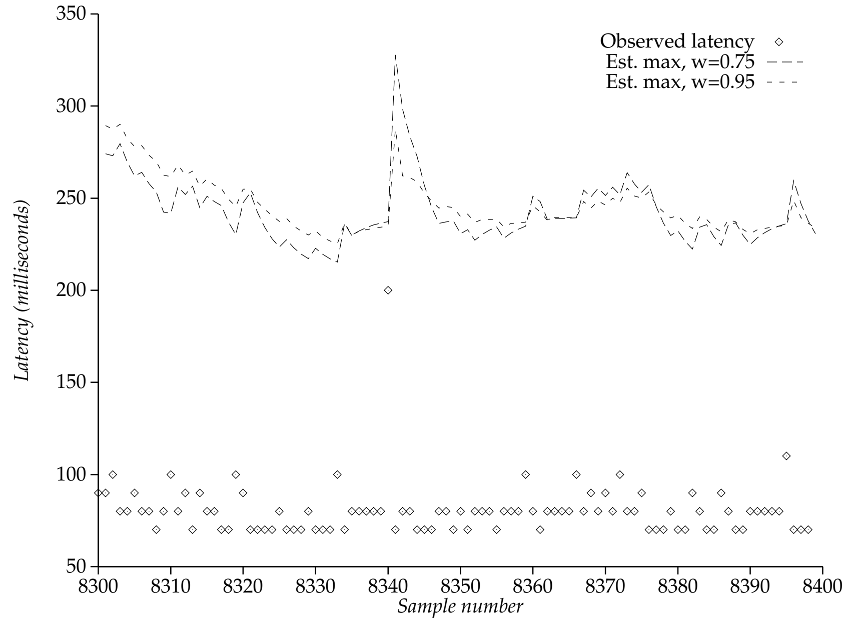

FIGURE 5.16: Sample estimated timeout periods for bromide.chem.utah.edu. Overall maximum 670 milliseconds.

TABLE 5.4: Fraction of replies rejected because of short timeout. 95th percentile used to set timeout. Moving average weight $w = 0.95$.

	Fraction rejected	
	No minimum	40ms minimum
sequoia.ucsc.edu	3.17%	1.20%
bromide.chem.utah.edu	0.27%	0.27%
cana.sci.kun.edu	0.54%	0.54%
brake.ii.uib.no	0.19%	0.19%

long, since a protocol must wait for that period before a message can be determined to have failed.

I examined the the traces using the 95th percentile estimator. The fraction of replies that were returned later than the estimated timeout period are shown in Table 5.4. It is very small for all but the nearby site, for which it was 3.17%. As seen in Figure 5.15, the estimated timeout period for this host often goes to zero. This occurs because the resolution of the trace samples is only 10 milliseconds, so the actual latency was almost always small enough to be recorded as zero. The fraction rejected dropped to 1.20% of successful replies upon applying a minimum timeout period of 40 milliseconds. While this is still high, from the distribution in Figure 5.7 it can be seen that the cutoff would have to be set to several hundred milliseconds to obtain less than 0.05% rejection fractions from this host. The average latency and variance are small enough that cutoff values more than about 40 milliseconds make no sense.

Moving averages can also be applied to the estimation of failure probability. Given a sequence of samples $F_i \in \{0, 1\}$, the moving average $\overline{f_t} = \sum_{i=0}^{t} w^{t-i} f_i$ gives an approximation of the likelihood of failure. I have found that a large weight, that is, a value of w near one, appears to work well.

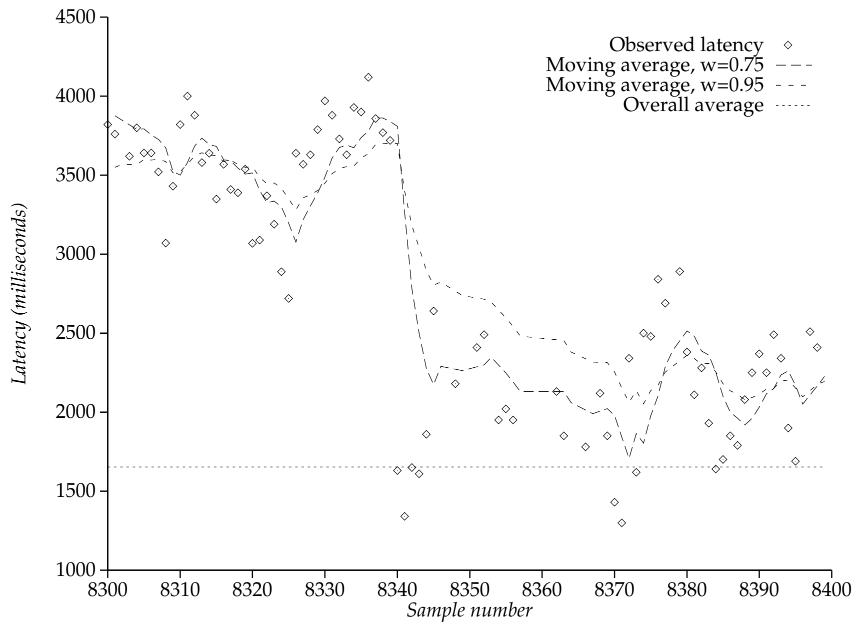

FIGURE 5.14: Sample moving averages of latency for brake.ii.uib.no.

mated for a sample using $\hat{\lambda} = 1/\overline{a_t}$, the maximum likelihood estimator [Rice88, Chap. 8]. The exponential is used to estimate a reasonable upper bound for message latency by taking, say, its 95th percentile. The rth percentile $a_{r,t}$ of the exponential approximating the predicted latency can be computed as

$$a_{r,t} = \frac{-\ln(1 - 0.01r)}{\hat{\lambda}} = -\ln(1 - 0.01r)\overline{a_t}. \tag{5.1}$$

For the 95th percentile, this leads to the formula

$$a_{95,t} \approx 2.995732\,\overline{a_t}. \tag{5.2}$$

Figures 5.15 through 5.18 show this curve for selected portions of four traces.

If this setting is to be useful for the timeout period, it must not be too short. When it is, the protocol will time out before a reply is received and either send another message or declare the host unavailable, even though the reply was on its way. The expense of retrying or declaring failure is likely to be unacceptable, so any timeout setting must not reject too many valid messages. On the other hand, the timeout period must not be too

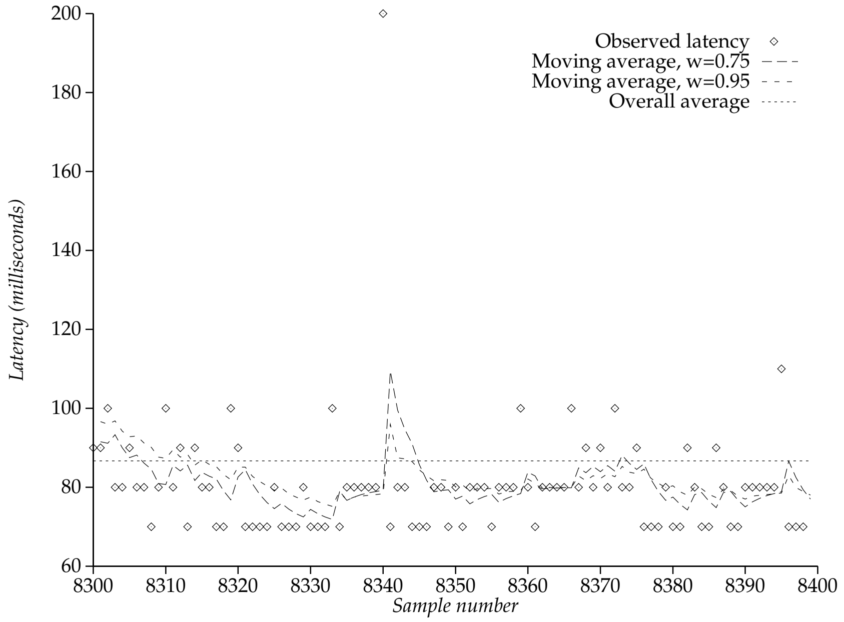

FIGURE 5.12: Sample moving averages of latency for bromide.chem.utah.edu.

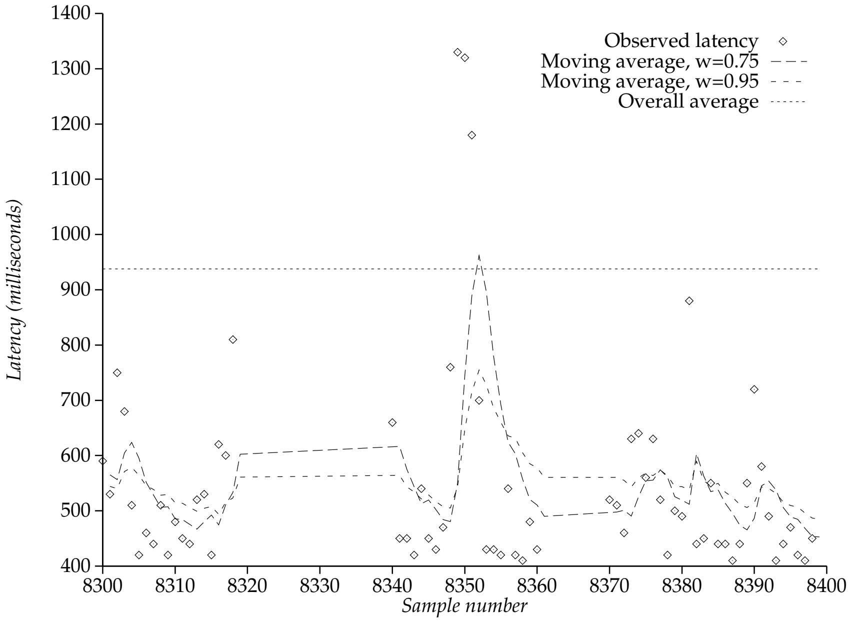

FIGURE 5.13: Sample moving averages of latency for cana.sci.kun.nl.

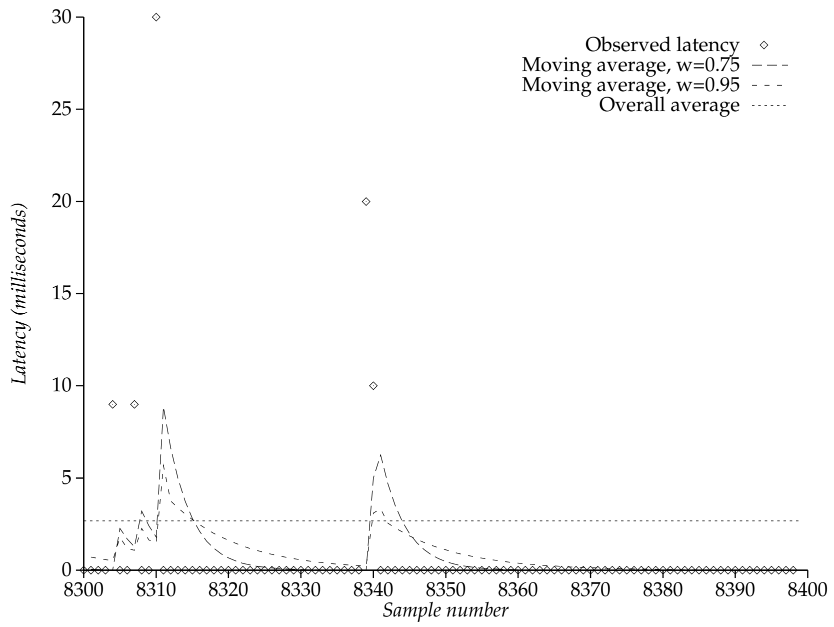

FIGURE 5.11: Sample moving averages of latency for sequoia.ucsc.edu.

Figure 5.11 illustrates how the moving averages behave. This figure shows 100 samples of communication latency from a trace of communication with sequoia.ucsc.edu. Two curves show the effects of different weighting values. As long-latency samples are observed, the moving average rises, then decays back to a lower value as latency returns to normal. Figures 5.12, 5.13, and 5.14 show similar curves for the other three hosts we considered in the last section. The moving average can be seen to track changes in behavior. The flat sections in Figure 5.13 (for cana.sci.kun.nl) and Figure 5.14 (for brake.ii.uib.no) represent failed samples. These are ignored when calculating moving averages.

An appropriate timeout period for determining when messages have failed can be based on the predicted communication latency. The moving average at time t is an estimate of the mean of the latency distribution for the next sample. While it is obvious from the latency distributions for cana.sci.kun.nl (Figure 5.9) and brake.ii.uib.no (Figure 5.10) that message latencies are not quite exponentially distributed, I treat them as if they are to calculate a timeout period. The parameter λ for the exponential distribution can be estifailure rate from recent past behavior.

There are two reasons why it is useful for a protocol to predict behavior. One reason is that it should generally communicate with the site that will respond most quickly. As will become evident in the performance analysis in later chapters, a determination of replica proximity can yield significant performance improvements. A protocol must also determine how long to wait for a reply to a message before deciding that the message has failed. Communication protocol performance is known to be sensitive to this timeout period. Protocols should adapt these predications to changes in network topology and load.

The performance predictions can be based on *a priori* information, such as the topology of the network, or on observed behavior, such as past message latency. I have concentrated on predictive methods based on past behavior in this thesis, though *a priori* methods are also important. Cheriton [Cheriton89] claims that a source routing internetwork protocol, unlike the IP protocol currently used in the Internet, will enable applications to accurately predict communication behavior from performance information maintained for potential routes.

I have chosen to use a *moving average* of recent behavior as the predictor of future behavior. If there is a series of observations

$$a_0, a_1, a_2, \ldots, a_n$$

the moving average $\overline{a_t}$ at sample t is

$$\overline{a_t} = \sum_{i=0}^{t} w^{t-i} a_i,$$

where $0 < w < 1$ is the *weighting* value. This can also be written as the recurrence

$$\overline{a_t} = w\overline{a_{t-1}} + (1 - w)a_t.$$

The effect of a moving average is that recent events have more weight than earlier events, and the weight of a sample decreases exponentially with time. The larger the weighting value w, the more slowly the moving average reacts to changes in the samples.

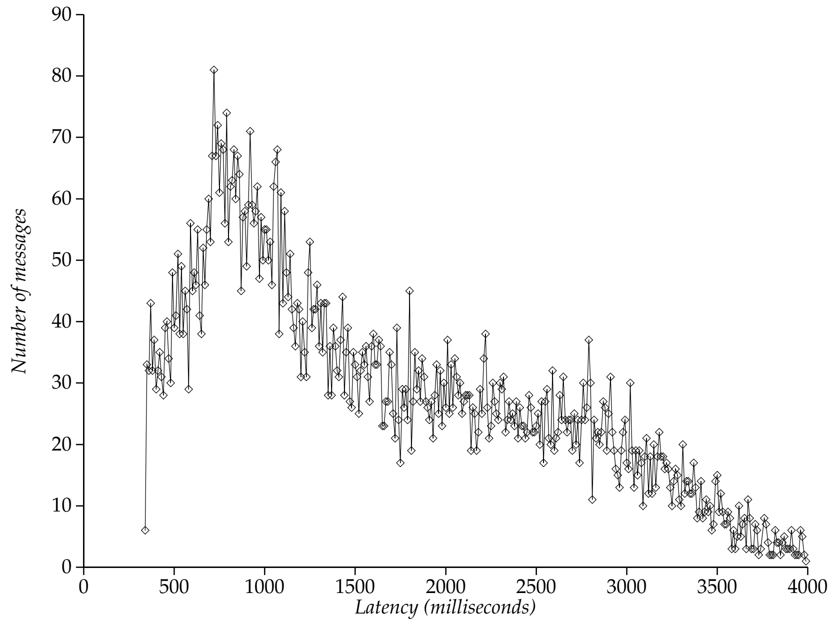

FIGURE 5.10: Distribution of communication latency for brake.ii.uib.no. Average latency 1653 milliseconds.

The distributions for the hosts cana.sci.kun.nl (Figure 5.9), a site in the Netherlands, and brake.ii.uib.no (Figure 5.10), a site in Norway, illustrate the range of distributions observed for overseas connections. The distribution for the Dutch host appears not unlike that of a host in North America, with the majority of messages having a small latency, though the variance is quite a bit larger. The Norwegian site exhibits a much more random distribution. I believe that the packets to this host are routed through a satellite channel, which usually causes high variability.

5.5 Predicting expected and maximum latency

The overall average values for communication latency A_r and failure probability F_r may not be good predictors of actual performance. It appears from the samples that failures cluster. Upon examining the traces it also appears that the latency of one message is related to the latency of the next. This section examines a way to predict latency and

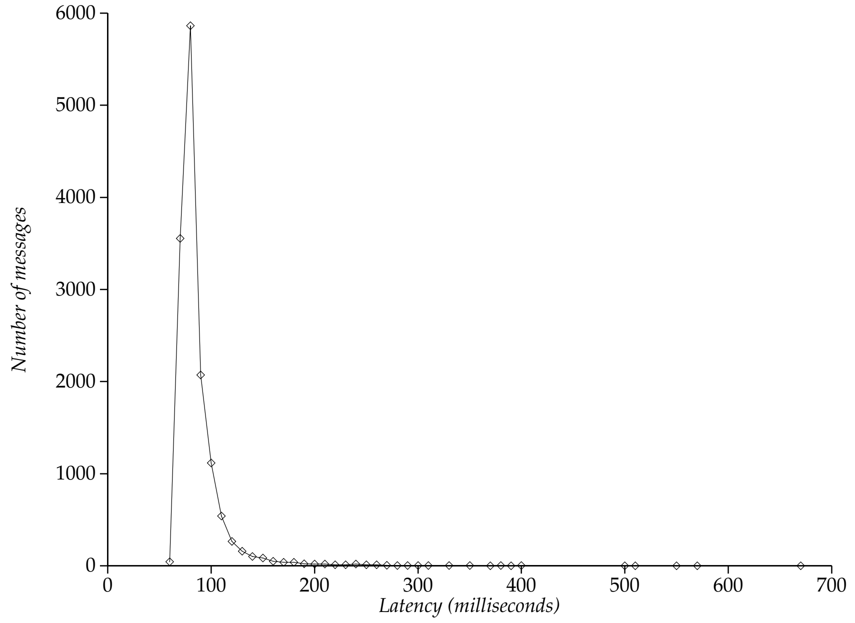

FIGURE 5.8: Distribution of communication latency for bromide.chem.utah.edu. Average latency 87 milliseconds.

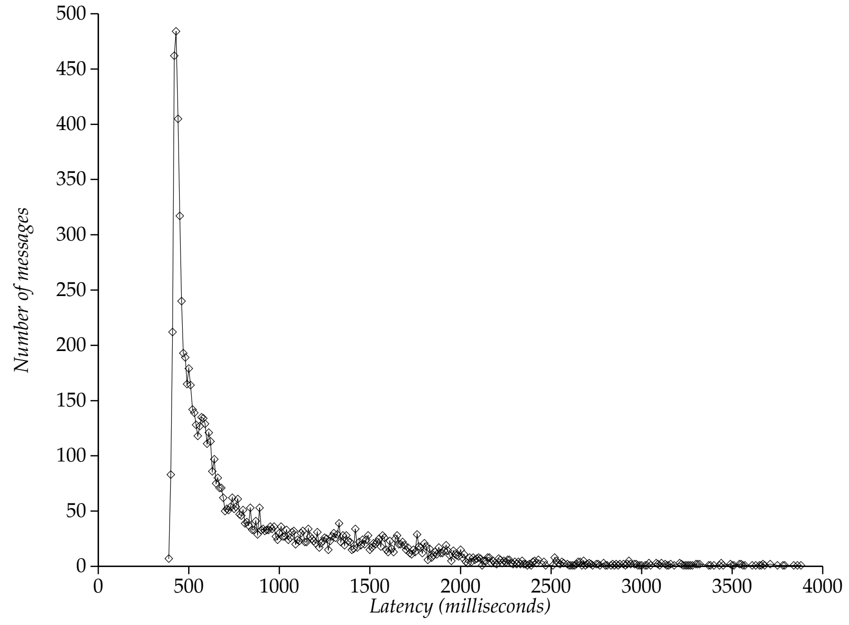

FIGURE 5.9: Distribution of communication latency for cana.sci.kun.nl. Average latency 938 milliseconds.

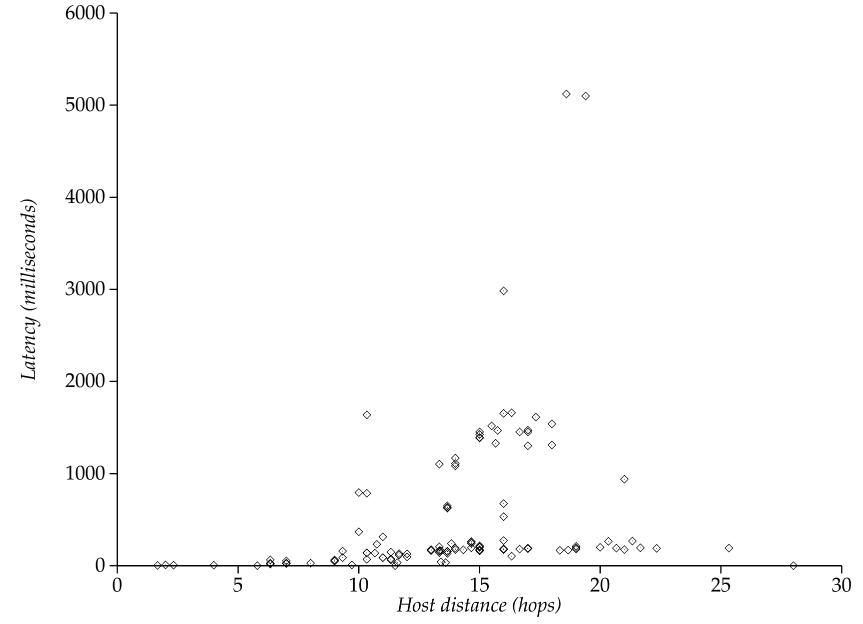

FIGURE 5.6: Message latency versus distance.

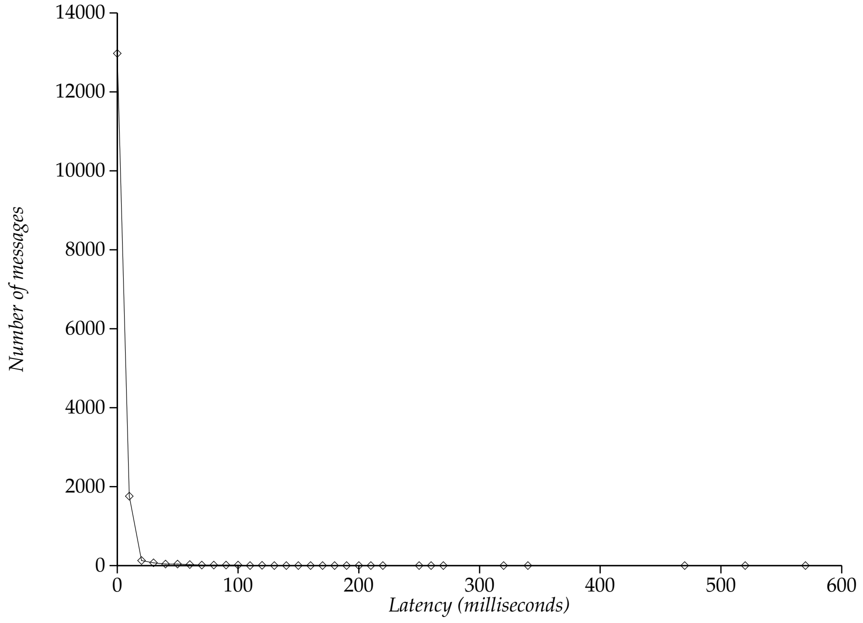

FIGURE 5.7: Distribution of communication latency for sequoia.ucsc.edu.

I also considered how many consecutive messages succeeded. As with failures, successful messages were classified by run length (Figure 5.5.) I found that there were many data sets in which all messages succeeded. However, I also found that there were many runs of a small number of successful messages. Short times between failures are further indication that failures tend to cluster.

5.4.2 Communication latency

Communication latency was the second focus of the measurements. The values were obtained using a Sun 4/20 workstation that has a clock resolution of approximately 10 milliseconds. (This resolution is obvious in Figures 5.11 and 5.12.) The average response latency for hosts in the first experiment is reported in Table 5.1.

As with message failure, I was curious whether the number of gateways traversed in communicating with a host was related to the average latency. Figure 5.6 shows the latency against the distance in hops. It would appear that there may be some relation between the two.

While the average latency may be of interest, its distribution is equally important. Figures 5.7 through 5.10 present four typical distributions. These graphs show histograms of the fraction of messages that fell into 10-millisecond ranges, starting from zero. Most hosts showed a very few short-latency messages, with a sudden peak dropping rapidly back to zero.

The host sequoia.ucsc.edu (Figure 5.7) is at UC Santa Cruz, in the same organization as the host from which the measurements were taken. One gateway machine connects the Ethernets used by either machine. Most response times were sufficiently small that the 10-millisecond sampling resolution is of some concern. This curve is typical of the results observed for hosts on the same or nearby Ethernet segments. The latency distribution for bromide.chem.utah.edu (Figure 5.8) is typical of the distribution observed for hosts in North America. It is similar to that of a nearby host, but shifted toward greater latency.

TABLE 5.2: Fraction of failed messages by size of run (24-host experiment).

Length	Failure fraction (%)		Independent
of run	All failures	Communication	$f = 93.32\%$
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	—	—

TABLE 5.3: Fraction of failed messages by size of run (125-host experiment).

Length	Failure fraction (%)		Independent
of run	All failures	Communication	$f = 80.0\%$
1	7.73	20.63	64.00
2	1.03	2.74	25.60
3	0.26	0.68	7.68
4	0.11	0.29	2.05
5	0.08	0.21	0.51
30	2.09	—	—

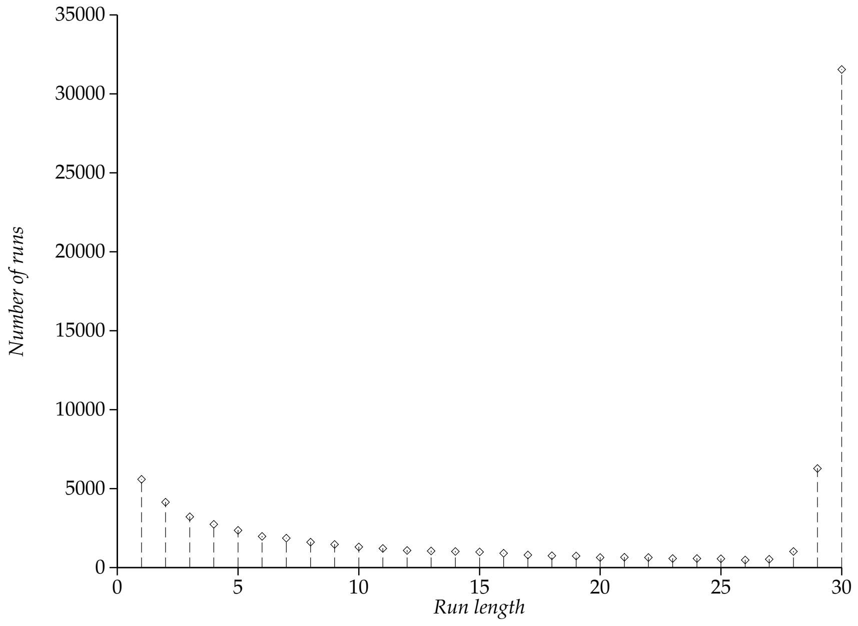

FIGURE 5.5: Lengths of runs of successful messages.

tributed to 1% or more of the number of failed messages. The second column reports the percentage of failed messages that were in runs of each length. I 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 30 or 50, the size of one data set, due to hosts being down for an entire data set. The third column in Tables 5.2 and 5.3 lists the percentage of message failures that were not in runs of length 30. This number approximates the percentage of each run length that is due solely to communication failure, such as congestion, loss of connectivity, or routing loops.

I compared this distribution to what would be obtained if all communication failures were independent. This can be modeled as a Bernoulli trial with parameter f [Trivedi82, Sect. 1.12]. The parameter corresponds to the probability that a message would be successfully acknowledged. The probability $p(n)$ that a failed message would be part of a run of length n is

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

Values of this distribution are shown in the fourth column of Tables 5.2 and 5.3, for the 24-host and 125-host experiments respectively.

If message failures were independent, there would be many more single- or double-message failures than were observed. The difference between the observed behavior and predicted behavior for independent failure leads to the unsurprising conclusion that message failures are not independent events.

There appear to be two behaviors for message failure: short, transient failures due to temporary network conditions, and longer failures due to host or network failure. These data indicate that an internetwork communication protocol would do well to retry failed messages. Further, it appears that most of the advantage can be obtained using a small number of retries. In the 125-host traces, when a sequence of three failures has been observed there is about a 60% probability that the host will be unreachable for the entire set of 30 or 50 polls.

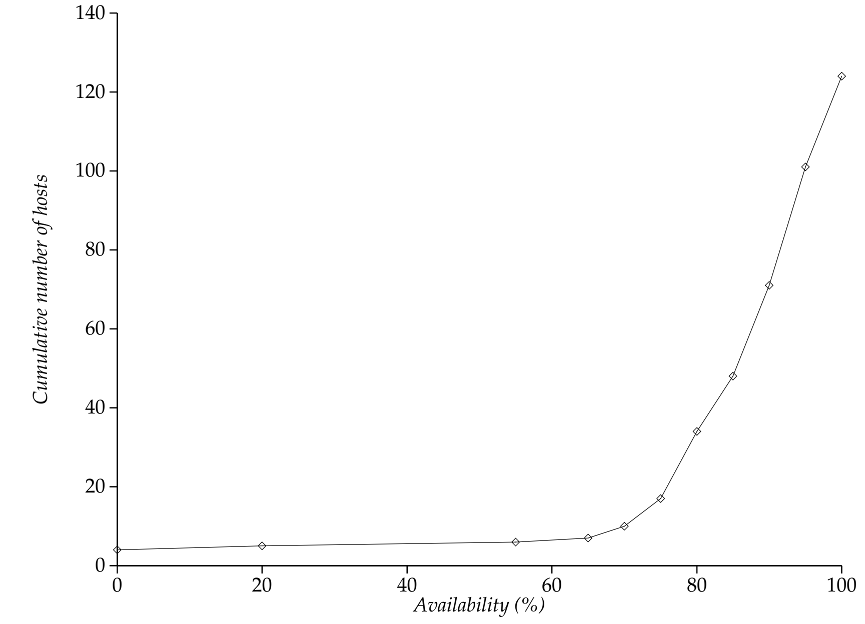

FIGURE 5.3: Approximate host availability.
Average availability 87.5%

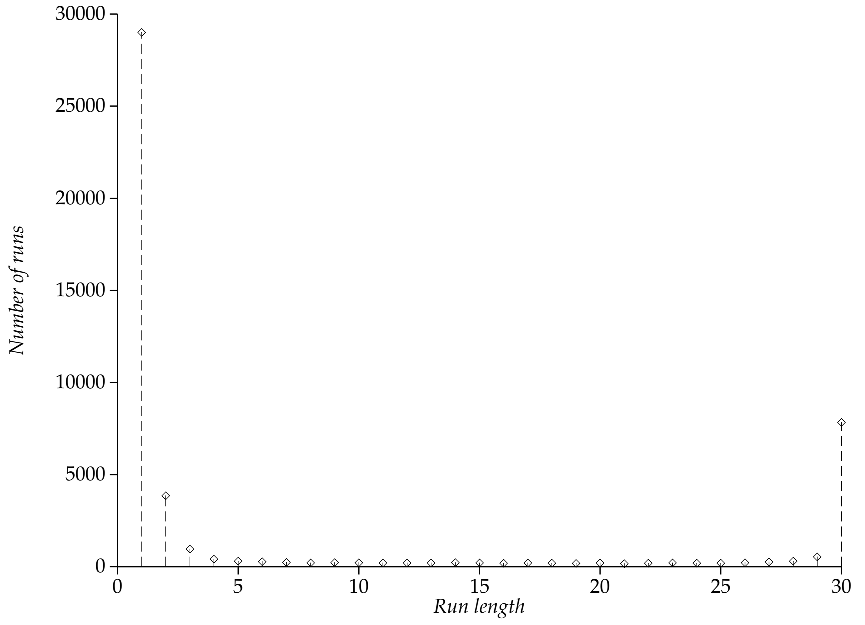

FIGURE 5.4: Lengths of runs of failed messages.

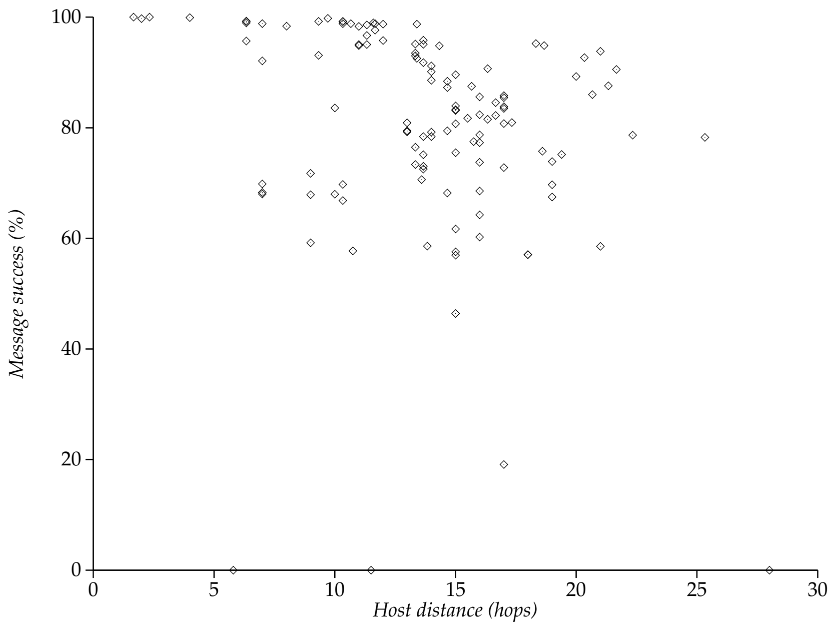

FIGURE 5.2: Message success versus distance.

or four number hops to reach, would have high success fractions. Indeed this appears to be the case. Outside of this local organization the number of hops does not appear to be a good predictor of message success.

Messages can fail for one of two reasons: they are lost in transmission, or the remote host is down. While host availability cannot be determined exactly, a host that does not answer any pings for 30 seconds is likely to have failed. This is not a perfect measure for two reasons: a gateway or link crash would appear to be a host failure, and because a very busy host could also appear to have failed. Bearing these limitations in mind, I computed an estimate of overall host availability as the fraction of 30-ping data sets containing at least one response to the total number of such data sets collected for the host. Figure 5.3 shows the distribution of overall availabilities in the 125-host experiment.

Next, I examined the data sets to determine how long communication failures lasted. Failures were classified by the length of the run of failing messages, as shown in Figure 5.4 and in Tables 5.2 and 5.3. The first column in these tables lists those run lengths that con-

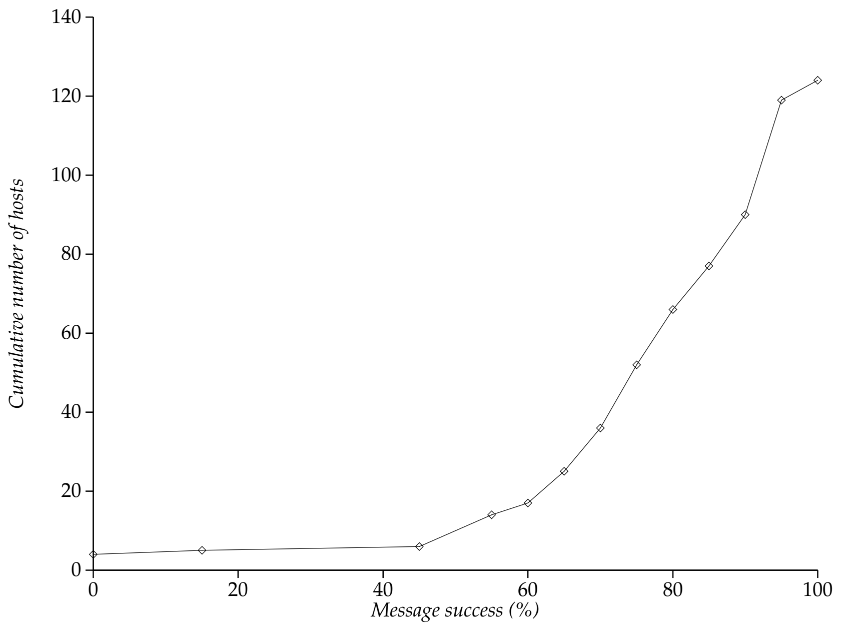

FIGURE 5.1: Overall message success. Average success 80%.

represented a host that appears to have been continuously unavailable for 7 of the 48 hours sampled. Combining this information with data on the reliability of hosts, I conclude that communication will succeed most of the time when a host is functioning.

The data from the 125-host experiment are less encouraging. In this set, four hosts were continuously unavailable for the entire seven days, while some hosts exhibited overall message success rates of less than 50%. All sample hosts selected were known to exist and function a few weeks before I recorded the traces, and it seems unlikely that these four hosts had been deliberately taken out of service in the interval. The mean success rate was 80.0%, as compared to 93.3% for the 24-host experiment. Figure 5.1 shows the fraction of hosts with different overall message success fractions.

I conjectured that the message success rate might be related to the number of gateways that must pass the message. Since I had already measured the number of hops required to reach each of the sample hosts in the 125-host experiment, I plotted overall availability against distance (Figure 5.2.) I had expected that nearby replicas, those that require three from each group was proportional to the size of group, so that the resulting 125 hosts exhibited the same distance properties as the original sample. This step helped to avoid unrealistic clustering of hosts that would bias later results according to the policies or hardware within one particular organization or location.

One problem with this approach is that several installations choose to shield their internal network behind a gateway; information about those subnetworks cannot be obtained by this method. Some large organizations such as sun.com and apple.com contributed only one data point to this study, since only the gateway machines were accessible.

I have been particularly careful in taking these measurements to limit them to those easily obtained by ordinary programs. No measurement made use of special information, such as direct knowledge of the topology of the Internet. In this way I believe that the results in this section can be applied to application software that has no special knowledge of network structure. Concentrating on behavioral measurements also reduces the dependence on current networking technology. As newer networking technology becomes available, it should be possible to easily measure the new communication behavior and re-evaluate the results in this thesis.

5.4 Communication behavior

The traces recorded two pieces of information: whether a message was successfully sent and acknowledged; and if the message succeeded, how long the round-trip took. This section looks in detail at message success and communication latency in these traces.

5.4.1 Message success

Message success is the simplest measure to be obtained from the traces. I first examined the success rate for communicating with each host. The results for the 24-host experiment are reported in Table 5.1. In the this experiment I found that most hosts would respond to a message more than 90% of the time. The one significant exception (andreas.wr.usgs.gov)

TABLE 5.1: Hosts selected for first study.

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

The selection was derived from a list of several thousand Sun 4 systems that came from a study on the reliability of hosts on the Internet [Long91]. The original list was obtained by querying top-level domain servers for the names of hosts at each site and for secondary domain servers, and this process was applied recursively to the entire Internet name-tree. This resulted in over 350 000 hosts, believed to be a substantial fraction of the total Internet. Once lists of hosts were obtained, duplicate names were consolidated and the domain servers were again queried to determine the type and operating system of each host. From these data I was able to obtain a list of several thousand Sun 4 systems.

Next I determined the "distance" of each host, measured as the number of gateways traversed when communicating with it. These data were obtained using the traceroute program, and obtaining at least three samples for each host. The hosts were grouped by distance, and a random selection was made from each group. The number selected

5.3 Methods

I conducted two studies of Internet messaging behavior. The two studies were quite similar; the primary difference was that the second study involved more hosts for a longer period of time. Throughout this thesis I will refer to these as the '24-host' and '125-host' samples.

These studies collected traces of message latency and success for communication with several hosts. Trace records were obtained by polling the remote host from maple.ucsc.edu, a Sun 4/20 workstation in the Concurrent Systems Laboratory (CSL) at UC Santa Cruz. The measurement software was built atop the the ping program, which sends ICMP echo messages. Hosts are expected to respond to ICMP echo messages by returning the message as soon as possible. It polled each host 30 or 50 times (depending on the experiment) every 20 minutes. The first experiment lasted 48 hours; the second lasted one week.

The first measurement study used 24 hosts chosen from those hosts with which CSL systems communicated regularly. The experiment collected 50 samples at one-second intervals every 20 minutes for each host, on a Wednesday and Thursday. This resulted in 7200 samples for each host. The hosts, and a summary of their behaviors, are reported in Table 5.1. One host, andreas.wr.usgs.gov, was unavailable for 7 of the 48 hours sampled; the other hosts appear to have been available the entire time.

The second study was similar to the first, except that it involved more hosts and behavior was traced over an entire week. For this study I selected 125 hosts on the Internet. One set of polls was collected for each host every 20 minutes over a seven-day period. Each set of polls consisted of 30 ICMP echo requests issued at one-second intervals. This resulted in 15 120 samples for each host.

I selected the 125 hosts in a way that I believe will provide a realistic approximation of the behavior of a distributed application on the Internet. My goal was to obtain a set of hosts that was uniformly spread throughout the Internet, avoiding geographic and topological clustering of hosts. I felt that this would approximate the locations where a widely-replicated data object might be placed.

and Sun-2 machines on 10 Mbit and 3 Mbit segments, reporting communication latencies of less than 10 milliseconds. Their study is significant in that they considered the effects of load on the sending and receiving hosts, and on the Ethernet. They did not explicitly study message success or host availability.

Lantz *et al.* [Lantz84] report on a study of local-area distributed applications using the V-system. While most of their performance results concern the drawing rate of a particular graphic data protocol, some of their findings are of general use. One finding was that too short a setting for the length of the packet-loss timeout in TCP could reduce throughput by a factor of two or more. They also found that extremely long timeout values produced only moderately degraded performance compared to shorter timeout values when the receiving host was heavily loaded. They also found that the timeout value should be sensitive to the error rate of the network, and that even on local-area networks there were periods of heavy packet loss.

Long *et al.* [Long91] conducted a study of the reliability of Internet sites. They collected up-time and availability data from several thousand hosts and then used them 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 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, by using 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 for those models ordinarily used as servers the MTTR was approximately 0.5 days.

5.2 Related measurement results

While there has been significant study of specific internetwork protocols, there has been little study of *measured* end-to-end performance of wide-area applications. Since the Internet is shared among many different organizations, direct manipulation of the Internet is not possible. The hardware and software environment changes constantly, making control yet more difficult. Further, it is not generally possible to directly instrument the network to determine performance. Any study of Internet performance must therefore account for the effect of measurement tools on the environment.

Pu *et al.* [Pu90] developed a methodology for measuring the end-to-end performance of Internet applications while studying the response time of applications that use Mach RPC. Their approach is to measure communication performance at as many layers as possible, while measuring end-to-end performance between sample applications. They have measured two applications: the Webster dictionary service between Columbia University and the University of Washington; and the Camelot distributed transaction facility between Columbia and Carnegie-Mellon Universities.

Their measurements used the traceroute [Jacobson90] facility to determine the route between hosts. They found that routes do not change often during periods of several minutes. They also used the ping program, which sends ICMP echo messages, to measure round-trip communication times. Since both the measured applications used Mach RPC, they constructed a simple test application that sent an empty RPC request to a server that "reflected" the message back to the sender. They simultaneously measured performance at the ICMP, RPC, and application layers, and correlated the results. This allowed them to determine the cost of each layer of the system. In addition, they measured end-to-end application performance *without* simultaneously measuring other layers, to establish the degree of interference caused by the other measurements.

Cabrera *et al.* [Cabrera84] studied the end-to-end performance of the TCP and UDP protocols under 4.2BSD Unix. They used a small internetwork, consisting of three Ethernet segments connected by gateways. They measured end-to-end performance between Vax

5.1 Measures and notation

There are three measures of network performance relevant to quorum multicast communication protocols. These measures are the probability of message failure, the communication latency, and the time required to detect message failure. Each is a random variable, sampled each time a message is sent. As a result the distribution and expectation of each of these measures must also be considered.

The probability of message failure reflects the likelihood of messages not being successfully received and acknowledged. If either the message or its reply are lost the communication is considered to have failed. Message success determines whether a replica is considered available or not. Failures can cause a protocol to try communicating with extra replicas, and can increase the overall communication latency. Message success is discussed further in §5.4.1. I use the random boolean variable F_r to indicate whether a message to replica r has failed. The expected failure probability for replica r is $f_r = \mathrm{E}[F_r]$.

Successful send and acknowledgment takes some time A_r, the *communication latency* to replica r. The expectation of A_r is the average communication latency $a_r = \mathrm{E}[A_r]$. The communication latency determines how long a multicast will take to collect a quorum of respondents, assuming no messages fail. The distributions and expectations of communication latency for Internet hosts are considered in §5.4.2.

In practice the distributions of A_r and F_r vary with time. Because of this, the overall expectations a_r and f_r are often not good measures of the current likely values of A_r and F_r. Therefore I consider how to approximate time-varying expectations of these variables in §5.5, and discuss how they can be used to predict future latency. For example, message failure is detected using timeouts that can be derived from this time-varying prediction of communication latency. The timeout period should not be too long, since that means time is wasted detecting a failure; nor should the timeout period be too short, since this could declare failure just before a message arrived, perhaps causing a retransmission.

Chapter 5

Measurement of the Internet

Those who would develop and evaluate efficient communication protocols for wide-area replication must first understand the internetwork environment where the protocols will be used. To this end I have made a number of measurements of the Internet. This section presents a methodology for measuring the Internet and the specific results I obtained for communication failure and latency. In addition, I will consider the problem of predicting the expected communication latency for a host.

The first section defines the measurements I have taken. The second section surveys some related studies. This is followed by a section on the goals and the methods I used to evaluate network behavior.

The penultimate section details the communication behavior of the Internet. To determine this behavior, I obtained measurements of the communication latency and message failure rate between a host at UC Santa Cruz and many other hosts on the Internet, as well as topology information for the portion of the Internet connecting the hosts.

The final section of this chapter takes up the problem of predicting communication latency. This prediction is important for two reasons: it can be used to determine which replicas are nearby and which are more distant; and the message failure timeout can be set from it. If the expected communication latencies are properly determined, the "nearness" and failure timeout periods will track network status changes.

a few times is usually of little value, since communication failures rarely lasted more than two or three messages. The simulator uses an arbitrary limit of $l = 5$.

The **retry** and **count** protocols are both parameterized on the same tuning value d as the **reschedule** protocol. In addition, both protocols use a backoff function to determine how to delay retries to a replica, and the **count** protocol uses a retry limit value. While this thesis examines how each protocol behaves as d varies, I have not attempted to examine the effects of different backoff functions and different retry limits.

Chapters 6 and 7 present performance comparisons of the four protocols described in this chapter.

collision-handling techniques used in the Ethernet [Metcalfe76]). The delay helps to avoid sending vast numbers of messages to a nearby replica that has failed.

The **retry** protocol terminates with failure when it receives a reply or a timeout for replica n, the most distant replica, and the reply count has not yet been met. By then all replicas should have been tried at least once if the timeout periods are increasing by replica, that is, if $T_r \leq T_{r+1}$. If this assumption does not hold, the protocol 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 $i = n$.

This protocol has a variable persistence. Nearby replicas may be retried many times before a distant replica can reply. In the simulator, which doubles the delay after each message, the expected number of retries for a replica r is bounded by $\log_2 (T_n/a_r)$, where a_r is the *expected communication latency* of the rth replica, and T_n is the failure timeout period for the most distant replica. I will discuss these measures in the next chapter.

4.6 The count protocol

The **count** protocol is similar to **retry**, except that it has a fixed persistence. It maintains a counter for each replica and stops retrying that replica when l messages have been sent to it. The protocol terminates when all replicas have been tried l times or the reply count is met.

This protocol improves on **retry** in a number of ways. First, by trying each replica a fixed number of times, it will meet the reply count more often than **retry**, since distant replicas will be tried more times. This bound causes the two protocols to exhibit significantly different behaviors for communication latency and number of messages when message failures are likely. Second, retrying a fixed number of times evens out the number of times messages are sent to each replica, preventing the protocol from trying a dead replica a vast number of times. The message failure measurements suggest that retrying more than

```
// 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-multicast(message m, replica set R, reply count q, delay d)

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

      // send initial messages
      sort R on expected communication latency;
      for i = 1 to q
         send m to R(i);
         delay[i] = 0;
      schedule delay timeout(q) in (d * T[q]) units;
      next = q + 1;

      // process responses
      do forever
         event = getevent();
         select (event):
               case reply(i): succ = succ + 1;
                              if succ >= q              // reply count met
                                 return SUCCESS;
                              else if i == n             // all replicas have been tried, and
                                    return FAILURE;   // reply count has not been met
               case failed(i): if i == n                 // again, all replicas have been tried
                                  return FAILURE;
                               else
                                  schedule retry(i) in delay[i] units;
               case delay timeout(i): if next <= n                 // try another replica
                                         send m to R(next);
                                         delay[next] = 0;
                                         schedule delay timeout(next) in (d * T[next]) units;
                                         next = next + 1;
               case retry(i): send m to R(i);                // try replica i again
                              delay[i] = backoff(i,delay[i]);
```

FIGURE 4.3: Access protocol with retry for failed messages.

of software expires, the protocol schedules a retry for that replica. The first retry occurs immediately, but later retries are delayed according to a backoff function. The performance simulators set each retry delay twice as long as the previous (a choice inspired by the

the reply count. The protocol will adapt somewhat to changing network conditions, in that it orders the replicas, and uses timeouts to observe failures. However, since the protocol only has a persistence of one message, it cannot handle transient communication failure well. I will discuss the performance of this protocol, in terms of messages and latency, in Chapters 6 and 7.

4.5 The retry protocol

Neither **naive** nor **reschedule** accommodate transient failures. The experimental results in Chapter 5 suggest that more than three-fourths of all message failures are transient.[1] The next two protocols accommodate transient failure by retrying messages to replicas after detecting a communication failure. These protocols, called **retry** and **count**, are similar to **reschedule** except that they have a higher persistence. They continually retry failed messages in the hope that the failure was due to some transient problem and the next message will be delivered and acknowledged. These protocols are called *persistent* protocols. They differ in the conditions that they use for determining when to stop retrying. **Retry** continues to retry messages until either the reply count has been met 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.

The persistent protocols improve both the success latency and the probability that the reply count will be met, though at the cost of sending more messages, and possibly at the cost of having longer failure latencies. If so, the persistent protocols will be most appropriate in situations where short transient failures predominate longer failures.

Figure 4.3 shows the protocol for **retry**. Initially, messages are sent to the q closest replicas, where q is the reply count. When the protocol receives a reply, it increments the count of successful replies, and, if sufficient replies have been obtained, declares the access a success. When it finds a message has failed, presumably because a timer in lower layers

[1] 77.12% of all failed messages were part of a run of only one or two messages long in the 24-host sample. See Table 5.2.

```
// 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-multicast(message m, replica set R, reply count q, delay d)

      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

      // send initial messages
      sort R on expected communication latency;
      for i = 1 to q
          send m to R(i);
      schedule delay timeout(q) in (d * T[q]) units;
      next = q + 1;

      // process responses
      do forever
          event = getevent();
          select (event):
                  case reply(i):succ = succ + 1;
                                 if succ >= q          // reply count met
                                   return SUCCESS;
                  case failed(i):fail = fail + 1;
                                 if n - fail < q       // can never meet count
                                   return FAILURE;
                                 else if next <= n          // send extra message
                                   send m to R(next);
                                   reschedule delay timeout(next) in (d*T[next]) units;
                                   next = next + 1;
                  case delay timeout(i):  if next <= n          // send extra message
                                            send m to R(next);
                                            reschedule delay timeout(next) in (d*T[next]) units;
                                            next = next + 1;
```

FIGURE 4.2: Access protocol with extra messages sent on failure.

communication failure time occurs.

This protocol meets the design goals better than the **naive** protocol. It sends to the closest replicas first, a technique that tends to minimize message traffic if the nearest replicas are available. It also will communicate with only as many replicas as are needed to meet able to identify the closest replicas, a problem I consider in the next chapter.

This approach has a problem: it will take much longer than **naive** to complete an operation in the presence of failures. The network services cannot determine that a message has failed until a timer has expired. Since timers aren't supposed to expire before the acknowledgment can arrive, the timeout period is set to a long value – one that covers more than 99% of all messages. The problem is that most replies take a lot less time than the timeout to complete. If additional messages are sent when replies are somewhat less likely to have arrived the operation can complete more rapidly.

To address this, the **reschedule** protocol sends additional messages at some fraction d of the failure timeout. The *delay* parameter d can be used to tune the protocol to account for different conditions. Since there are some situations where the protocol may be able to detect communication failure in a shorter period than the failure timeout – perhaps because IP reported an unreachable network – the protocol can also make use of this early failure detection.

The complete protocol is shown in Figure 4.2. The network layer automatically sets a failure timer for each message. When it expires it causes a message failure event. The timer period is set to the current estimated failure timeout period T_r, which I will discuss in the next chapter.

The implementation orders replicas by expected communication latency to determine the order in which messages should be sent. This will generally cause the protocol to communicate with the closest available replicas. However, as discussed in the last chapter some replication protocols can use other orderings to advantage.

When the delay parameter d is set to zero, **reschedule** is identical to **naive**: messages are sent to all replicas right away because the delay timer for sending the next message expires immediately. When d is set to one, **reschedule** only sends additional messages when communication failures are detected, because the delay timer period is the same as the failure timeout period. When d is set to one-half, messages are sent to additional replicas either if a failure is reported, or if a timeout of one-half the longest expected

```
// Naive – send to all replicas
naive-multicast(message m, replica set R, reply count q)

    int n = |R|;
    int i;

    // send messages
    for i = 1 to n
        send-datagram(replica=R(i), message=m);
    schedule timeout in max(fail(i),i=1..n) units;

    // collect responses
    do forever
        event = getevent();
        select (event):
                case reply(i): q = q - 1;
                               if q == 0
                                  return SUCCESS;
                case timeout: return FAILURE;
```

FIGURE 4.1: Naive access protocol.

meet the reply count or decide that it is unobtainable. The second is that the protocol has a persistence of one message, that is, the failure of just one message to a host causes the protocol to treat the host as unavailable. The **naive** protocol neither accounts for transient communication failures nor does it use proximity to improve performance.

4.4 The reschedule protocol

Reschedule is the first of the quorum multicast protocols, and addresses the first problem with **naive**. This protocol sends fewer messages than **naive**, though often at the expense of requiring a little more time. It still has a persistence of one message, so it does not solve the transient communication failure problem. The **reschedule** protocol sends messages to nearby replicas before sending to more distant replicas. A protocol that sends a minimal number of messages will first send messages to the q closest replicas, and send to additional replicas as the earlier messages are observed to fail. The protocol must be

This is the approach taken in a naive implementation of multicast. On the other hand, the lowest network traffic is achieved when the protocol sends messages to exactly the minimum number of replicas that will meet the reply count, sending to additional replicas as failures are detected. The protocol ceases to send them when either the reply count has been met, or sufficient failures have been observed to be sure that the it cannot be met. This is the basis for the three quorum multicast protocols.

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 the new protocols is parameterized by a *delay parameter* $0 \leq d \leq 1$ that determines how long to delay sending messages. This mechanism allows an application to specify an intermediate position, where sending more messages than strictly necessary is used to improve operation latency.

This section presents four protocols. The first, called **naive**, is a straightforward implementation of multicast that sends a message to every replica. This protocol provides a baseline to which the other protocols can be compared. The other three protocols are quorum multicasts. The second, called **reschedule**, uses the delay parameter to send to fewer replicas. The third and fourth, called **retry** and **count** respectively, will send to replicas according to the delay parameter, but have a higher persistence and will retry messages to replicas after a first message has failed.

4.3 The naive protocol

The baseline protocol in this study is a simple simulation of multicast called **naive**, shown in Figure 4.1. It operates by sending one message iteratively to all replicas. Replies from replicas are counted, and when a reply count has been obtained the protocol returns, indicating success. The protocol schedules a timeout for the longest expected failure time. If the timeout occurs before the protocol can obtain the reply count, it assumes that the replicas that have not yet replied are unavailable, and declares the access a failure.

There are two problems with this protocol. The first is that it uses more messages than are strictly necessary, though in doing so it requires the minimum possible time to either for several days, and measured the number of operations performed between different registries. He found that the time to communicate with registries on different Ethernet segments was several orders of magnitude larger than that required to communicate on the local segment, and that most traffic was local. The sample consisted primarily of electronic mail traffic. He concluded that the locality of work groups and organizations led to a natural clustering of hosts with shared information.

Bulk data transfer protocols provide yet another common communication protocol. While broadcast, multicast, and RPC are usually used for small quantities of data, bulk transfer protocols are more efficient when large amounts of data are to be transferred. Many of these protocols use sliding-window techniques to avoid waiting for an acknowledgment on every packet. Comer [Comer88, Comer91] provides a nice summary of these techniques in his discussion of the TCP protocol [Postel80b] that is ubiquitous throughout the Internet. Carter and Zwaenepoel [Carter89] implemented a bulk transfer protocol for the V system that got peak bandwidth of more than 8 Mbit/sec on a 10 Mbit/sec Ethernet. Boggs *et al.* [Boggs88] measured the performance of bulk transfer protocols on Ethernets, and concluded that well-constructed protocols can generally make use of all the available Ethernet bandwidth.

4.2 Quorum-based multicast protocols

In Chapter 2, I defined the abstract model of a quorum multicast protocol. This section presents some designs for such protocols. The protocol designs are guided by four goals: make use of proximity (or other orderings); communicate with subsets of the destinations; adapt to changing network conditions; and minimize latency and message traffic. Some of these goals conflict. For example, to provide the lowest latency, a protocol must obtain sufficient reply messages at the earliest possible time. This implies that the protocol should send messages to all replicas at once, since delaying any one message could delay success or failure. Unfortunately this approach produces high message traffic, since messages are sent to all replicas even though not all replicas need to reply to fulfill the reply count.

Remote Procedure Call [Birrell84] is another communication model, providing a specialized communication service to higher layers that implement replication or application policy. Simple Remote Procedure Call (RPC) provides request-response communication between two processes, while extended versions provide one- or many-to-many communication. A process performs an RPC by making what appears to be a normal procedure call; the procedure's parameters are copied into an RPC request message that is sent to the process that provides that function. When the function is to return, the return values are copied into a reply message that is sent back to the calling process. RPC communication is essentially request-response datagram communication packaged with a convenient interface.

Most RPC systems provide one-to-one communication. Birrell and Nelson [Birrell84] report on one of the early and most influential RPC systems. Their RPC made use of a custom network protocol optimized for request-response behavior. Host failure was detected by the caller periodically polling the callee; no time limits were set on the called computation. Many other RPC systems have been developed.

Some RPC systems provide one-to-many and many-to-many communications. These systems provide communication semantics similar to a request-response multicast protocol. The Circus replicated RPC system [Cooper84] provided many-to-many communication by extending the idea of RPC to include replicated calls to a set of processes, called a troupe; each process in the troupe was required to perform the same computation, and issue the same RPCs in the same order. Calls were tagged in such a way that multiple calls from a troupe could be identified as representing the same call. The Parallel Remote Procedure Call (PARPC) system [Martin87] implemented one-to-many replicated procedure calls. The PARPC system associated a block of code with each RPC statement that was executed once for each return message the client received. Replication protocols could be implemented in both these system using only a few lines of code.

Terry [Terry85] measured the effect of using nearby data sources for name service operations. He recorded the activity of the Xerox Grapevine [Birrell82, Schroeder84] service inherent broadcast capability. Boggs [Boggs83] developed *directed broadcast* capabilities for internetworks, and includes a survey of broadcast techniques. A directed broadcast is delivered to all hosts on a network segment, even if the host sending it is not connected to that segment. His work argues persuasively that internetworks should not be built without broadcast for name and routing services. Garcia-Molina and Kogan [Garcia-Molina88] extend internetwork broadcast algorithms with a novel mechanism that provides a reliable multicast facility on an internetwork with unreliable multicast. Their work is especially interesting since it deals efficiently with partitioned networks, and because it takes advantage of straightforward knowledge of the topology of the internetwork. In related work, Alon *et al.* [Alon87] present a broadcast-like protocol that requires $O(n \log n)$ messages to coordinate work among n hosts.

In contrast to this work, the Isis system [Birman87] has concentrated on providing a distributed programming environment based on reliable *atomic broadcast* in a local area. The Isis system provides specialized broadcast protocols [Birman90, Birman91] among a group of processes, providing strong guarantees on the ordering and atomicity of delivery and failure detection. Several different process grouping structures have been identified, including client-server and peer relationships. Communication occurs between a process and the other members of a process group. The system can be used to implement reliable communication for data replication, or for redundant computation in a fault-tolerant system.

Many researchers have investigated multicast protocols. These protocols provide one-to-many communication services. The multicast is usually used for request-response communication. Cheriton [Cheriton84] used multicast as a primary communication mechanism in the V system, a locally-distributed operating system at Stanford. The V system used distributed *process groups* as the endpoints for a multicast. He also considered how source-routing techniques could reduce the cost of multicast in an internetwork in his work on the Sirpent system [Cheriton89]. Boggs [Boggs83] considered multicast as a part of his Internet broadcast work.

Chapter 4

Quorum multicast protocols

In my four-layer model of systems that use replicated data, the *communication layer* defines mechanisms that are used to implement replication protocols. These mechanisms are used to send messages to the replicas storing the shared data. The communication layer in turn uses the primitives operations provided by the network.

This chapter examines the communication layer in detail. In previous chapters I have introduced one possibility for this layer, the *quorum multicast* protocols. This chapter begins with a survey of several related communication mechanisms, then proceeds to detail four multicast protocols. One of these protocols, called **naive**, is a simple multicast. The other three implement quorum multicasts, and are called **reschedule, retry,** and **count.** Each of these three take advantage of replica locality. They differ in the ways they respond to message failure.

4.1 The communication layer

The communication protocol is the middle layer in my replication model. It is responsible for sending messages to the hosts named by the replication layer, and for detecting and reporting host failures.

Many prior analyses have assumed that a broadcast communication protocol would be used at this layer, and such protocols are well-researched. Several researchers have considered the problem of providing a multicast facility on an internetwork that has no

3.3 Summary

A great many replication protocols have been proposed. Some of them define consistent semantics that mimic the behavior of a single copy of the data. These protocols are generally synchronous. Other protocols relax this constraint, generally providing single-copy serializability for important operations or transactions while allowing less critical operations to observe slightly inconsistent data. Epidemic protocols take this to the extreme, providing only weak, probabilistic bounds on consistency, but requiring the least overhead to execute.

Quorum multicast communication protocols can be used profitably to implement several replication protocols. The Majority Consensus Voting protocol can be implemented as one or two quorum multicasts, so the effect on replication performance is nearly linear in the performance of quorum multicast. Available Copy and Dynamic Voting protocols cannot be implemented entirely in terms of quorum multicasts, so their performance improvement is somewhat less. However, quorum multicasts can still effect significant improvements. Epidemic replication techniques do not use multicast, so they show no performance improvement.

Even if a replication protocol does not exhibit better performance using quorum multicast, the technique is useful for its clear definition of failure detection and its fault-tolerance. Some quorum multicast protocols declare a replica unavailable only after it has not responded to a given number of messages. This can be used to approximate actual failure detection with high probability. The ability to retry communications makes quorum multicasts more robust in the face of transient network problems than a simple multicast protocol.

operations.

In the epidemic replication model proposed by Demers *et al.* [Demers88, Demers89], applications contact a nearby replica for all operations. They also define three replica-to-replica operations to propagate information updated by the application: *direct-mail, rumor mongery,* and *anti-entropy.* Read and update operations can use quorum multicast with the reply count set to one as a fault-tolerant means of contacting the closest replica. A variant on multicast that sends to all replicas but only requires a reply count of responses can be used to implement best-effort dissemination for direct-mail operations.

Rumor mongery and anti-entropy, like an update, are one-to-one operations. These operations can show significant performance improvement when performed with nearby replicas more often than distant replicas. The performance analysis by Demers *et al.* showed that real internetworks often contain a small number of heavily-used links connecting different regions, and traffic across these links should be avoided when possible. Quorum multicast protocols can achieve this aim using a reply count of one, and an ordering of closest to farthest on replicas. Unfortunately, the epidemic replication algorithms do not disseminate information properly when rumors are always propagated to the nearest replica. Instead they work best if all replicas have some chance of being selected to receive a rumor. The replicas can be ordered on some random function of the communication latency which generally prefers nearby replicas but which will, from time to time, prefer more distant replicas.

Epidemic replication is a candidate for implementing the distributed version of the reference database. This method has some very attractive aspects: it allows heterogeneous, batch-style transport mechanisms for connecting replicas, and it has very low overhead. Quorum multicast protocols would connect clients to nearby replicas for querying and updating the database. The primary benefit of using these multicast protocols within an epidemically-replicated system is that it provides a convenient way to provide fault-tolerant access to nearby replicas.

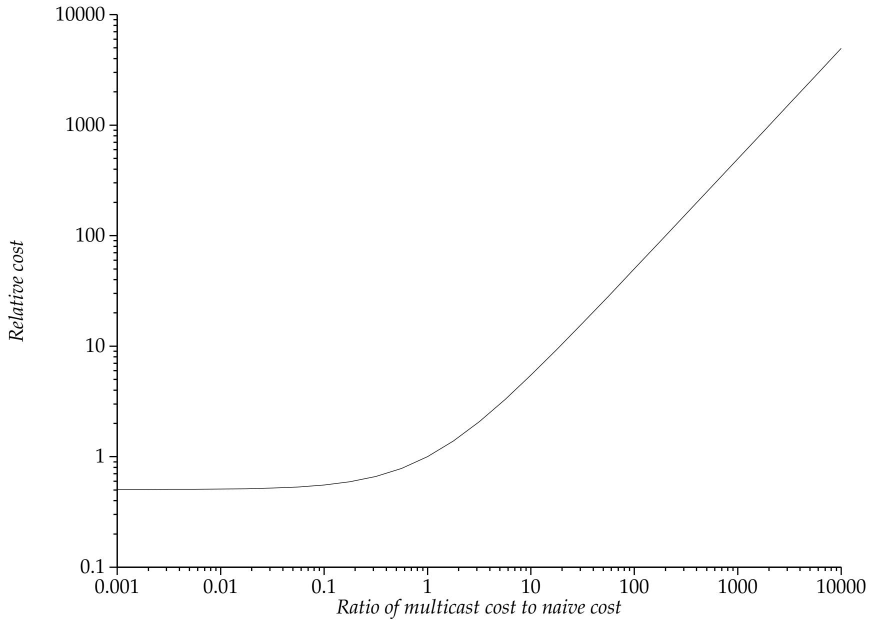

FIGURE 3.7: Estimated performance improvement for DV as σ varied.

3.2.5 Epidemic replication protocols

Epidemic replication techniques are similar to the Available Copy protocols in that read operations need only be performed at one replica. However, epidemic replication protocols relax consistency requirements so that not all replicas need to participate simultaneously in update operations. The usual approach is for update operations to occur at only one replica, and for other background methods to spread the update to other replicas. This generally gives probabilistic guarantees on the currency of information retrieved on a read.

Applications using epidemically-replicated data will not observe a performance benefit from quorum multicast protocols. However, the *convenience* of these protocols can be a significant aid to implementation of the replication protocols. When a process performs a quorum multicast with the reply count set to one, the communication will contact the closest available replica, providing fault-tolerant one-to-one communication. Further, replicas can use an alternative ordering on replicas (other than proximity) in reconciliation that all operations operate on large amounts of data, making two-phase read operations preferable, the cost of using quorum multicast for a read operation is

$$C_{r,q} = C_n + C_q \tag{3.5}$$

while the cost of performing a read using naive multicast is

$$C_{r,n} = 2C_n \tag{3.6}$$

Likewise the cost of an update is

$$C_u = 2C_n \tag{3.7}$$

Combining these equations, the relative cost of using quorum multicast instead of naive multicast is

$$\begin{aligned} Q_{DV} &= \frac{C_q}{C_n} \\ &= \frac{\rho C_{r,q} + C_u}{\rho C_{r,n} + C_u} \\ &= \frac{\rho(C_n + C_q) + 2C_n}{2\rho C_n + 2C_n} \\ &= \frac{\rho C_n + \rho\sigma C_n + 2C_n}{(2\rho + 2)C_n} \\ &= \frac{\rho + \rho\sigma + 2}{2\rho + 2} \end{aligned} \tag{3.8}$$

Figure 3.7 shows the effect of various values of σ on the relative performance, again assuming from the reference database example that the read-to-update ratio $\rho = 100$. The asymptotic performance improvement is only two-thirds of that observed for the Available Copy protocol, since the DV protocols must multicast some messages to all replicas.

This analysis does not include the effects of replica recovery. When a replica using DV recovers from a failure, it must execute an expensive recovery protocol. The probability that an operation will observe a spurious failure is related to the persistence of the multicast protocol, which also affects the protocol's performance.

imal version number, and the operation is complete. If the data being returned are large, then it may be better to use two rounds rather than one. In the first round replicas reply with only their version number and partition vector. In the second round the client requests one of the replicas with the maximal version number to send a copy of the data.

If the read operation is being performed in one round, the only advantage the client can obtain from quorum multicast protocols is that of convenience. All replicas must be contacted to ensure that the most current version is obtained, so no advantage can be obtained by using small reply counts. On the other hand, the multicast protocols provide well-defined failure semantics which are important to the replication protocol. A replica that is considered to have failed must execute a costly recovery protocol. Highly-persistent quorum multicasts declare few spurious failures, thereby avoiding unneeded recovery costs.

However, if the read operation is done in two phases, the quorum multicast protocols can be used in the second phase. If the client performs a multicast to the set of current replicas, with the reply count set to one, quorum multicast protocols will contact the closest available replica to obtain the data. Since the two-phase protocol will generally be used when the data are large, using the nearest replica can provide significant performance improvements.

Performing an update operation in a DV protocol is more complicated, requiring two phases to complete. In the first phase the client sends a write request to every replica, to which each replica replies with its version number and partition vector. In the second phase the client multicasts a commit message containing a new version number and a new partition vector to all replicas that responded in the first phase. The new partition vector is composed of those replicas that responded with current versions in the first phase. At the end of the second phase the replicas respond with the results of the update operation. This operation does not make use of quorum multicast.

To determine the performance improvement, I will once again define the read-to-update ratio as ρ and the ratio of quorum multicast cost to naive cost as σ. Assuming and a *partition vector* indicating which replicas participated in the last update operation. An out-of-date replica will have a version number less than that of current replicas.

Figure 3.6 shows a possible implementation of DV using quorum multicast.

```
context:
    R: list of replicas
read(handle) → data

    // first phase: obtain version numbers
    reply list = multicast(message='read data version', replica set=R);
    if reply list does not contain a majority of the hosts in the latest partition vector
      raise exception 'data set unavailable';
    else
      // second phase: get a copy of the data
      replica set M = replicas in reply list with max version number;
      data reply = quorum-multicast(message='read data', replica set=M,
                                   reply count=1);
      if exception 'reply count not met'
        raise exception 'data set unavailable';
      else
        return result from message in data reply;

update(handle,data)

    // first phase: send data and obtain partition vectors
    replies = multicast(message='write data request', replica set=R);
    if reply list does not contain a majority of the hosts in the latest partition vector
      multicast(message='abort write', replica set=R);
      raise exception 'data set unavailable'
    else
      // second phase: commit write
      generate new version number and partition vector from replies;
      multicast(message='commit write', replica set=new partition vector);
```

FIGURE 3.6: DV implementation using quorum multicast.

To perform a read operation, the client multicasts the read request to all replicas. Every available replica responds with their version number, their partition vector, and the requested data value. The client picks the data value returned by a replica with the max-

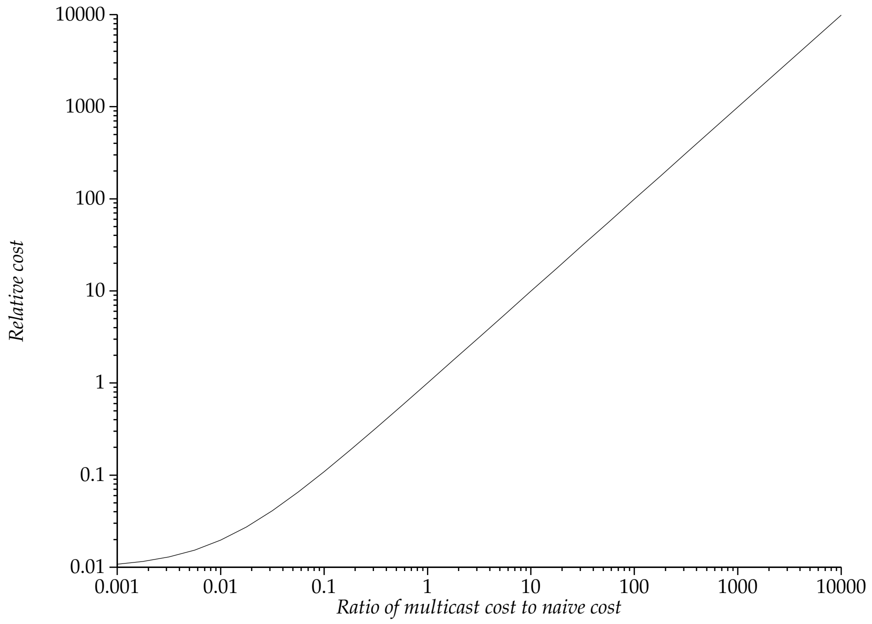

FIGURE 3.5: Estimated relative cost for MCV as σ is varied.

3.2.4 Dynamic Voting

Dynamic Voting (DV) protocols [Davčev85, Jajodia87, Long88] can provide higher availability than MCV protocols. They do so by adjusting the quorum sizes required for success as replicas fail and recover. Replicas may fail at any time, and failure removes them from the set of replicas from which a quorum is formed. An operation must be performed at a majority of the available replicas to complete successfully. When a failed replica recovers it must execute a protocol to rejoin the set of available replicas. The number of available replicas can be as small as two, if global ordering is maintained on the replicas to break ties. For this discussion I will not consider versions of DV that make use of regeneration [Long89a], but rather concentrate on DV protocols that use a fixed set of replicas.

One way to implement DV is to provide the applications using a set of replicated data with the addresses of the set of replicas storing the data. Since the set of replicas are assumed fixed, the application can obtain this information once at startup. Each replica maintains the state of the shared data, a *version number* indicating how current its data are,

```
context:
    R: list of replicas
read(handle) → data

    quorum size = $\lfloor |R|/2 \rfloor + 1$;
    reply list = quorum-multicast(message='read data', replica set=R,
                                  reply count=quorum size);
    if exception 'reply count not met'
      raise exception 'data set unavailable'
    else
      return result from message in reply list

update(handle,data)

    quorum size = $\lfloor |R|/2 \rfloor + 1$;
    request reply list = quorum-multicast(message='write data request', replica set=R,
                                          reply count=quorum size);
    if exception 'reply count not met'
      multicast(message='write data abort',
                replica set=replicas in request reply list);
      raise exception 'data set unavailable'
    else
      multicast(message='write data commit',
                replica set=replicas in request reply list);
```

FIGURE 3.4: MCV implementation using quorum multicast.

Again assuming a read-to-update ratio of $\rho = 100$, the overall relative cost is

$$Q_{MCV} = \frac{101\sigma + 1}{102}.$$

Figure 3.5 shows the sensitivity of the overall relative cost as to relative performance of quorum multicast. For values of σ near 1, the the overall cost is nearly proportional to the cost of a quorum multicast. This linear effect makes quorum multicast protocols attractive for implementing MCV replication protocols.

3.2.3 Majority Consensus Voting

When the number of read operations only slightly exceeds the number of update operations, the AC replication protocol can be slower than requiring majorities of the replicas to participate in both read and update operations. Read and update operations have about the same cost using the Majority Consensus Voting (MCV) replication protocol [Thomas79, Gifford79]. If the set of replicas is fixed, the multicast protocols provide the mechanism to ensure that a majority of the replicas participate in an operation.

A read operation can proceed in one phase, with the client multicasting a request message and available replicas replying. The client can set the reply count to a simple majority of the replicas. An update operation, on the other hand, will usually require two phases, one to multicast the request and another to commit the operation once a majority has been obtained. The request can be multicast to at least a majority, and the reply count should be set to a majority. The commit phase should be multicast to all the replicas to which the request was sent, and the reply count must be the same as that in the request phase. Figure 3.4 shows how this replication protocol can be implemented using quorum multicast. An implementation using naive multicast would substitute a naive multicast for each quorum multicast.

As in the last section, an estimate can be computed for the relative cost of using quorum multicast. This estimate ignores the cost of recovering failed replicas. If the cost of using quorum multicast instead of naive multicast is σ, as in the last section, then the relative cost is

$$\begin{aligned} Q_{MCV} &= \frac{C_{MCV,q}}{C_{MCV,n}} \\ &= \frac{\rho C_q + C_q + C_n}{(\rho+2)C_n} \\ &= \frac{(\rho+1)\sigma C_n + C_n}{(\rho+2)C_n} \\ &= \frac{\rho\sigma + \sigma + 1}{\rho + 2} \end{aligned} \tag{3.4}$$

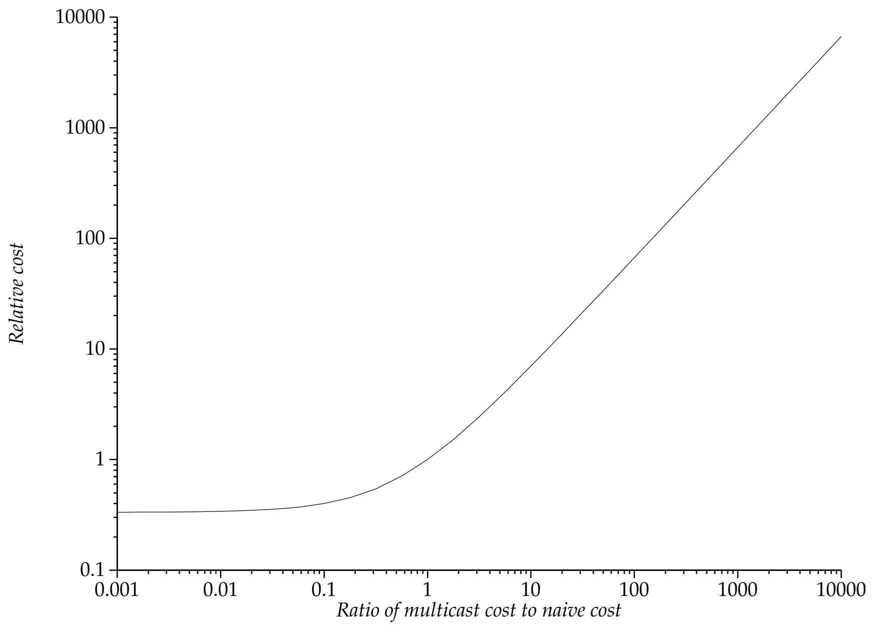

FIGURE 3.3: Estimated relative cost for AC as σ is varied.

of an expensive protocol for adding new references is small. This is potentially a good candidate for AC protocols.

I estimate the read-to-update ratio for the reference data bases at about $\rho = 100$ accesses for every update. In the absence of a concrete implementation I rather arbitrarily estimate that an update requires about $\omega = 50$ times as long as a naive multicast. The expected relative cost for communication latency is

$$Q_{AC} = \frac{100\sigma + 50}{150}.$$

Figure 3.3 shows the effect of different relative protocol performance on the overall communication cost for the reference database. When quorum multicast protocols make read operations nearly cost-free, the overall cost is about one-third the cost using naive multicast. On the other hand, if a quorum multicast implementation causes read operations to cost 80% more than a naive multicast implementation, the overall cost is about 50% greater.

quorum multicast for reads and one atomic multicast for writes.

```
context:
      R: list of replicas
read(handle) → data

      reply list = quorum-multicast(message='read data', replica set=R,
                                    reply count=1);
      if exception 'reply count not met'
        raise exception 'data set unavailable'
      else
        return result from message in reply list

update(handle,data)
      atomic-multicast(message='write data', replica set=R);
```

FIGURE 3.2: AC implementation using quorum multicast.

If the cost C_a of an atomic multicast in a particular configuration configuration is a factor of ω more expensive than the cost C_n of a naive multicast, the overall relative cost is

$$\begin{aligned} Q_{AC} &= \frac{C_{AC,q}}{C_{AC,n}} \\ &= \frac{\rho C_q + C_a}{\rho C_n + C_a} \\ &= \frac{\rho\sigma C_n + \omega C_n}{\rho C_n + \omega C_n} \\ &= \frac{\rho\sigma + \omega}{\rho + \omega} \end{aligned} \tag{3.3}$$

The only cost not considered in this analysis is the cost of recovering a failed replica, which is dependent on the rate of failure.

In the reference database, hundreds of reference accesses are often performed for every reference update. Further, users can often tolerate some delay between the time they submit a new reference and the time that the reference is available in the database. In the current implementation at Hewlett-Packard Laboratories, new references are entered into a log during the day and generally only added to the database at night. Since updates are comparatively rare, and because delays do not generally directly affect users, the cost access is almost completely read-dominated. In such a system, update operations can be very expensive, perhaps using a multi-phase atomic multicast protocol to achieve consensus among all replicas [Cristian86, Gopal90]. These protocols can require several rounds of messages, or may require message delivery to be delayed. This can result in order-of-magnitude greater latency and message counts than that required by the simple (but far less capable) quorum multicast protocols. In spite of this the combination can be efficient if read-only operations predominate because they need only be performed at one current replica.

In the analysis that follows, I will assume that an atomic multicast is a factor of ω more expensive than a naive multicast. This treatment is only valid for a particular configuration of replicas and clients, for the actual performance of an atomic multicast depends on the number and placement of replicas.

Figure 3.1 shows an implementation of AC using simple multicast. This implementation uses one naive multicast for reads, and one atomic multicast for writes.

```
context:
    R: list of replicas
read(handle) → data

    reply list = multicast(message='read data', replica set=R);
    if size of reply list < 1
      raise exception 'data set unavailable'
    else
      return result from message in reply list

update(handle,data)
    atomic-multicast(message='write data', replica set=R);
```

FIGURE 3.1: AC implementation using naive multicast.

The quorum multicast protocols ensure that reads occur at the closest available replica when the reply count is set to one. If the nearby replica becomes unavailable, the protocol will switch to the next closest replica. Figure 3.2 shows an implementation. It uses one

3.2.1 Sample application: a bibliography database

Throughout the rest of this section I will use an example of a simple replicated database to make discussions concrete. I use the refdbms reference database [Wilkes91] for these examples, and consider ways in which a distributed version could make use of quorum multicast protocols.

The refdbms system is a bibliography database system developed at Hewlett-Packard Laboratories. It provides tools to add references to the database, to search for references using keywords, and to format references for bibliographic citations in scholarly communications.[1] Refdbms is similar to systems such as refer [Lesk78], BibTeX [Lamport85, Patashnik88], and tib [Alexander87], but has been designed from the start to support databases shared among groups of users.

Refdbms is an example of an application well-suited to distribution. The database structure is simple, and activity is almost entirely read-only, consisting of searches for interesting references and extraction of references for documents. The majority of operations that update the database add new references without modifying old ones. A small fraction of the operations modify existing references, either to correct errors or to add information.

Two models are being considered for building a distributed version of refdbms. One model uses Epidemic replication techniques (§3.2.5) to provide nearly-consistent replication, while the other uses direct replication techniques such as the Available Copy (§3.2.2) or voting protocols (§3.2.3, §3.2.4) for completely consistent databases.

3.2.2 Available Copy

The Available Copy (AC) replication protocol requires that update operations are performed at all replicas, while read operations can be performed at any available replica. While this protocol is generally not considered useful in an internetworked system because of its vulnerability to communication partitions, it can be useful in a system where

[1]The bibliographic information for this thesis was maintained using refdbms.

of the latency and number of messages that can be expected if quorum multicast protocols are used instead of regular multicast to all the replicas.

In the analyses that follow I use ρ as the *read-to-update ratio,* the number of read operations performed for each update operation. Many applications, including the reference database described in the next section, exhibit read-to-update ratios on the order of $\rho = 100$. I define σ as the ratio of the cost of reading data using quorum multicast to the cost of reading them using naive multicast. If the cost metric is the number of messages sent, the simulations in Chapter 6 show that this ratio is around $\sigma = 0.8$ in some situations. For latency the ratio might be $\sigma = 0.6$. If $\sigma < 1$, quorum multicast techniques are preferable.

Throughout this section I write the cost of a read operation as C_R, and the cost of an update operation as C_U. When the cost is specific to one multicast protocol I add a second subscript indicating the protocol. For example, $C_{R,q}$ is the cost of a read using quorum multicast.

Given a read-to-update ratio ρ, the mean cost C over all operations for a replication protocol P and communication protocol c is

$$C_{P,c} = \frac{\rho C_{R,c} + C_{U,c}}{\rho + 1}. \tag{3.1}$$

This equation only accounts for the cost of operations directly caused by an application. Operations such as background update and failure recovery are not considered because they do not affect the immediate performance of an application, and can often be performed in off-peak times. I will discuss the costs of such operations when appropriate.

The *relative cost* of using quorum multicast is given by

$$Q_P = \frac{C_{P,q}}{C_{P,n}} \tag{3.2}$$

for a replication protocol P. Values of $Q_P < 1$ suggest that quorum multicasts will be beneficial.

as a *hint.* Hints provide good performance as long as they are usually correct and the cost of identifying and recovering from out-of-date hints is low.

Epidemic replication and the causally-consistent techniques of lazy replication can be combined, for example by allowing applications to specify predecessor operations and consistency constraints [Golding91]. When applications do not specify such constraints, the system provides probabilistic guarantees. When constraints are specified, the system guarantees consistent execution although the delay on an operation is only probabilistically bounded.

3.1.4 Related techniques

The *Regeneration Algorithm,* proposed by Pu [Pu86a, Pu86b], is a technique that can be combined with several replication protocols. The algorithm regenerates new replicas when it detects that one or more of the replicas have become inaccessible due to site failures. While reads are allowed under this protocol so long as one current replica of the data remains accessible, updates are disabled if fewer than the initial number of replicas are accessible and there are not enough spares for the missing replicas to be regenerated. No provisions are made by the algorithm for enforcing mutual exclusion or for recovering from a total failure. Long and Pâris [Long89b, Long90b] extended the regeneration algorithm to compose with Available Copy and Dynamic Voting protocols, as well as with volatile witnesses. Long, Carroll, and Stewart analyzed the reliability of these composite protocols [Long89a].

3.2 Implementing replication protocols

In this section I study the performance and implementation of some of the replication protocols in the last section. The section covers Available Copy, Majority Consensus Voting, Dynamic Voting, and Epidemic replication protocols. For each of these protocols I discuss whether it can benefit from quorum multicast protocols, and present an analysis

Lazy replication

The *lazy replication* scheme [Ladin90, Ladin91] provides application-specified causal consistency. Each operation is sent to one replica, where it is given a unique identifier. That replica then forwards the operation to other replicas.

When the application issues an operation request, it specifies a set of operation identifiers which must precede the operation. When a replica receives a request, it must delay executing the operation until all the predecessor operations have been received and executed. If all applications list *all* operations they have performed as the predecessors of a new operation, every operations will observe causally consistent data. Operations that do not depend on other results – such as the addition of a new datum – need not specify any predecessors, and replicas can execute the operation immediately.

Ladin *et al.* reduce the cost of this technique using *vector timestamps* [Mattern88] to identify operations. A set of operation identifiers can be compactly represented as a single vector timestamp. This vector is formed from the piecewise maximum of each of the vectors in the set.

3.1.3 Unbound-inconsistency protocols

The bound-inconsistency techniques provide hard bounds on the divergence among replicas. *Epidemic* replication schemes [Alon87, Demers88], on the other hand, only provide probabilistic bounds. Applications send operation requests to one or more replicas, which execute the operation as soon as possible. The operation is later propagated to other replicas. Several propagation methods can be used, including *direct mail, rumor mongery,* and *anti-entropy.* Each method propagates the update at different rates, requiring different amounts of message traffic and providing different guarantees on eventual consistency.

Epidemic techniques have been used in internetworked name services such as Clearinghouse [Oppen81] and Grapevine [Schroeder84, Terry85]. Name services are a good candidate for Epidemic replication because an application can treat location information

mation, and execution of an update is delayed until the earlier updates have completed. Commutative operations, on the other hand, can be executed at any replica in any order. As long as all update messages are reliably delivered, all replicas will eventually converge to the same value. It is not clear exactly how fast they converge.

Controlled inconsistency

The *controlled inconsistency* approach [Barbará90], which allows data to be annotated with inconsistency constraints, has been proposed by Barbará and Garcia-Molina. They discuss arithmetic and temporal constraints in some detail. Arithmetic constraints allow a replica to diverge from another master replica by some value. For example, a datum might vary no more than three units from the correct value. Update operations can proceed at any replica as long as they will not cause the resulting value to exceed some constraint. If they do, then the operation is performed at the master. Temporal constraints ensure that a replica will not diverge from the master replica for long periods.

Escrow

The *escrow* methods [Kumar90] define specialized replication protocols that can be used when performing commutative updates on aggregate data. These methods are a variation on voting protocols. With this method, the degree by which an update can change data is proportional to the number of votes it can collect. For example, an operation that collected six votes might only be able to change the value of the data by six units. This technique avoids the 'all-or-nothing' behavior of usual voting protocols, which either succeed or fail. One variant of the algorithm provides serializability. Pâris [Pâris90a] has proposed a variation intermediate between escrow and MCV protocols, where each update consumes votes given to an application or replica.

The replication protocol used in the Echo file system [Hisgen90, Mann89] uses a *master* replica to coordinate operations for a set of *slave* replicas. An application forwards its operation requests to the master, which in turn synchronously sends the request to all slaves. When a replica recovers from a failure, or when the master replicas fails, an *election* is held to determine a new master. The protocol also allows applications to *cache* data. An application obtains the right to cache a datum for a limited time. The application must write any changes back to the replicas before the time expires, and the replicas reject any changes written after expiration.

3.1.2 Bound-inconsistency protocols

The consistent replication protocols guarantee that all replicas observe updates "at the same time". This can be an expensive guarantee to provide, since most of the known protocols require synchronous operations at some or all replicas. The *bound-inconsistency* protocols define slightly weaker consistency semantics, and so gain performance by permitting asynchronous operations. These protocols strictly control the divergence among replicas.

Epsilon-serializability

Pu and Leff have worked on formalizing asynchronous replication protocols. Their *epsilon-serializability* [Pu91b, Pu91a] techniques define a model of serializability and transactions that allow read transactions to observe transient inconsistencies between transactions and between replicas. Unlike voting protocols, the replication protocols do not perform synchronous update of a number of replicas. Update messages are propagated asynchronously among replicas. The replicas are guaranteed to converge to a common, serializable view of the data after all messages have been received.

They detail several kinds of updates, including commutative, ordered, and timestamped updates. Each kind of update requires a different implementation to ensure convergence. Ordered updates are sent between replicas in messages tagged with order infor-

One difficulty of the Majority Consensus protocols is that it uses a static assignment of votes to replicas. Barbará *et al.* [Barbará86, Barbará89] examined the idea of dynamically reassigning votes on node or link failure. They describe two policies for vote reassignment: *group consensus* and *autonomous assignment*. In the group consensus method, sites in the majority group decide on a global vote assignment. Sites that are not in the majority group receive no votes under the new vote assignment. A coordinator site is elected and collects topology information from the other sites. The coordinator then decides on a new vote assignment and distributes the votes among the sites.

While the vote assignment protocols handle failure and recovery by reassigning votes and maintaining quorum sizes, the Dynamic Voting (DV) protocols [Davčev85] adjust quorum sizes while preserving vote assignment. The initial formulation of this protocol was unrealistic in that it assumed instantaneous knowledge of the system. Jajodia and Mutchler removed this limitation [Jajodia87], and explored the possibility of using both DV and MCV to increase availability [Jajodia88]. Pâris and Long defined an optimistic version of the DV protocol that can operate on out-of-date information [Long88, Pâris88]. They also considered how DV could be used when true stable storage is not available, requiring that updates be recorded in at least two replicas [Pâris90b].

Explicit subsets of sites, known as *coteries* [Garcia-Molina85], have also been studied for controlling access to replicated data. Access is allowed if a superset of the sites in one or more of the coteries are present. For maintaining mutual consistency the only restriction on the composition of coteries is that their intersection be non-empty. For more than six replicas, coteries provide a greater flexibility than simple majority consensus.

Other protocols

The *Virtual Partition* protocol [El-Abbadi86] is a hybrid protocol based on Available Copy and voting. A transaction is executed with a *view* consisting of the sites with which it believes that it can communicate. A transaction executes by reading any replica and updating all replicas in the view. A view is required to contain a quorum of the replicas.

This protocol requires that some very strong assumptions be made about the operating environment: the communication network must be reliable; it must not be susceptible to partitions; reliable message delivery must be assured; when a site recovers from a failure it must obtain a current copy of the data (if there is another operational site that holds a current replica then the recovering site can request a copy of the data from that site); in the event of a total system failure the AC protocol must determine the last site to fail since that site held the most up-to-date copy of the data. As a result this protocol is not well suited to internetworks, since they provide unreliable communication and can partition.

Voting

Voting protocols are conceptually quite simple. Each replica is assigned one or more votes. Each operation collects votes from replicas, and can proceed when it has collected a quorum of votes. The simplest approach is to assign each replica one vote, and require each operation to collect a majority of votes. Since all operations operate at a majority, a read operation is certain to contact at least one replica holding the results of the last update operation. Most voting protocols also maintain a version number at each replica, so that out-of-date information can be discarded.

Replica failure can affect voting in one of three ways: not at all, the vote assignment can change, or the quorum requirement can change. Each of these three possibilities is the basis for a voting replication protocol.

The Majority Consensus Voting (MCV) protocol [Thomas79, Gifford79] is one of the simplest replication protocols. This protocol requires that each operation on the replicated data be performed at a majority of the replicas. The size of the majority is determined once for the set of replicas, and never thereafter changed, even if replicas fail. By using a fixed majority, mutual exclusion between client processes is assured. The MCV protocol requires at least three replicas to improve availability over that of a single replica. The MCV protocol was extended by Pâris [Pâris86] with *witnesses*, light-weight replicas which hold no data but which can attest to the state of the replicated data.

bound-inconsistency protocols, and inconsistent protocols. I will also discuss techniques that can be combined with replication protocols to improve their performance or modify their consistency guarantees.

3.1.1 Consistent protocols

The consistent replication protocols make changes to shared data visible to all applications at the “same time”. Since all applications observe the same values at the same time, their views of the data are *consistent.*

The subtlety arises in the meaning of “same time”, since time is a slippery concept in a distributed system. The word *time* can mean *global time,* as an external observer could measure. Time could also mean a *virtual time* measure that is causally consistent among all processes but which does not correspond to global time. These distinctions are discussed further in papers by Mattern [Mattern88], Birman [Birman87], and Lamport [Lamport78].

Consistent replication protocols can be divided into three subcategories. The *Available Copy* protocols require that update operations involve all available replicas, while reads need access only one. These protocols require strict and unrealistic assumptions about the communication environment. The *voting* protocols make less stringent requirements of the network. They assign votes to replicas, and require operations to be performed at a set of replicas holding a quorum of votes. The third group includes hybrids between the first two.

Available Copy

The Available Copy (AC) protocol [Bernstein84, Bernstein87] is often known as the “write-all read-one” protocol. Update operations must be applied at all available replicas. If all available replicas participated in the last update, an application can read from any replica and observe the update. When a replica crashes it becomes unavailable, and must perform a recovery protocol before it becomes available again. Pâris and Long analyzed the availability and performance of several variants of the AC protocol [Long90a].

Chapter 3

Replication protocols

Applications that use replicated data do so according to a replication protocol. These protocols determine how an operation will be performed on replicated data, including the number of replicas that must be involved in the operation. Some protocols define single-copy semantics that ensure changes become visible to applications at the same time, while others define looser semantics. All these protocols are implemented using the services provided by the communication layer.

This chapter has two foci. In §3.1 I survey a number of replication protocols, covering both consistent and inconsistent techniques. I explore ways to implement a select few of these protocols in §3.2, and determine how their performance can benefit from the application of quorum multicast protocols.

3.1 A short survey of replication protocols

The replication layer in my model defines the semantics of the replication. This layer provides the application the illusion of a single data collection, and defines *consistency* semantics for that collection. It also determines how data operations will be coordinated among the replicas.

I survey several replication protocols in the next few sections. The list is not exhaustive, since many protocols have been proposed and there are many variations on each. The protocols are classified by the consistency guarantees they provide: consistent protocols, The communication layer also provides mechanisms for detecting failure, though not for handling it. Handling failures is done in the communication, replication, and application layers. IP will report failures in some cases, such as when it cannot find a way to route a message to its destination. Most of the time, however, failures must be detected using timeout. This requires that the recipient of a message acknowledge it in a timely manner. The majority of replication operations involve request-response communication, so this is not a terribly onerous requirement. Schemes that use timeout and acknowledgment cannot distinguish between failures in the network and in the remote host. For example, a message lost because a router is congested is indistinguishable from a message lost because its destination crashed. The discussion in §5.4.1 presents ways to distinguish between the two probabilistically.

2.4 Implementing replicas

Replicas maintain copies of data. They communicate amongst themselves, and receive requests from clients. A replica can perform operations on the data just as any client can, by issuing a request to its replication layer, which in turn uses the communication layer to send messages to other replicas.

Replicas participate in some protocols that clients do not. For example, when a replica fails and recovers, it may perform a *recovery* protocol to ensure that its copy of the data is current. Some replicas, particularly those using epidemic replication (§3.2.5), will send updates to other replicas from time to time. Once again these operations are performed by issuing a request to the replication layer and so on.

Requests from clients and other replicas are different. They are received as messages at the network layer, and propagated up through the communication layer, which ensures that the request is from a proper client, to the replication layer, which may issue a request to the storage layer depending on the message. Many replicated operations require several rounds of message exchange, and the replica may need to maintain appropriate context to determine its response. Other operations are stateless and require no context.

be declared only after several consecutive message failures. The Internet measurements also indicate that long transient failures are uncommon, so probable host failure can be declared after observing only a few messages. A protocol's *persistence* is the number of messages that must fail before a host is declared failed. One of the quorum multicast protocols described here allows the replication protocol to control this persistence.

Quorum multicast semantics allow the communication protocol to send a message to a subset of the replicas. The protocols I have developed use an ordering on the replicas to determine which to prefer. The simplest ordering is based on expected communication latency, though many others are possible. As we will see in §3.2.5, orderings based on probable currency of data and randomized orderings can also be useful.

2.3 The network layer

The Internet defines a very simple communication environment. It provides a base *Internet Protocol* (IP) [Comer88] that routes datagrams through the internetwork to remote hosts. This protocol uses 32-bit integer host identifiers that can be combined with a *port number* at that host to form a communication address.

Two popular protocols are layered atop IP: a reliable byte-stream protocol (TCP) [Postel80b], and an unreliable datagram protocol (UDP) [Postel80a]. The TCP protocol provides virtual circuit semantics, maintaining a continuous end-to-end connection between two hosts. This protocol can require significant time to initiate a connection, and requires both ends of the connection to maintain state for the duration of the connection. TCP is a communication layer protocol in our model.

The UDP protocol provides a connectionless datagram service from one host to another. No delay is incurred by making a connection and no state is required to maintain one. However, the protocol does not provide reliable communication. Messages can be lost, delayed, and reordered in transit by the protocol. I will examine some of the temporal and reliability behaviors of this kind of protocol on the Internet in Chapter 5 .

this multicast should be fault-tolerant, using more distant replicas when those nearby are unavailable.

quorum-multicast(message,replica set,reply count) → reply set

> The message is sent to at least a reply count of the replicas.
>
> Exceptions: reply count not met

FIGURE 2.3: Quorum-based multicast communication protocol.

The *quorum multicast* protocols provide such semantics, as shown in Figure 2.3. These protocols take an additional parameter: the *reply count.* The reply count is the minimum number of replicas that must reply to the message. The protocol may not be able to meet the reply count if some of the replicas are unavailable. Replicas can be unavailable due to host failure (such as a system crash or controlled shut-down), replica failure (perhaps due to insufficient resources or software errors), failure of the network (gateway or link failure), or controlled failure (such as removal of a replica that is no longer needed).

Replicas may also *appear* to be unavailable due to congestion or partial failure, such as when a gateway becomes overloaded. When the protocol fails to receive a reply, it cannot determine whether that message was lost due to transient network problems or due to host failure. If the Internet provided reliable, FIFO or bounded-latency communication channels this determination might be possible. Unfortunately the Internet IP protocol provides an unreliable communication channel between hosts that is neither FIFO nor provides bounded communication latencies, so the communication layer cannot detect replica failure with certainty.

The question of when to declare a probable replica failure can be answered by looking at measurements of the Internet, which I will detail in Chapter 5. Those data show that short transient failures are very common (*e.g.* one or two messages are not delivered by the network or overflow buffers at the receiver). This suggests that sending only one message is not a good way to determine whether a host has failed, and probable host failure should order". *Globally-consistent* data all change value at the same global time [Birman87], while *causally-consistent* data can change value at different times at different replicas, but any application will observe a change at the same *virtual* time [Lamport78, Mattern88].

Other replication protocols do not attempt to provide strict single-copy semantics. Some applications, such as name services or some scientific databases, can get by with less strict guarantees [Terry85]. These applications can obtain better performance using *epidemic* or *controlled inconsistency* techniques, which I will discuss in more detail in Chapter 3.

2.2 The communication layer

Several models can be applied to the interface between the replication and communication layers, including bulk data transfer, remote procedure call, and multicast. This thesis concerns multicast protocols, so I will only consider them here. (See §4.1 for others.)

Simple request-response multicast protocols define an interface like that in Figure 2.2. Replication protocols can use multicast to send a message to all available replicas, later receiving a number of responses from some or all of them.

multicast(message,replica set) $\rightarrow$ reply set

Sends the message to all replicas in the set.

Exceptions: none

FIGURE 2.2: Simple multicast communication protocol.

There are several possible variations to this basic multicast scheme. The most significant variation arises when only some fraction of the replicas need to respond to a message. For example, many replication protocols require that a simple majority of the replicas perform an operation, while others require only one or two replicas. In this case the communication protocol might be able to send the message only to enough of the closest replicas to satisfy the requirement, avoiding message traffic to the most distant replicas. Of course,

2.1 The replication layer

The replication layer generally provides semantics as close to those of a single-copy data object as possible. An application can request that an operation be performed without regard for distribution of data replicas. Operations are usually partitioned into several different types, such as *update* and *read* operations. The replication protocol uses this type to determine how many replicas must participate in the operation, and forwards the request to replicas using a multicast communication protocol.

Figure 2.1 shows an abstract view of the interface to this layer. This is a simplified version that only provides initialization and two operations on the data, reading and updating the entire set of data. Any real data replication would require more complex operations, such as reading or updating some subset of the data, transaction commit and abort, or locking. Every application using a set of data uses the same replication protocol to access it.

open(name) → handle

Given a name for a replicated data set, opens a session with it. Returns an abstract handle to be used in future operations on the data.

Exceptions: no such data set

read(handle) → data

Reads the replicated data.

Exceptions: data set unavailable

update(handle,data)

Writes new data into the replicated data.

Exceptions: data set unavailable

FIGURE 2.1: Minimal replication-layer interface.

This simple interface hides any consistency guarantees provided by the replication protocol. Some replication protocols provide *strict single-copy semantics*. Any application using such protocols is guaranteed to observe all changes to the data in the same time order as any other application using the data. There are different definitions of "in the same time

Chapter 2

Using replicated data

This chapter discusses the four-layer model of replicated data in detail. The *application* provides functions that use shared data. The *replication layer* provides an abstract view of the replicated data that looks much like a single copy. The replication layer uses high-level communication mechanisms in the *communication layer.* The communication layer in turn uses low-level message services in the *network layer* to get messages to and from replicas.

In this chapter I examine this model of building distributed applications in detail. I will discuss the layers from top (the application) to bottom (the network) on the client side, then from bottom to top on the replica side. This thesis is concerned with the lower layers in the four-layer model, so I will mention the application only briefly.

The application defines the functions a user sees. It provides a user interface and determines the semantics of the data. For example, a replicated picture database system might provide applications to retrieve a picture into a local file, to browse through indices of pictures, and to query the database based on tags associated with the pictures. The applications might enforce the constraint that a picture must have an associated format and size on record. The application is written in terms of a single collection of data, without regard for distribution. It uses a *name service* [Terry85] to locate and bind to the replicated data.

try sending a message to a replica once, while other protocols will try several times before giving up. This *persistence* is another tunable parameter in some protocols. Once a protocol has declared enough replicas unavailable, it will return a negative indication to the replication protocol and abandon the operation.

There are several direct uses for the quorum multicast protocols developed in this study. Federated databases that unite several separately-administered databases into one logical database can use these protocols when operations must proceed at several component databases. Other distributed and fault-tolerant services such as a name service can also benefit, since translating a name may involve several name servers. Replicated file systems providing service to a large number of users can also use these techniques to find and use the closest replica of a file.

1.5 Organization of the thesis

The first part of this thesis is organized around the four implementation layers. I start in Chapter 2 with a detailed look at the model and a discussion of the application layer. Chapter 3 steps down one layer to discuss replication, surveying several replication protocols and analyzing how they can use quorum multicast protocols. Chapter 4 goes one layer lower to discuss the communication layer, and quorum multicast protocols in detail. Chapter 5 discusses the lowest layer, where we reach the network level to discuss measurements I have taken of the Internet. The following two chapters take up different performance evaluations of the quorum multicast protocols: Chapter 6 details the simulations; and Chapter 7 covers measurements taken of a sample application. Finally, I draw some general conclusions and future research directions for quorum multicast protocols in Chapter 8.

sensitive to the communication latency of replicas, and should 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. The protocols should also address the problems associated with transient failures by resending messages to replicas. In Chapter 5 I show that this produces a significantly higher availability than can be obtained without retry.

Quorum multicast protocols are a specialization of ordinary *multicast protocols.* Ordinary multicast sends a message to a set of destinations in one operation. Since the message must be delivered to all destinations, the protocol cannot just use nearby ones.

Quorum multicast protocols send a message to a *subset* of the destinations. The number of destinations is a parameter to the operation. For example, replication protocols often do not need all replicas to respond, only a majority. If the communication protocol can obtain at least that many responses the operation will succeed. Since replication protocols generally require request-response communication, the responses to a multicast serve as acknowledgment that the message was received and processed.

The quorum multicast protocols maintain an *expected communication latency* for each possible host. When a request is issued to communicate with q members of a set of replicas, the communication protocol can order the set by expected latency and communicate with the q closest replicas. If responses are not received from all q within a certain time, then messages can be sent to more distant replicas. The time to wait before sending to distant replicas is a tunable parameter that can be used to trade operation latency for message traffic. The expected latency can be determined by measuring recent performance, on the assumption that replicated operations will be performed much more often than the structure of the network will be changed.

If the communication protocol has not received a reply from a replica after some amount of time, the protocol assumes that the message has failed. After some number of messages have failed, the protocol declares the replica unavailable and does not attempt to send further messages until the next communication request. Some protocols will only to provide service even when several parts of the system have failed. The set of failures the application can withstand is called the *fault-tolerance* of the application. Using replicated data rather than a single copy can improve fault tolerance, though at the cost of software complexity and extra communication.

Replicated data can provide lower latency and lower network load than a single data copy, if a copy can be located close to where it will be used. Access requests will usually only need to travel short distances, generally improving latency and throughput, as long as the operation can be completed using the local copy. Operations that require the full set of replicas will lead to longer latencies and provide lower throughput. Experience has shown, however, that in many useful distributed applications operations requiring a small set of replicas can dominate those that require all replicas. These applications should benefit from properly-implemented replicated data.

1.4 Quorum multicast protocols

I have developed a family of *quorum multicast* communication protocols that can be used to take advantage of good replica placement. The protocols in this family send a multicast to a subset of a group of replicas, rather than to the entire group. These protocols can be used to implement a replication policy to maintain consistency between replicas. The protocols use the closest available replicas, falling back on more distant replicas when nearby replicas are not available. They are parameterized so that the replication protocol can provide optimization hints to further improve performance. In this way an application can use replicated data for fault tolerance and throughput, while limiting the cost of message traffic and using nearby replicas for low latency.

A communication protocol that is to work well in internetworks must address their particular performance characteristics: long, variable latency and occasional high message loss. These characteristics make some of the techniques used for replication in a local-area network inappropriate for internetwork use. The protocols should not require broadcast, but instead send messages to replicas in a more controlled fashion. The protocols should be for communications across a continent, and as long as several seconds when satellites are used for transoceanic communication.

Throughput is the rate that operations can be completed. I have not measured throughput in this thesis because it involves many more factors than just communication protocol performance. Just as with latency, the overall throughput is determined by the throughput of the components used in the distributed application: the network, processors, and storage devices. High operation throughput does not imply low operation latency, since several operations could proceed in parallel yielding a high rate of completion while requiring long times to complete an individual operation. I expect throughput in an internetwork to be governed by the loads on the network, determining how many messages can be transmitted per unit time, and on the replicas, determining how many operations can be satisfied. The scale of the Internet implies that components can become very heavily loaded.

The amount of *message traffic* required for an operation governs the degree the operation will interfere with other communication in the network. The number of messages, their size, and their destinations contribute to this effect. When a process sends a message to another host, it takes up bandwidth on each link and at each gateway between the two hosts. A communication protocol will cause less interference if it can send a message to a nearby host rather than a distant host since the message will likely traverse fewer links and gateways. Broadcast messages on a LAN allow replication protocols to send requests to all replicas in one message, while a separate message must be sent to each replica in an internetwork, increasing the message traffic required for replication.

The *reliability* of an application (or service) can be defined as the likelihood of the application (or service) providing continuous correct service to users over a period of time [Trivedi82]. This must be contrasted with *availability,* the probability at any particular time of the application providing correct service. An application can be highly available while having low reliability if it has a very short mean-time-to-failure (MTTF) and mean-time-to-repair (MTTR). Highly-reliable applications must be *fault-tolerant:* they must continue protocol performance, for a protocol that performs well in a local-area network may not scale to the world-wide Internet.

Wide-area networks contain many more systems that might share resources, implying that the potential load on highly utilized components in an internetwork is much higher than the load on components in a LAN. The potential users of an application at a local site number at most a few hundred to a few thousand, and can be handled by a few fast computer systems. The number of potential users of a wide-area application is orders of magnitude larger – we can conservatively estimate Internet users in the millions, and they continue to increase. While the load on many wide-area services does not scale with the number of users, systems such as the Domain Name Service and Usenet show that there are some applications where load is of concern.

Sending a message between any two hosts on an internetwork requires a variable amount of time, termed the *communication latency*. The communication latency of a message depends on the load on the network and the performance of the available routes between hosts. Hosts can communicate quickly with other hosts on a local Ethernet segment, while communications with more distant hosts take longer because they must pass through several gateways. Internetworks are *unreliable*, meaning they lose and duplicate messages from time to time, and they may deliver them out of order. Hosts sending a message can use timeouts and acknowledgments to detect with high probability that a message has not been received. They cannot distinguish whether a message has been lost due to network or host failure. Systems generally fail for short periods of time, returning to service after only a few minutes or hours.

In a wide-area internetwork, communication latency can be the predominant factor determining the latency of operations on the replicated data. Operations on replicated data often require only a few milliseconds, involving a few disk accesses or a fairly small amount of computation. In a local-area network the time required to communicate with a replica is also on the order of a few milliseconds. In contrast, access times for replicas on an internetwork are non-uniform, and are greater, often several hundred milliseconds scheme for identifying hosts and routing packets of data between them. More sophisticated communication protocols are layered atop the unreliable datagram mechanism.

The replicas are built using a similar layered model. Within a replica, the *storage* layer maintains a copy of the data. The data are modified according to requests issued by the *replication protocol* layer, which contains protocols that complement those in the application's replication protocol layer. This layer also contains replica-specific protocols such as failure recovery and background update. The replication protocols are once again implemented in terms of communication protocols and network services.

This model is similar to many other layered application models. In particular, it can be compared to the ISO OSI reference model [Tanenbaum81]. Figure 1.1 shows the correspondence between the two.

TABLE 1.1: Correspondence between four-layer and OSI models

Four-layer model	OSI model
Application	Application
Replication	Presentation
Communication	Session
Network	Transport

1.3 Performance measures

The goal of this thesis is to develop communication protocols that facilitate efficient and convenient implementations of replication protocols in an internetwork. There are several measures that can be used to evaluate replication performance, depending on the environment and application. These include the latency of operations, the maximum operation rate (throughput), the amount of message traffic caused by an operation, the overall reliability, and the data availability. Though the ideal protocol would provide the highest reliability, availability, and throughput with the lowest cost and latency, tradeoffs between these measures have to be made. The scale of the Internet further complicates

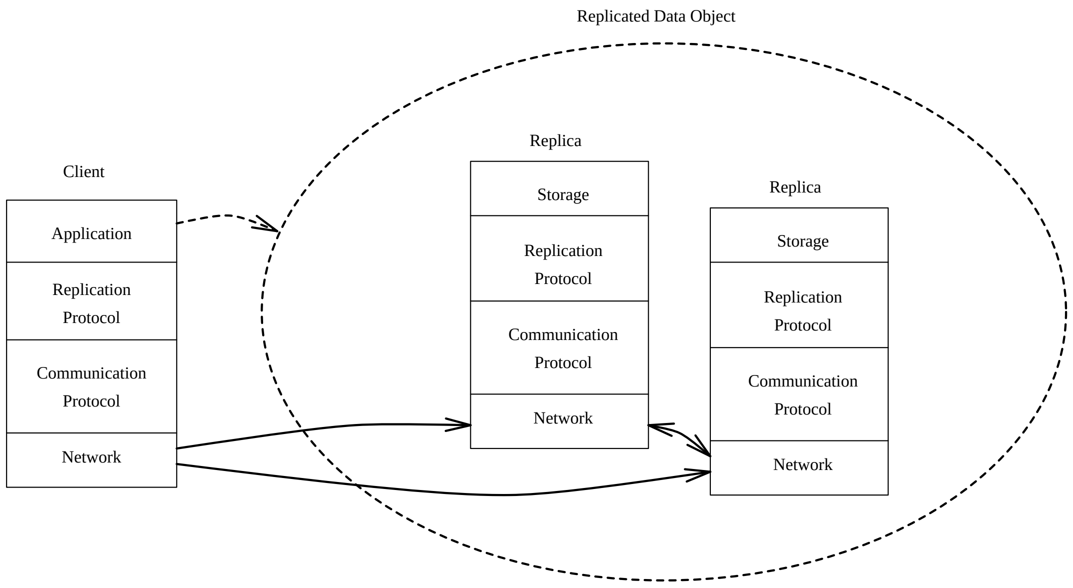

FIGURE 1.1: Communication layers for data replication.

The replication protocol provides the single-copy illusion by determining the replication policy and controlling the distribution of information between systems. Many different algorithms have been proposed for this layer, and their performance is well understood. Chapter 3 presents several of these replication protocols.

The replication protocol in turn uses the services of the communication layer to send messages to replicas. The communication layer provides communication mechanisms, including multicast datagrams, remote procedure call, and bulk transfer protocols. The multicast datagram mechanism sends a short message to all available replicas efficiently. The bulk transfer protocols are more efficient for transmitting large amounts of data from one system to another. This layer determines whether communication is successful or not. In this thesis I will concentrate on multicast protocols at the communication layer, as detailed in Chapter 4.

The communication protocol is in turn dependent upon the network communication services provided by the host. At the lowest level, hosts communicate by sending unreliable datagrams using a protocol such as IP [Comer88]. This protocol only provides a the replicas to read or update the data. Communication between the client and the replicas is performed according to a *replication protocol* that provides the client with the illusion of a single data object. This is more complex than using a single copy of the data, since operations must be coordinated among the replicas.

Replication protocols generally provide a set of operations performed by clients and a set performed by replicas. The operations clients can use include reading and updating the data. Replicas can perform operations such as failure recovery, creation of new replicas, and background information transfer.

Both the clients and the replicas reside on *hosts*. All hosts are connected using an internetwork that consists of local-area networks with *gateways* and *point-to-point links* connecting them. When a host sends a message to another host, the message might first be transmitted on a local Ethernet to a gateway, which might forward the message across a point-to-point link to another gateway, which might then forward the message through several other links and gateways until it reached a network where the destination host is connected.

1.2 A layered implementation model

Distributed applications – at least those that use replicated data – can be built using a four-layer model of implementation. *Applications* and *replicas* use a *replication layer* to manipulate shared data. The replication layer is in turn built on the *communication layer.* The communication layer uses the services of the *network layer* to send messages between clients and replicas. Figure 1.1 illustrates this abstract model.

The application perceives a single highly-reliable object, and does not observe the details of replication. The replication layer provides the application an abstraction of a single copy of the data. Different replication protocols may provide different consistency guarantees to applications, but the interface is always of a single (albeit possibly inconsistent) data copy. The application can issue requests to the replication layer to perform operations on the data.

quorum multicast mechanisms, and how well this works. The performance evaluation is based upon measurements of the Internet, and uses simulation and direct measurement techniques.

1.1 Background

To answer the question *"how do we build a wide-area distributed application?"* I must first define what such an application is. Two distinguishing characteristics are that such applications involve the cooperation of multiple distinct processing systems, and that these systems share information by message-passing. If multiple systems are not applied to the problem, or if the systems communicate through shared memory, then well-known techniques for sharing information in a single system can be applied. If the systems are not sharing information, then there is no difficulty because each system can act autonomously.

There are several reasons why people build distributed applications on a wide-area internetwork. The primary reason is that the application might be used to share information between geographically-distributed users or between users in different organizations, where no centrally-administered system can be constructed. Several such systems already exist, such as electronic mail, news, and name services.

When systems dispersed over a wide area are to have access to the same data, that data can be *replicated*. Several sites around the connecting network maintain copies of the replicated data. Replicated data can be more fault-tolerant than unreplicated data, and its use can improve performance by locating copies of the data near to their use. Many existing wide-area systems, such as airline reservation systems and library card-catalogues, do not use replication techniques, relying instead on large central servers. This is in part due to the apparent inconvenience of replication as compared to centralized solutions, and to the perceived poor end-to-end performance of replicated data.

Copies of the replicated data are held at a number of *replicas*. A replica consists of some storage and a process that maintains the data copy. A *client* process can communicate with

Chapter 1

Introduction

Exactly how does one build a reliable distributed application on the Internet? Many of the necessary pieces are available: replication protocols, internetwork communication protocols, protocols to handle distributed agreement, and so on. The problem is how to put the pieces together, and how to tell if the result is a good one.

This thesis is the result of trying to assemble some of the pieces. There are a few basic lessons I learned doing so: that a clear definition of *failure* is necessary; that distributed applications can be *composed* from pieces built according to a simple architectural model; and that there are well-defined *measures of performance* that can be used to evaluate an application. In addition, two principles have become clear: that a distributed operation should use as small a portion of the Internet as possible, and that applications should respond to changes in the Internet environment.

The concrete result of this effort is a family of what I term *quorum multicast* communication protocols. This family provides a communication mechanism similar to traditional multicast protocols, which send a message in parallel to a set of destinations. Unlike other multicast protocols, the quorum multicast protocols communicate with just a *subset* of the set of destinations. The quorum multicast protocols can dynamically select this subset to be the closest available destinations, limiting the portion of the internetwork affected by any particular multicast.

This thesis starts with a simple layered model of replication, then examines some distributed applications. I will demonstrate how replication protocols can be built using

peek at these protocols a year ago. He is the force behind the *refdbms* database that has proved an invaluable research tool. And, of course, it is his clear and concise style of writing I try to emulate. Darrell Long, my advisor, got me through the time when I marched into his office to announce I'd had enough and was going to earn a living. He has provided my research with a rigor I now appreciate (though I didn't when he first started).

This research was supported, in part, by a University of California seed grant, and by the Concurrent Systems Project at Hewlett-Packard Laboratories.

This thesis is dedicated to four people who got me here: to my parents, Richard and Julia Golding, for introducing me to reading and learning at a young age, and for providing me a home where such pursuits were rewarded; to Douglas Frick, my long-time friend, for years of hiking, diving, and long walks; and to George Neville-Neil, for his matzoh-ball soup, his maniacal laugh, and his love.

Acknowledgments

A great many people have contributed to this thesis, directly and indirectly. I could not have done the final measurement experiment described in Chapter 7 without the help of all the people who took some of their time and machine resources to run the test daemon. They are listed in that chapter. Jim Hayes at Apple Computer provided me invaluable assistance through long talks about networks and about life. He gets credit for asking me the question, “So why don’t you just do it better?” Matthew Lewis at the Universiteit van Amsterdam provided encouragement when it was sorely needed and a wealth of knowledge about the state of the Internet in Europe. Jehan-François Pâris at the University of Houston pointed me to the work on escrow accounts in Chapter 3. Calton Pu at Columbia University provided me with copies of his work on Epsilon Serializability, and provided useful feedback on an early report on this work.

Several people gave me important nudges along the way. Robin Albrecht, at Mentor Graphics, David Brown, at Microsoft, David Mason, at Fairhaven College, Jim Murphy, at CSU Chico, and George Neville-Neil, at UC Berkeley, all helped me mull around the ideas, out of which foment this thesis arose. The other residents of the Concurrent Systems Lab at UC Santa Cruz have provided a congenial atmosphere to work in. The members of the Concurrent Systems Project at Hewlett-Packard Laboratories have been stimulating, argumentive, and are full of good advice about lots of things.

The three members of my committee have helped life along in a great many ways. Kim Taylor has been fun to talk with, and is a seemingly inexhaustible resource for theoretical systems work. John Wilkes at HP Labs provided me the time and funding to do an initial

Accessing replicated data in a large-scale distributed system

Richard Andrew Golding

ABSTRACT

Many distributed applications use replicated data to improve the availability of the data, and to improve access latency by locating copies of the data near to their use. This thesis presents a new family of communication protocols, called *quorum multicasts*, that provide efficient communication services for replicated data. Quorum multicasts are similar to ordinary multicasts, which deliver a message to a set of destinations. The new protocols extend this model by allowing delivery to a subset of the destinations, selected according to distance or expected data currency. These protocols provide well-defined failure semantics, and can distinguish between communication failure and replica failure with high probability.

The thesis includes a performance evaluation of three quorum multicast protocols. This required taking several measurements of the Internet to determine distributions for communication latency and failure. The results indicate that the behavior of recent messages is a useful predictor for the performance of the next. A simulation study of quorum multicasts, based on the Internet measurements, shows that these protocols provide low latency and require few messages. A second study that measured a test application running at several sites confirmed these results.

List of Figures

Contents

UNIVERSITY OF CALIFORNIA

SANTA CRUZ

Accessing replicated data in a large-scale distributed system

A thesis submitted in partial satisfaction
of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER AND INFORMATION SCIENCES

by

Richard Andrew Golding

June 1991

The thesis of Richard Andrew Golding is approved:

Darrell Long

Kim Taylor

John Wilkes

Dean of Graduate Studies and Research