Simulating Wide-area Replication

Sape J. Mullender
Martijn van der Valk
University of Twente, P.O. Box 217, 7500 AE Enschede, Netherlands
E-mail: `martijn@pegasus.esprit.ec.org`

Abstract

We describe our experiences with simulating replication algorithms for use in far flung distributed systems. The algorithms under scrutiny mimic epidemics. Epidemic algorithms seem to scale and adapt to change (such as varying replica sets) well. The loose consistency guarantees they make seem more useful in applications where availability strongly outweighs correctness; e.g., distributed name service.

1 Introduction

Traditionally, replication has been an important area of research in distributed systems. Most of the existing work concerns small-scale (LAN) interconnects [Black87, Dixon89]. In recent years, replication of data across an interconnect that spans continents is becoming ever more popular. As an example, every time one clicks on a hyperlink in a World Wide Web browser, the corresponding data is replicated in a cache on your local disk.

The success of the Web [Berners-Lee94a] lies partly in the simplicity of its protocols. However, simplicity comes at a price; e.g., *http* does not cater for coherency control between source and replica. Also, with the appearance of commercial web servers, replication of data becomes important to increase availability. URLs do not work well with replicated data.

This paper argues that naming, being a fundamental activity of all distributed applications, is a prime candidate for a particular branch of replication algorithms that continue to work well if they are heavily replicated on a wide-area network such as the Internet.

2 Local vs Wide-area Replication

Problems concerning replication are fundamental to distributed computing as a whole. These include maintaining a shared state between a number of nodes that fail independently. To exchange updates, nodes that share a replicated datum should be able to refer to one another. References may break because of transient failures or because nodes decide to give up a datum. The way in which broken references may be re-established differs in local and wide areas. In the local area network, the use of global search as a last resort for mending broken links seems to be taken for granted by the distributed systems community [Tanenbaum90, Shapiro92]. The success of hardware-support broadcast in Ethernet seems responsible for this. However, this assumption will prove untenable in switch-based networks such as ATM, especially in large interconnects.

Conclusion: Broadcast as a mechanism for locating objects does not cover a wide area and therefore cannot be used to perform global search.

3 Naming

Some form of distributed naming mechanism sits at the bottom of all distributed applications. Therefore, naming will benefit immensely from a tailored implementation of wide-area replication.

Again, consider the Web. Data on it are referred to by URLs. URLs identify the *location* of data; not their contents or type. Following his definition laid out in [Needham93], URLs are *impure* names. Impure names, as contrasted with pure names, are subject to change when the data they refer to is moved, or replicated. Especially replication is bound to become a feature of many commercial sites that wish to increase their availability around the Internet. Name resolution failures due to change are annoying for all that want to refer to data with any degree of permanency (e.g., users!)

Web researchers have realised the need for a different form of naming, URNs, which do not have this disturbing property. This work is still in progress, and up to now, URNs have not been defined; see [Berners-Lee94b]. We believe that for naming to be efficient, names have to be impure to some extent, so they may guide the name resolution. Efficiency can be realised, even in the face of replication using the following scheme. Instead of pointing to the node managing an object replica, as URLs do, names should really be pointing to the name service *directory* storing the links to all object replicas. The extra level of indirection provides the necessary and sufficient impurity to guide search whilst retaining a unique name for a replicated datum.

Conclusion: The underlying naming facilities of wide-area applications, such as the Web, are greatly enhanced with support for replication and migration of data.

4 Implementation Considerations

As an example of wide-area replication, we consider algorithms for use in a widely distributed name service.

Wide-area replication or far-flung replication is very different from the local case. To scale, coherency control cannot rely on efficient multi-cast. Further, transient communication failures make it long-winded and tedious to apply traditional methods, such as weighted voting [Gifford79] or three-phase commit [Ceri84].

An approach that has been followed is to use a layer of replication strategies, on a spectrum ranging from unreliable but efficient to reliable and slow. The first is used to distribute updates as best one may, and the latter to fill in the holes [Lampson86, Ma92]. Also, different strategies could be applied to different classes of updates: changes to the set of replicas are considered more important than new data values and and hence are distributed in a stronger order [Ladin92],

Since a name service sits at the bottom of any distributed application, high availability is our prime concern. The *order* in which updates are applied to the database is of less concern; global time stamps are used to provide a consistent shared state *eventually* [1], at a low cost. To provide causal order, vector time stamps would be needed [Ladin92]. Their size is substantial, and subject to the current replica set. Also, to maintain them, more shared state is introduced. To provide total order, a *sequencer* would be needed [Kaashoek89], whose single point of failure reduces availability.

We are investigating the use of *epidemic* algorithms — so called because their behaviour mimics *virii* spreading infectious disease — for distributing updates among the nodes of a highly-available name service. The algorithms are partly randomized, have simple behaviour and keep little state, yet are capable of distributing updates fairly rapidly among a large percentage of the nodes. The usefulness of epidemic algorithms was reported in [Demers87], as part of the Grapevine [Birrel82] project. Again, they were intended to be backed up by a separate mechanism that was to provide the necessary robustness.

5 Simulating Far-flung Replication

To test the usefulness of epidemics, we have simulated their behaviour in a large scale system made up of potentially hundreds of nodes connected by a network. The nodes conspire to manage a replicated directory. Each node acts as a server to clients that continually generate updates to the directory's contents. It is a node's job to further updates to its peers.

The simulator we built uses a queueing system which runs under virtual time. At the beginning, the simulator is fed with a sequence of injections, the node-to-node round trip delays, and the epidemics type. Each injection describes an update to a specific node at a specific time. It gets translated into an event by the simulator. The round trip delays are stored as weights in an incidence graph depicting the interconnect. The epidemic is specified by its type and a type-specific number of parameters [2]

[1] Following Needham, when no more updates are made, all replicas will eventually have the same value.

[2] For the two-button crowd who shuns complex command line arguments, a Tcl/Tk GUI is available.

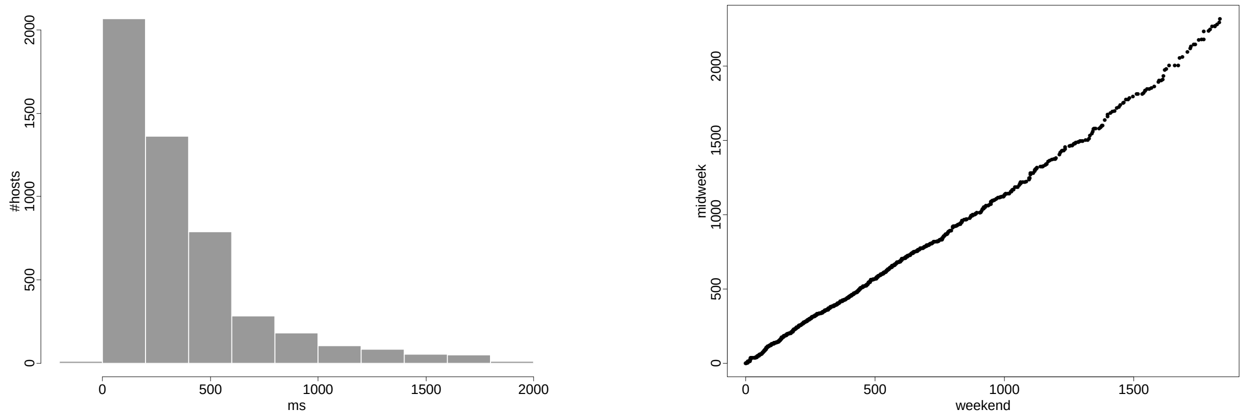

Figure 1: Average round-trip delays: histogram and QQ-plot

Once the simulator takes off, all activity is controlled by the events that it schedules. Update events trigger epidemics, whose actions are also implemented by events. Each event is associated with a time stamp that tells when it should be handled. The run-time simply advances its virtual clock up to the earliest event. Simulation ends when there are no more events to process or when a specified time has been reached.

At specified times, the run-time emits part of its internal state for analysis. It turned out that simulating and analysing software had best be split, because of different goals: to make simulation fast, we use ANSI C. However, this decision also means that the code is relatively inflexible. Yet, during analysing, one often needs to filter out uninteresting data or magnify the good bits. Therefore, analysis should be done using a language that allows rapid adjustments. Python was a good choice [VanRossum93]. As it is interpreted, it is very flexible. Yet, its built-in type generators (lists, hash-tables, &c.) are fast enough for our purposes.

To forego the need for a shared data format that has to be marshalled by the simulator front end and unmarshalled by the analyser back end, we decided to have the former *generate* most of the latter: the simulator emits Python code that forms most of the analyser. When run, this imports the generic analyser code through Python libraries. The output of the analyser can be run directly through `gnuplot` or `Splus` [Becker88], to create the graphs in this paper.

5.1 Round-trip Delays

It is impossible to foresee all the kinds of future applications that will make use of far-flung replication. At present, the only existing candidates would be (selected) Web servers and their clients. To be useful, we have to simulate likely node-node round trip delays; i.e., that correspond to client/server interactions.

To simulate node-node round trip delays that match those on the Internet, we have pinged a large number of hosts and recorded the average round trip delays. The hosts comprise a subset of over 5000 hosts scattered across the Internet and were picked from our local `httpd` server log.

Figure 1 shows a histogram and quantile plot of the average round trip delays. We have repeated our pinging experiment at different times during the week and their relative distributions match: The quantile plot relates test data sets generated in the weekend and during mid week.

Quantile analysis would seem to indicate that our test data sets may be exponentially distributed. Although, for lack of time we have not yet conducted a proper statistical analysis, we are confident enough to use exponential random variables for round-trip delays in our simulations.

It is important to realise the implicit assumptions that we have made in this respect:

- All nodes on an interconnect share similar round trip delays with their peers[3]

[3] Similarity is understood in terms of probability distribution functions.

- Round trip delays are similar regardless of the time of day.

Without a global joint effort (i.e., many people pinging in harmony), the first assumption cannot be verified. With the exception of [Long95], we have not been able to find many joint efforts to measure properties of a great many hosts on the Internet simultaneously. As far as the second assumption is concerned, our experience with weekend and mid-week pinging does not contradict it.

6 Simulations

The number of different types of epidemics that our run-time can simulate is quite large, due to the sizable number of parameters that can be used to tweak their behaviour. Our experiments investigate only some of them. Our main focus of attention is on the so-called push/pull epidemic which seems to work best in most circumstances.

Each experiment consists of a number of runs using random network configurations and injection patterns. The result is taken as the average of all runs.

The results of each experiment are stated in a number of graphs and tables. The graphs plot the average number of octets needed to achieve a certain infection rate against the passage of time. The infection rate is understood as the number of node infections divided by the number of injections in the system. The table summarises the end state of each run.

6.1 Push vs Pull

We have tested the merits of five epidemics using identical network configurations and injection patterns. *Pushing* means that an update is spread by nodes that know it (i.e., are *infected* by it). *Pulling* means that *any* update is spread by asking for it by *any* node. A node initiates either pushing or pulling when the conditions for it are true. One of the conditions depends on how often it has initiated an infection attempt previously. In a *feedback* epidemic, the node that initiates an infection attempt is made aware of the success of its action, and bases its future decisions on this outcome. If a node is *blind*, it has no such knowledge and is forced to take a random decision. Naturally, conveying this information over the interconnect increases the message overhead.

An important distinction between pushing and pulling is that the message overhead of pushing is determined by the update frequency, whereas with pulling it is not. A combination of the two works very well: infected nodes push updates whereas every node still occasionally pulls for new updates.

Figure 2 shows the results of two experiments varying the mean time between events (mtbe). The x-axes of all graphs depict virtual time. The y-axes on the upper two graphs show the amount of data that is needed to spread an infection. These numbers are not to be taken as absolute values but rather allow different epidemics to be compared in the amount of overhead they introduce. The ramp-up effect in the amount of overhead is explained by the fact that initially not many injections have occured so most infection attempt messages carry information about only a single update.

The lower graphs depict how many of the nodes at any one time have been infected by all injections. Let n be the size of the replica set. Let $u(t)$ and $i(t)$ respectively be the number of updates to the system (injections) and the number of succesful infections upto time t. Then the y-axis depicts $i(t)/(u(t) \times n)$.

Due to lack of space, we will limit our discussion in the remainder of this Section to combined push/pull epidemics.

6.2 Varying the Number of Nodes

To show that epidemics scale well, we have run the same epidemic among a increasing number of nodes. We found that the message overhead and infection penetration converge when the number of nodes increases above ten. Figure 3 shows the results: an increase in the replica set causes a negligible increase in the data overhead. Also, infection ratios are barely affected.

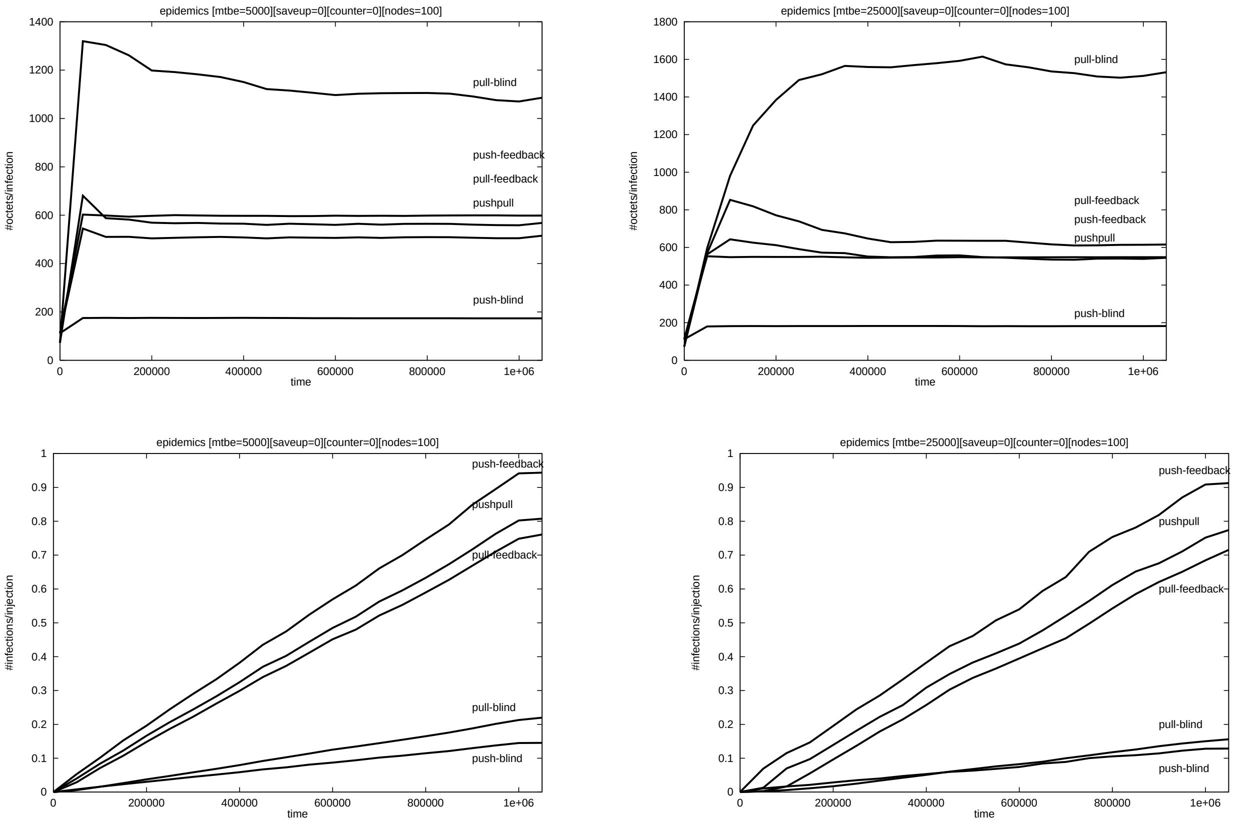

Figure 2: Epidemics compared

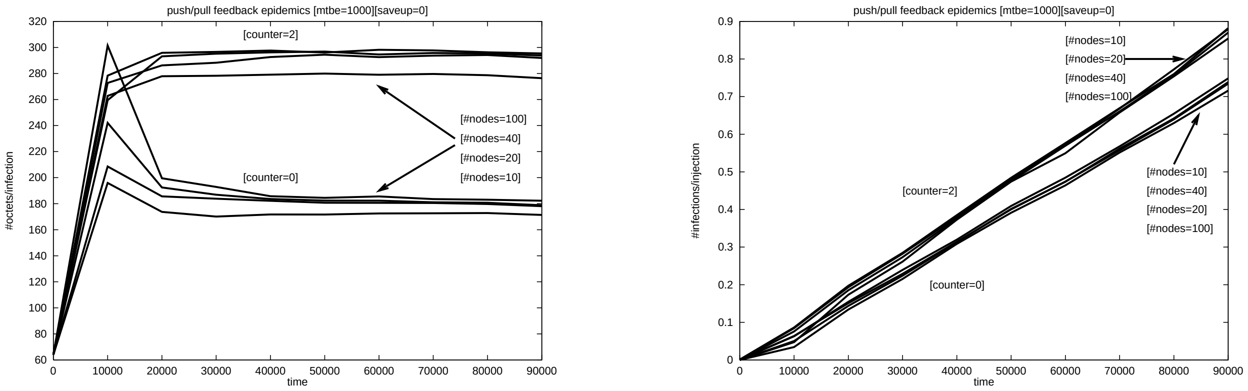

Figure 3: Varying the Replica Set Size

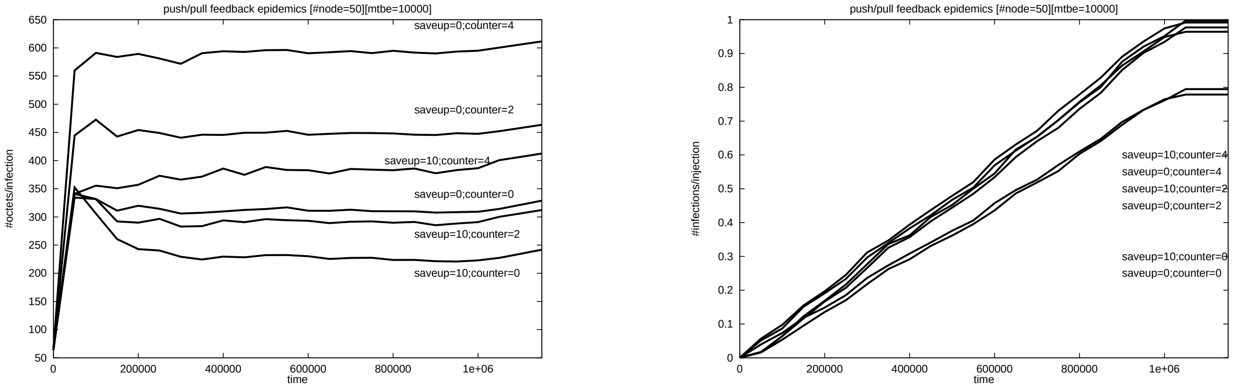

Figure 4: Fine-tuning Epidemics Parameters

#saveup	**counter:** 0		**counter:** 2		**counter:** 4	
	#octets	**infected**	**#octets**	**infected**	**#octets**	**infected**
0	328.879839	0.778551	463.464291	0.964263	611.466449	0.991716
10	241.899529	0.794723	312.269647	0.977178	412.445986	0.997475

Table 1: Fine-tuning Epidemics Parameters

6.3 Varying Conditions

An epidemic can be tweaked by altering the conditions under which it is initiated by a node. These experiments vary three of these: *counter*, *save-up* and *depth-influence* thresholds.

The counter threshold is a condition in deciding whether a node will continue an epidemic: every (unsuccessful) infection attempt increases a counter. The threshold determines when the node is to give up. A high threshold results in a higher infection penetration at the cost of increased message overhead.

The save-up threshold postpones an infection attempt until a node has a certain number of infections to spread or until a time in the future has been reached. A high threshold results in a lower message overhead by combining many infections in a single message at the cost of delaying infections.

The results of experiments in which these parameters were varied orthogonally are in figure 4. As some of the plots in the graphs are almost indistinguishable, we also provide the raw numbers that hold at the end of the simulations in table 1.

To limit the number of superfluous infection attempts, a number is included in each infection attempt: the infection depth. When an infection is injected, the number is zero. With each subsequent infection attempt by a newly infected node, the depth is increased. When a node becomes infected, a high depth value decreases its determination to infect others: as it was late to be infected, most of the others will probably already be infected with this update. The depth influence threshold determines the importance of the depth value in curbing new infection attempts.

Figure 5 shows the results of an experiment in which we vary the influence of the depth value: as it increases, the number of average octets needed to spread an update decreases, with only a negligible reduction of the infection rate. As before, the raw numbers are in table 2.

6.4 Conclusion

The ability to adapt to a changing environment (replica sets, update frequency) is a characteristic that makes epidemics well-suited for use in replication in the large. It is tempting to develop epidemics that behave well enough that no heavyweight backup algorithms are needed. We are led to believe that this is indeed possible by combining several

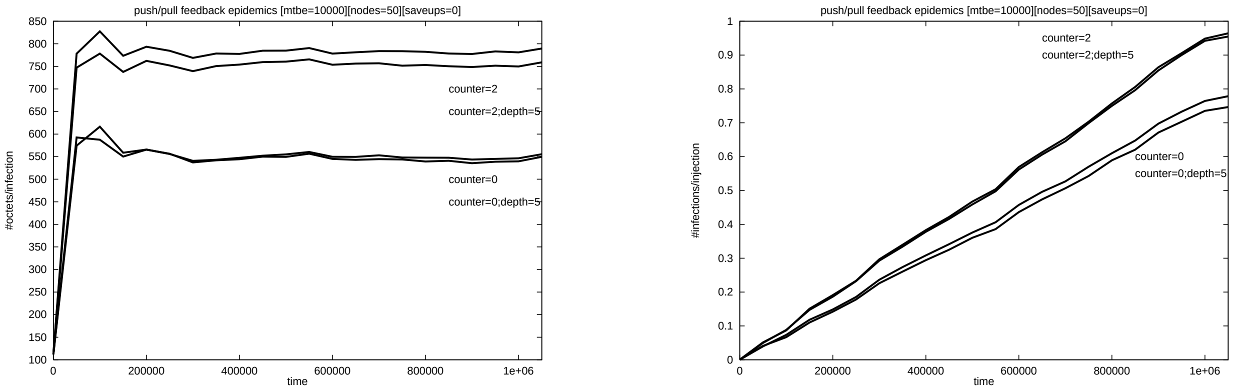

Figure 5: Infection Depth

depth	**counter:** 0		**counter:** 2	
	#octets	**infected**	**#octets**	**infected**
–	555.417258	0.778510	789.565054	0.964263
5	549.640441	0.746640	758.880160	0.954921

Table 2: Infection Depth

kinds of existing epidemics with slightly altered characteristics. As an example, epidemics must trade communication overhead against the feasibility of infecting all nodes in a changing replica set. A way out of this dilemma is to make uninfected nodes pull for updates, and to introduce *dormant* infections, which are inactive until two nodes find their respective databases out-of-sync. An MD5 [Rivest92] ticket is a small (16 bytes) and efficiently computed digest of a database value, and can be used for comparison.

7 Acknowledgement

Mrs. Yeung Siu Lan's help with understanding the Splus language proved indispensable.

References

[Becker88] Richard A. Becker, John M. Chambers, and Allan R. Wilks. *New S Language: A Programming Environment for Data Analysis and Graphics.* Wadsworth, 1988.

[Berners-Lee94a] T. Berners-Lee, R. Cailliau, A. Luotonen, H. F. Nielsen, and A. Secret. The WorldWide Web. *Communications of the ACM*, **37**(8):76–82, August 1994.

[Berners-Lee94b] T. Berners-Lee. Universal Resource Identifiers in WWW. Request for Comments 1630. ARPA Network Working Group, June 1994.

[Birrel82] Andrew D. Birrel, Roy Levin, Roger M. Needham, and Michael D. Schroeder. Grapevine: An exercise in distributed computing. *Communications of the ACM*, **25**(4):260–74, April 1982.

[Black87] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering*, **SE–13**(1):65–76, January 1987.

[Ceri84] Stefano Ceri and Giuseppe Pelagatti. *Distributed Databases; Principles and Systems*, Computer Science Series. McGraw-Hill International Editions, 1984.

[Demers87] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. *Proceedings of the* 6^{th} *Annual ACM Symposium on Principles of Distributed Computing*, pages 1–12. ACM, August 1987. Order no. 536870.

[Dixon89] G. N. Dixon, G. D. Parrington, S. K. Shrivastava, and S. M. Wheater. The treatment of persistent objects in Arjuna. *Proceedings of ECOOP '89*, July 1989.

[Gifford79] D. K. Gifford. Weighted voting for replicated data. *Proceedings of the* 7^{th} *Symposium on Operating System Principles* (Pacific Grove), pages 150–61. ACM, 1979.

[Kaashoek89] M. Frans Kaashoek, Andrew S. Tanenbaum, Susan Flynn Hummel, and Henri E. Bal. An efficient reliable broadcast protocol. *Operating Systems Review*, **23**(4):5–19, October 1989.

[Ladin92] Rivka Ladin, Barbara Liskov, and Sanjay Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems*, **10**(4):360–91, November 1992.

[Lampson86] Butler W. Lampson. Designing a Global Name Service. *Proceedings of the* 5^{th} *ACM Symposium on Principles of Distributed Computing*, pages 1–10. ACM, 1986.

[Long95] Darrell Long, Andrew Muir, and Richard Golding. A longitudinal survey of Internet host reliability. Technical report UCSC–CRL–95–16. Computer and Information Sciences Board, University of California at Santa Cruz, February 1995.

[Ma92] Chaoying Ma. Designing a Universal Name Service. Technical report TR 270. University of Cambridge Computer Laboratory, November 1992.

[Needham93] Roger M. Needham. Names. In Sape J. Mullender, editor, *Distributed Systems*, Frontier Series, pages 315–27. Addison-Wesley Publishing Company, New York, 1993.

[Rivest92] R. Rivest. The MD5 Message-Digest Algorithm. Request for comments 1321. ARPA Network Working Group, April 1992.

[Shapiro92] Marc Shapiro, Peter Dickman, and David Plainfossé. SSP chains: robust, distributed references supporting acyclic garbage collection. Technical report 1799. INRIA, November 1992.

[Tanenbaum90] A. S. Tanenbaum, Robbert van Renesse, Hans van Staveren, G. J. Sharp, S. J. Mullender, A. J. Jansen, and G. van Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, **33**(12):47–63, December 1990.

[VanRossum93] Guido van Rossum. An Introduction to Python for UNIX/C Programmers. *Proceedings of the NLUUG* (The Netherlands), November 1993.