Proceedings of the 1990 Winter Simulation Conference
Osman Balci, Randall P. Sadowski, Richard E. Nance (eds.)

TIME-BASED PERFORMANCE EVALUATION OF PROTOCOLS FOR DISTRIBUTED SYSTEMS: AN ALTERNATIVE TO MESSAGE COUNTING

Phil Kearns

Department of Computer Science
The College of William and Mary
Williamsburg, Virginia 23185

Jeffery E. Payne

BDM International, Inc.
1300 N. 17th St. #950
Arlington, Virginia 22209

ABSTRACT

Protocols for distributed systems are complex due to their intrinsic concurrency and asynchrony. Given two protocols for a certain task, a systems programmer is faced with the decidedly non-trivial task of evaluating the performance of the alternatives in a specific system context. A scan of the literature generally results in "performance evaluation" which is expressed as asymptotic worst-case number of messages required for implementation of a protocol (or, equivalently, asymptotic worst-case number of bits transmitted). We contend that, in many practical contexts, such message counting is not particularly informative. We present a performance evaluation of two protocols for distributed first-come-first-served service. This study utilizes GENSIM, a general purpose facility for simulating distributed systems. The performance metric of choice is real time, a natural metric which is easily handled by simulation but which is only tangentially related to message counts.

1. INTRODUCTION

A distributed system consists of a group of relatively autonomous computing *nodes* connected by *links* of a communications *network.* A computation on this system is viewed as a group of *processes* which are assigned to, and executed on, the nodes. In general, the only mechanism by which two processes may communicate is message-passing: we cannot assume the existence of shared memory between any two processes. Further, and significantly, we make no assumptions about the relative speeds of processes executing on the distributed system and the speed of message transmission on the various links. There is no global physical clock accessible to the nodes of the system. However, we do assume *well-ordered message passing* between all pairs of processes: messages from a source process to a destination process arrive in the order in which they were transmitted.

The degree of concurrency in such a system presents a real problem for performance analysts in that a single event and several events simultaneously performed can take the same time. For example, in a point-to-point network, the transmission of a single message over a single link may be done concurrently with other transmissions on other links. In a broadcast medium network, a single broadcast message can be delivered physically to all nodes in roughly the same amount of time as a single message from one node to another. The degree of concurrency is a difficult statistic to produce by analytic means. It depends on explicit, deterministic interactions among processes. As a result, protocols for distributed systems are often evaluated in asymptotic worst-case terms. Message counts, or counts of the number of bits transmitted in messages, are typically used in such analyses. We have found that worst-case analysis may be misleading, but that, given the difficulty of expected-case analysis in the presence of asynchrony and concurrency, it is typically the only analysis available in the literature.

This paper presents two protocols which implement a common facility in a distributed system. Worst-case analysis is shown to be inaccurate in general, and a simple expected-case analysis is shown to be pessimistic in certain network contexts. Simulations of the protocols permit a relatively straightforward performance comparison to be made. An optimization of one protocol, not amenable to worst-case or expected-case analysis, is developed and evaluated.

2. DISTRIBUTED ORDERED SERVICE

Many distributed systems are structured such that a resource is centrally managed by a single designated process, the *server* for that resource. Other processes, *clients* of the server, access the resource by passing request messages to the server. Such a system, consisting of server S and clients C_1 and C_2, is shown in Figure 1.

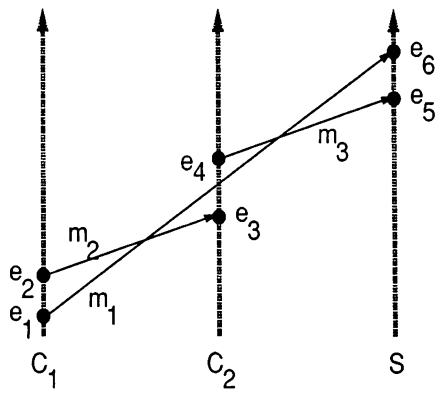

Figure 1. Centralized Resource Management

This system passes three messages. C_1 sends a request for service to S in message m_1. It then sends message m_2 to C_2. After C_2 has received m_2, it sends a request to S in m_3. In our example, S receives m_3 before it receives m_1. This scenario is consistent with the well-ordered message passing assumption in that ordering is required only between pairs of communicating processes. From the perspective of S, two request messages have been received. Service in the arrival order may result in inconsistent behavior of the overall system if components of the system maintain state. The essential cause of this potential inconsistency is m_2. This message may causally affect the computation of C_2, including the nature of the service C_2 requests of S in m_3. We require a protocol which guarantees that service is performed in the order in which requests are made.

Lamport [Lamport 1978] formalizes this problem, and provides a framework for a solution. He views a distributed program in execution as a set of events, $\mathcal{E}$. That set must include all message-passing events and others as appropriate to the application. In Figure 1 the relevant events are depicted by the solid circles on the process time lines; they are labeled e_1 through e_6. Lamport further defines a *partial* order on the set of events

$$\rightarrow : \mathcal{E} \times \mathcal{E},$$

as the smallest relation such that for $e,f,g \in \mathcal{E}$: (a) $e \rightarrow f$ if e precedes f within the same process; (b) $e \rightarrow f$ if e is the transmission of a message and f is the receipt of that message; and (c) if $e \rightarrow f$ and $f \rightarrow g$, then $e \rightarrow g$. It is this partial order which naturally defines "before" and "after" in a distributed system: e is before f if and only if $e \rightarrow f$. Events which are incomparable under the partial order are *concurrent*.

Given the partial order, it is clear that $e_1 \rightarrow e_4$ in Figure 1. Thus the service request conveyed in message m_1 is made before the request conveyed in message m_3. S needs more information before it can decide to serve a request.

2.1 Logical Timestamp Protocol

$T(e)$ is the *logical timestamp* associated with an event $e \in \mathcal{E}$ if $T(e)$ is a natural number, and for $e,f \in \mathcal{E}$, $e \rightarrow f \supset T(e) < T(f)$. Several techniques for implementing logical timestamps have appeared in the literature [Lamport 1978; Ricart and Agrawala 1981]. The essence of the technique is that each processor maintains a local integer, its local logical clock. It increments that clock between all local events. It also appends the logical clock value at the time of a **send** to the message transmitted. On receipt of a message, a process is obligated to set its local logical clock such that it is greater than the minimum of its current local logical clock value and the time value appended to the message.

If the distributed system implements logical timestamps, the protocol for distributed ordered service is relatively straightforward. A client request must be logically timestamped. Let $M =< T, i, R >$ denote the composite message where T is the timestamp, i is the index of the client making the request, and R is the request itself. Upon receipt of M, S must assure that there is no message $M' =< T', j, R' >$ in transit, where $T' < T$. This may be achieved by having S multicast a query to all $C_j : j \neq i$. When a client receives this request, it sends a timestamped acknowledgment, an ACK, to S. The server may honor the request conveyed in M if it has the smallest timestamp of all unprocessed requests known to S, and if a message with a timestamp greater that T has been received from all nodes in the system. A detailed presentation of such a protocol may be found in [Schneider 1982].

Figure 2 shows the protocol applied to our example. The small integers on each process line are the local logical times. The dotted lines denote messages sent and received as part of the query/ACK handshake. Note that when S receives m_3 at time 5, it cannot serve the request since it has not received a message from C_1 timestamped after 4. However, when it receives m_1, it may serve it immediately.

2.2 Immediate Ordered Service

In [Kearns and Koodalattupuram 1989] we present an alternative to the logical timestamp approach to ordered service. That work is based on a generalized timestamp technique called *vector time* [Fidge 1988a,b; Mattern 1988] . A vector time timestamp for a system of N clients is an N-tuple of natural numbers. When a local event of C_i takes place, $T_i[i]$, the i-th entry in C_i's vector must

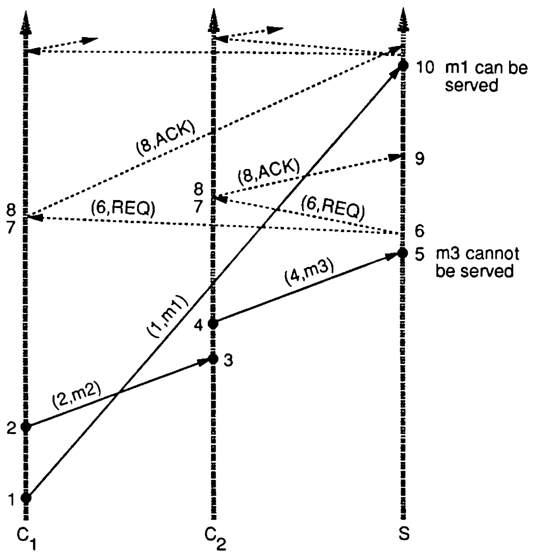

Figure 2. Logical Timestamp Protocol

be incremented. If C_i sends a message, T_i (the current local vector) must accompany the message data. If a client receives a message, it must update its local vector such that each component is greater than or equal to the corresponding component in the vector which accompanies the message.

If we maintain vector time in the distributed system (events are the transmission of server request messages), then the service protocol is somewhat simpler. The server maintains its own time vector. The ith component of the server's vector records the ith component of the vector timestamp on the last request from C_i which was served. Upon receipt of a request message from C_i, if any component of the timestamp of that message, excluding the ith component, is strictly greater than the corresponding component of the server's vector, then the request is deferred. Otherwise, the request is honored immediately. A deferred request will be served when the server's vector is updated, as a result of other services, such the timestamp of the deferred request is component-by-component less than or equal to the server's vector, excluding the component associated with the originator of the request.

Figure 3 shows the immediate ordered service protocol applied to the example. Note the absence of the query/ACK message passing.

2.3 Performance Comparison

We have presented two alternatives for providing ordered service to a centralized resource in a distributed system. A natural question is how the two protocols differ in terms of performance. The logical timestamp protocol, hereafter abbreviated LTP, requires shorter messages in that a single integer suffices as a timestamp as opposed to a vector of N integers. Shorter messages result in faster message transmission. Although the immediate ordered service protocol, IOS, does produce longer messages due to the longer timestamps, it avoids the query/ACK interaction between S and all of the clients on every service request. If a distributed computation generates n_s server requests and n_c client-to-client messages, then a worst-case asymptotic analysis of the number of integers passed in messages in order to implement each protocol is:

LTP: $O(n_c) + O(Nn_s)$, and

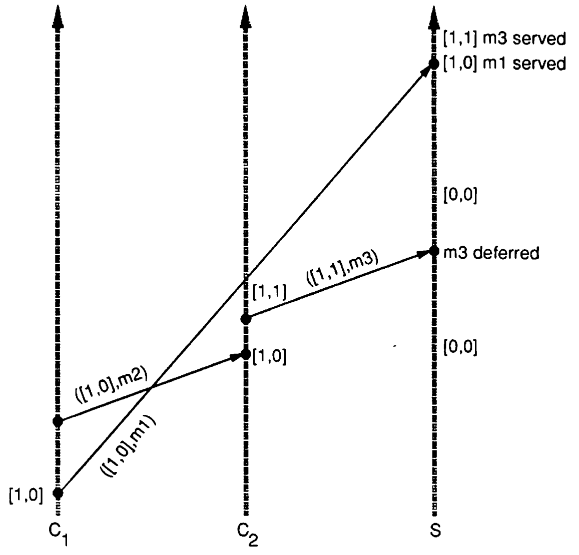

Figure 3. Immediate Ordered Service Protocol

IOS: $O(N(n_c + n_s))$

In these terms, IOS is clearly inferior to LTP. We contend, however, that this analysis is simplistic, particularly if one is interested in performance metrics which are time-based. Response time and resource utilization are two important such metrics. Further, optimizations which accommodate *expected* behavior are simply not amenable to worst-case analysis. It is at this point that the analyst is faced with a choice: analytic performance evaluation which is likely to be intractable or simplistic, given the asynchrony and concurrency in the system, or simulation. In this paper we describe our simulation approach to this problem.

In order to get a feel for the quantitative nature of the relative performances of LTP and IOS, we develop a "back of the envelope" analysis. Service at S is assumed to be instantaneous. The time required to transmit a message of length l from any node to any other is $\alpha + l$: there is a network acquisition latency of α time units, and a message is propagated from sender to receiver at the rate of 1 integer per unit time. Let l_s be the mean length, in integers, of a server request message, and l_c is the mean length of a client to client message.

Our performance metric of interest is the real time expended in message passing. This is a direct indicator of the overhead imposed by a protocol. Let T_I denote the total real message passing time of the distributed computation under IOS, and T_L is the real time of message passing under LTP. We may show that

$$T_L = n_c(l_c + 1 + \alpha) + n_s(l_s + 1 + \alpha) + 2n_s(2 + \alpha),$$

and

$$T_I = n_c(l_c + N + \alpha) + n_s(l_s + N + \alpha).$$

The expression for T_L assumes that the multicast of $N - 1$ queries and the return of the ACKs requires as much time as a single round-trip message transmission. We also assume that a query or an ACK is conveyed in a message data field which is one integer long. Thus T_L is likely to be quite optimistic. The reader is referred to [Kearns and Koodalattupuram 1989] for a more detailed derivation of these quantities.

The condition under which IOS outperforms LTP is $T_I \leq T_L$. Letting

$$\rho = n_s/n_c,$$

we may establish that $T_I \leq T_L$ for networks in which there are N clients, where

$$N \leq N^*(\alpha, \rho) = \frac{\rho(2\alpha + 5) + 1}{\rho + 1}.$$

N^* may be viewed as a threshold: a system with more than N^* clients should use LTP instead of IOS in order to reduce the real time overhead associated with ordered distributed service. Figure 4 shows the predicted thresholds for specified values of α and ρ.

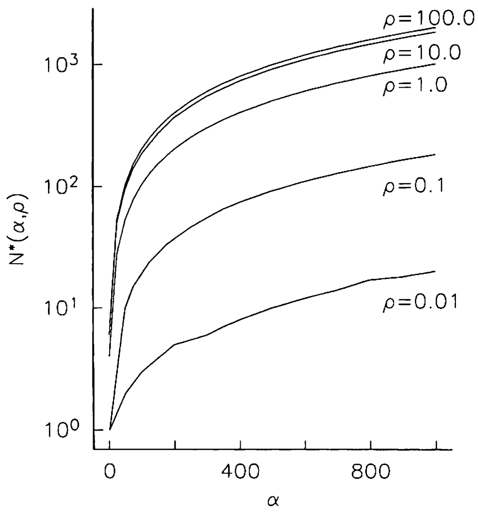

Figure 4. Predicted Network Size Threshold

Figure 4 is illuminating in that it stands in contradiction to the asymptotic worst-case integer counting described above. It gives us some preliminary indication that IOS may, in fact, be a worthwhile protocol for ordered service. This indication is the principal motivation of a detailed simulation-based performance evaluation of IOS and LTP.

We emphasize that the domain in which IOS is predicted to be superior to LTP is not unrealistic. For a value of α near 1000 and ρ near unity, immediate service is superior to logical timestamps for networks of up to about one thousand nodes. In local area networks, we expect the physical transmission of an integer to require about 10 μsec. In 4.2bsd Unix systems, the sendmsg kernel call is the user-level mechanism by which a process transmits a datagram. The CPU time expended on a VAX-11/750 during an execution of sendmsg has been measured as roughly 10 msec [Cooper 1985]. Since the execution time of sendmsg would be a component of α, we may infer that a value of ρ near 1000 would not be unusual for distributed systems consisting of hosts connected by a medium-speed local area network.

3. SIMULATION METHODOLOGY

GENSIM [Payne and Kearns 1990] is a general network simulation facility which models a distributed system at the level of nodes and virtual links between those nodes. Networking considerations at or near the physical layer are not handled in GENSIM. Instead, attributes such as link latency, link transmission speed, link error probability, and a distinction between broadcast networking and point to point networking are supported. Essentially, GENSIM takes the place of the network in that application code which uses message passing to implement the distributed protocol interact with the simulator as opposed to the actual networking interface provided by a host operating system.

In our simulation studies, a distributed computation was characterized by three parameters:

- o ρ,
- o λ, the per-node mean message transmission rate, and
- o N, the number of clients.

A network link between node i and node j is characterized by:

- o $\alpha_{i,j}$, the link-specific transmission latency,
- o $\varepsilon_{i,j}$, the link-specific error probability, and
- o $\tau_{i,j}$, the link-specific propagation rate (integers/sec).

The above quantities are specified as parameters to a run of a simulation under GENSIM. On a given message transmission from a client, ρ determines the relative frequency with which a server message is generated as compared to a client message. If a client message is to be sent, the destination is equally likely to be any other client process. For this study, λ is fixed at 100 messages per second per node (a heavily loaded system). $\alpha_{i,j} = \alpha$ is independent of i and j. We assume a fault-free system. $\tau_{i,j}$ is independent of i and j and is 1.25×10^6 byte/sec. N, ρ and α are parameters of our study. A more general set of simulations are described in [Payne 1989]; however, those results do not differ markedly from what is presented in this paper.

4. RESULTS

Given the indication that the average-case real time delay imposed by IOS may be less than that imposed by LTP, we constructed a series of simulation experiments which test that indication in the presence of more realistic system assumptions. We emphasize that the plots of in Figure 4 show the behavior of an analytically derived $N^*(\alpha, \rho)$. The plots in this section present the results of our suite of simulation experiments.

4.1 Broadcast Network

Our initial experiment is conducted in the context of a broadcast network. A broadcast network is characterized by the fact that only a single message may be in transit at any time. Network access is, in effect, a critical section implemented in hardware. For various values of N, α, and ρ, simulations of IOS and LTP were replicated 1000 times. On a given replication, 500 requests were serviced. The output of each simulation was the total real time of the synthetic distributed computation. The real time outputs were processed to obtain a 95% confidence interval of the difference (time for LTP minus the time for IOS) between the two sets of times. If the 95% confidence interval is a range which is strictly positive, then we are 95% confident that IOS outperforms LTP for a given set of parameters. If one plots the minimal value of N for which the 95% confidence interval is strictly positive for a given value of α and ρ, a graph virtually identical to Figure 4 results. This is not totally unexpected in that a broadcast medium network restricts the degree of concurrent operation in the system. In fact, many of the assumptions which we made in deriving the analytic value of N^* actually hold in this domain.

4.2 Point-to-Point Network

We next turn our attention to an environment in which more concurrency is possible: a point-to-point network in which each link may be active simultaneously. A set of simulations, similar to those described in the previous section, was done. The results are presented in Figure 5. The compression of the lines towards higher

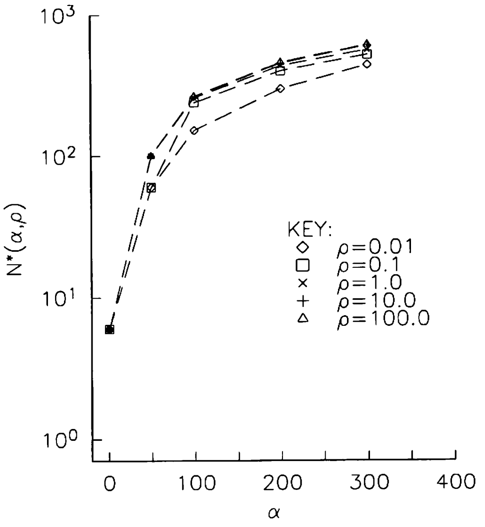

Figure 5. Point-to-Point Network (Heavy Load)

values of N is expected, and is due to the increased concurrency in the system. Several messages in transit on separate links cost only as much, in real time terms, as does a single message. It is interesting to note that all of the lines are tending to lie along the asymptote

$$\lim_{\rho \to \infty} N^*(\alpha, \rho) = 2\alpha + 5.$$

In some sense, the concurrency achieved through the use of multiple links drives the performance of the system towards the optimal. Figure 6 shows the same point-to-point network under a lighter load: λ is one message per second per node. The reduced load results in less opportunity for concurrent message passing and an attendant degradation in performance.

4.3 The Sparse Vector Technique

The obvious optimization of IOS involves reducing the length of the vector timestamps by sending only a list of those components which have changed since the last communication between two nodes. Specifically, if fewer than $N/2$ vector components have changed since the last message from a source to a destination, a list of pairs of integers is sent in place of the vector timestamp. The first integer in each pair denotes a component index; the second is the new value of that component. If at least $N/2$ components have changed since the last communication, then the entire vector timestamp is passed in the message. As such, we can do no worse that the full vector approach. Of course, this technique does not change the asymptotic analysis in the least. In the worst-case, the sparse vector approach is worthless—it is the average case under realistic loading which interests us. Through a set of GENSIM simulations, we produce the plot of Figure 7.

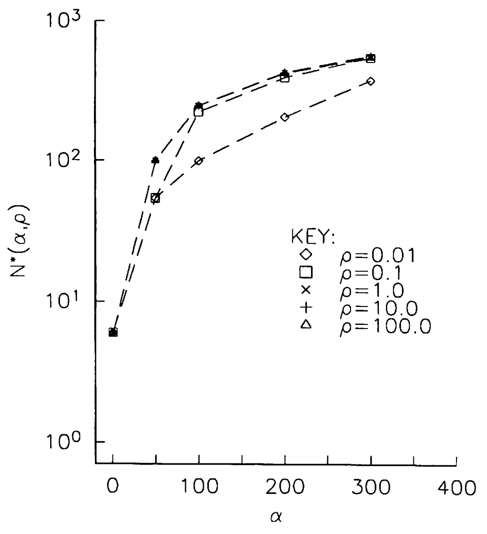

Figure 6. Point-to-Point Network (Light Load)

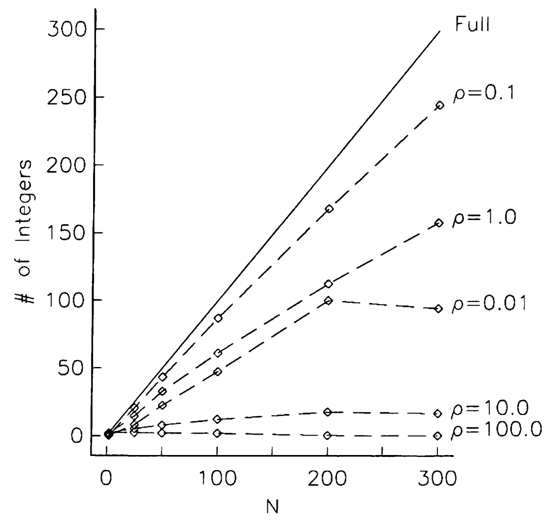

Figure 7. Sparse Vector Size

This plot, derived in the context of a heavily loaded point-to-point network, shows the size of the sparse vector as a function of ρ and N. The value of α has little impact upon this plot [Payne 1989], and its value is unity in the plot presented. Each point in the plot was produced by one replication of a simulation in which 100,000 requests were served. The point is the average size of the sparse vector over all messages passed in the simulation. Clearly, extreme (large and small) values of ρ are handled quite well by the sparse vector technique. It effectively reduces the size of vector timestamps to a fraction of what they would normally be. Hence, it reduces the effective "size of the network" proportionately, making IOS even more attractive.

5. CONCLUDING REMARKS

It is disturbing that the development of many protocols for distributed systems is based upon a performance metric which is the asymptotic worst-case number of messages required to implement the protocol. We have shown that such analysis may be of little practical worth, resulting in false conclusions about the relative performance of competing protocols. We have also shown that expected-case analysis of time-based metrics may be quite pessimistic when a high degree of concurrent activity is possible in the system. Simulation offers a straightforward means of acquiring reasonable expected-case time-based metrics. The use of GENSIM to structure set of simulations produces metrics which argue convincingly in favor of IOS over LTP for most reasonably small and reasonably fast networks. Conventional analysis does not permit derivation of this intuitive result.

ACKNOWLEDGEMENTS

This research was supported in part by the National Science Foundation through grant CCR-8705603.

REFERENCES

Cooper, E.C. (1985), "Replicated Distributed Programs," In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, 63–78.

Fidge, C.J. (1988a), "Partial Orders for Parallel Debugging," In *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, 130–140.

Fidge, C.J. (1988b), "Timestamps in Message-Passing Systems Which Preserve the Partial Ordering," In *Proceedings of the Eleventh Australian Computer Science Conference*, 56–66.

Kearns, P. and B. Koodalattupuram (1989), "Immediate Ordered Service in Distributed Systems," In *Proceedings of the Ninth International Conference on Distributed Computing Systems*, 611–618.

Lamport, L. (1978), "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM 21*, 7, 558–565.

Mattern, F. (1988), "Virtual Time and Global States of Distributed Systems," Technical Report, University of Kaiserslautern, Kaiserslautern, FDR.

Payne, J.E. (1989), "Ordering Events in a Distributed System: A Performance Evaluation," M.S. Thesis, Department of Computer Science, The College of William and Mary, Williamsburg, VA.

Payne, J.E. and P. Kearns (1990), "GENSIM: A Simulator for Distributed Protocols," In *Proceedings of the 1990 Summer Simulation Conference*, to appear.

Ricart, G. and A.K. Agrawala (1981), "An Optimal Algorithm for Mutual Exclusion in Computer Networks," *Communications of the ACM 24*, 1, 9–17.

Schneider, F.B. (1982), "Synchronization in Distributed Programs," *ACM Transactions on Programming Languages and Systems 4*, 2, 125–148.