Efficiency of Synchronous Versus Asynchronous Distributed Systems

ESHRAT ARJOMANDI

York University, Downsview, Ontario, Canada

MICHAEL J. FISCHER

University of Washington, Seattle, Washington

AND

NANCY A. LYNCH

Massachusetts Institute of Technology, Cambridge, Massachusetts

Abstract. A system of parallel processes is said to be *synchronous* if all processes run using the same clock, and it is said to be *asynchronous* if each process has its own independent clock. For any s, n, a particular distributed problem is defined involving system behavior at n "ports." This problem can be solved in time s by a synchronous system but requires time at least $(s-1)\lfloor \log_b n \rfloor$ on any asynchronous system, where b is a constant reflecting the communication bound in the model. This appears to be the first example of a problem for which an asynchronous system is provably slower than a synchronous one, and it shows that a straightforward step-by-step and process-by-process simulation of an n-process synchronous system by an n-process asynchronous system necessarily loses a factor of $\log_b n$ in speed

Categories and Subject Descriptors: F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*parallelism, relations among modes*; F.2.m [**Analysis of Algorithms and Problem Complexity**]: Miscellaneous

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Synchronous system, asynchronous system, clock, distributed system

1. *Introduction*

A system of parallel processes is said to be *synchronous* if all processes run using the same clock, so the processes operate in lockstep, and it is said to be *asynchronous* if each process has its own independent clock. Examples of synchronous systems are certain large centralized multiprocessing computers and VLSI chips containing many

This paper is based on research supported by the Office of Naval Research under Contracts N00014-80-C-0221 and N00014-79-C-0873, by the U.S. Army Research Office under Contract DAAG29-79-C-0155, by the National Science Foundation under Grants MCS 77-15628 and MCS 79-24370, and by the National Sciences and Engineering Research Council of Canada

Authors' present addresses: E. Arjomandi, Department of Computer Science, York University, Downsview, Ontario, Canada M3J 1P3; M. J. Fischer, Department of Computer Science, Yale University, P.O. Box 2158, New Haven, CT 06520; N. A. Lynch, Laboratory for Computer Science, M.I.T., Cambridge, MA 02139

separate parallel processing elements. Examples of asynchronous systems are distributed computer networks and I/O systems for conventional computers.

In this paper we compare time efficiency of a simple model of a synchronous system with a similar asynchronous model. For $s, n \in \mathbb{N}$, we define a particular distributed problem involving n "ports." This problem can be solved in time s on a synchronous system, but we show that it requires time at least $(s-1)\lfloor \log_b n \rfloor$ on any asynchronous system. Here b is a constant reflecting the communication bound in the model, whose precise definition is given in the next section. Finally, we show that if the communication system is slightly strengthened by permitting a single designated process to broadcast to all others, or if we provide each process with access to a global clock, then the asynchronous model can solve the problem in time $O(s)$.

Note that our argument must do much more than just compare two particular systems. Rather, we must reason about *all possible* asynchronous systems (within a particular formal model) and prove that none could possibly solve the given problem efficiently. In general, obtaining lower bounds and impossibility results is a more difficult task than obtaining corresponding upper bounds.

2. *The Model*

The model used in this paper is similar to the model defined in [1]. Briefly, it consists of collections P of *processes* and X of *shared variables*. The *global state* of the system consists of the internal state of each process together with the value of each shared variable. A *step* is an atomic action which consists of simultaneous changes to the state of some process and the value of some shared variable. Formally, a *step* σ is a pair of triples $((s, p, t), (u, x, v))$, where s, t are possible internal states of process p, and u, v are possible values of variable x; we define $process(\sigma) = p$ and $variable(\sigma) = x$ and say that σ *involves* p and *accesses* x.

Step σ is *applicable* to any global state in which process p has internal state s and variable x contains value u. The effect of performing σ is to change the state of p to t and simultaneously to change the value of x to v.

A system is specified by describing P, X, an initial global state, and a set OKSTEPS of possible steps. A process p *blocks* in a global state g if there is no step σ in OKSTEPS applicable to g with $process(\sigma) = p$. In this paper we require our systems to be nonblocking for all processes and all global states. By requiring our systems to be nonblocking, we are saying that any process is able to take a step on its own at any time. We are not saying that all of these steps must perform useful work, however—some might be "busy-waiting" steps.

Let $x \in X$, and define $locality(x) = \{process(\sigma) : \sigma \in \text{OKSTEPS and } variable(\sigma) = x\}$. A system is *b-bounded* if $|locality(x)| \leq b$ for every $x \in X$.

A *computation* of a system is a finite or infinite sequence of steps in OKSTEPS such that the first step is applicable to the initial global state and each succeeding step is applicable to the state resulting from the application of the previous step. The *result* of a finite computation is the global state after applying the sequence. An infinite computation is *admissible* if every process appears in infinitely many steps of the sequence.

A *round* is any sequence of steps such that every process appears at least once in the sequence. A *minimal round* is a round such that no proper prefix is a round. Every sequence of steps can be uniquely partitioned into segments such that every segment is a minimal round, except possibly for the last segment. We call this a partition into minimal rounds, even though the last segment is not necessarily a round.

A sequence of steps is *synchronous* if in the unique partition into minimal rounds:

(1) No two steps in the same round involve the same process;
(2) No two steps in the same round access the same variable.

Conditions (1) and (2) together imply that the steps in each round are independent and can be performed in any order, or simultaneously, with the same result.

The *run time* for a finite sequence of steps is defined to be the number of segments in the partition into minimal rounds. (This definition is equivalent to the one in [1], which says that the run time is the longest amount of elapsed real time that the system could take to execute the sequence, subject to the constraint that the time delay between two steps of the same process is at most unity.) Our requirement that the system be nonblocking, which enables processes to continue taking steps independently, makes our definition for "time" reasonable. If the system were permitted to block, then our definition would count the time until any process took a step as bounded, even if that process had to wait for an arbitrary amount of activity to occur before it could proceed. This seems quite unreasonable.

We imagine an outside agent who restricts the set of computations to be "allowable," as follows. An *asynchronous system* is a concurrent system whose allowable computations are all of its infinite admissible computations. A *synchronous system* is a concurrent system whose allowable computations are all of its infinite synchronous computations.

For synchronous systems, note that our definition for "time" is equivalent to the more usual one which simply counts the number of synchronous steps of the system, where one synchronous step consists of the simultaneous execution of a step by each process.

Although our results use a shared-variable model, they are intended to apply to models which use other communication primitives (such as messages) also. Other distributed computing models can generally be formalized within our model; for instance, a message system which accesses buffers can be modeled as a particular kind of process, which accesses particular kinds of shared variables. The measure of time for such systems which is derived from our basic time measure seems to be a reasonable one to consider.

3. *The Problem*

We now define a particular problem for a concurrent system. Let $Y \subseteq X$ be a distinguished set of variables called *ports*. A *port event* is any step that accesses a port. A *session* is any sequence of steps containing at least one port event for every port. A computation *performs* s sessions if it can be partitioned into s segments, each of which is a session. An infinite computation is *ultimately quiescent* if it contains only a finite number of port events. The *time to quiescence* of an ultimately quiescent sequence is the run time of the shortest prefix containing all port events.

Let $s, n \in \mathbb{N}$. The (s, n)-*session problem* is the problem of finding a concurrent system with n ports such that every allowable computation performs (at least) s sessions and is ultimately quiescent.

Note that the (s, n)-session problem, like the mutual exclusion and dining philosophers problems, concerns possible orderings of sequences of events rather than the computation of particular outputs. It is an abstraction of the synchronization needed in many natural problems. Consider, for example, a simple message distribution system in which a sending process writes a sequence of s messages one at a time on a board visible to all and waits after each message until all n other processes have

read the message. Let us regard each reading step by a process p as a "port event at port p." Any protocol which ensures that the sender has waited sufficiently long will also solve the (s, n)-session problem.

4. *Main Result*

We show that any asynchronous b-bounded system solving the (s, n)-session problem requires time at least $(s - 1)\lfloor \log_b n \rfloor$ to quiescence, whereas there is a trivial synchronous system which solves the problem in time exactly s. This is the first example we know of a problem for which an asynchronous system is provably slower than a synchronous one, and it shows that a straightforward step-by-step and process-by-process simulation of an n-process synchronous system by an n-process asynchronous system *necessarily* loses a factor of $\log_b n$ in speed.

The result is even more surprising when one realizes that the trivial asynchronous system with one process per port (and no communication among the processes) in which each process does nothing except access a port on each step in fact performs s sessions within time s. The difficulty is that no process knows when time s has elapsed (because of the lack of a global system "clock"), nor does it know when the s sessions have in fact been achieved, so none of the processes knows when to stop accessing its port.

A procedure which works is for a process associated with each port to perform a port event, broadcast that fact, and then wait until it has heard that all other port processes have performed their port events and that the session has been completed. This is repeated s times. By making the port processes the leaves of a tree network, the necessary communication for one session can be accomplished in time $O(\log n)$; hence the total time to quiescence for the solution is $O(s \log n)$. It seems very inefficient to wait after each port event, and one might try to invent clever schemes to increase the concurrency in the system. Our lower bound shows, however, that this method is optimal to within a constant factor, so only a limited amount of improvement is possible.

We now present the formal results.

THEOREM 1. *For all $s, n \in \mathbb{N}$ there is a 1-bounded synchronous system which solves the (s, n)-session problem such that the time to quiescence for each allowable computation is s.*

PROOF. The system has n processes, one corresponding to each port. Each process accesses its port on each of its first s steps and then ceases performing port events. In every infinite synchronous computation, each of the first s minimal rounds constitutes a session, and the system becomes quiescent after s rounds. Hence the system solves the (s, n)-session problem in time s. □

THEOREM 2 (MAIN RESULT). *Assume $b, s, n \in \mathbb{N}$, $b \geq 2$. For every b-bounded asynchronous system which solves the (s, n)-session problem, the time to quiescence is at least $(s - 1)\lfloor \log_b n \rfloor$ for some allowable computation.*

PROOF. Assume an asynchronous system which solves the (s, n)-session problem. Enumerate the processes arbitrarily. Construct an infinite admissible computation α by running the processes in round-robin order (one step of process 1, one step of process 2, ..., one step of process N, one step of process 1, ...). Each round-robin round is minimal; hence the time to perform the first r rounds is exactly r. Because we asssume a correct solution, this computation is ultimately quiescent. Let t be the time to quiescence. Then round t is the last round at which any port event occurs. We show $t \geq (s - 1)\lfloor \log_b n \rfloor$.

Let $\alpha = \beta\gamma$, where β contains the first t rounds of α and γ is the remaining tail. Our strategy is to construct a new infinite admissible computation $\alpha' = \beta'\gamma$, where β' is a reordering of β that results in the same global state as β, but β' performs at most $t/\lfloor \log_b n \rfloor + 1$ sessions. Since no port events occur in γ, it follows that α' performs at most $t/\lfloor \log_b n \rfloor + 1$ sessions. Since α' is an infinite admissible computation for the system, $t/\lfloor \log_b n \rfloor + 1 \geq s$, and the result follows.

To construct β', we first construct a partial order of the steps in β, representing "dependency." (Formally, the domain of the partial order consists of ordered pairs (i, ξ_i), where ξ_i is the ith step of β.) For every pair of steps σ, τ in β, we let $\sigma \leq_\beta \tau$ if $\sigma = \tau$ or if σ precedes τ in β and either process(σ) = process(τ) or variable(σ) = variable(τ). Close $\leq_\beta$ under transitivity. $\leq_\beta$ is a partial order, and every total order of steps of β consistent with $\leq_\beta$ is a computation which leaves the system in the same global state as β. (Clearly β itself defines such a total ordering.)

Now let $m = \lceil t/\lfloor \log_b n \rfloor \rceil$, and write $\beta = \beta_1 \cdots \beta_m$, where each β_k $(1 \leq k < m)$ consists of $\lfloor \log_b n \rfloor$ minimal rounds. Let y_0 be an arbitrary port. For $k = 1, \ldots, m$, we claim that there exists a port y_k and two sequences of steps ϕ_k and ψ_k, such that the following properties hold.

(i) $\phi_k\psi_k$ is a total ordering of the steps in β_k, consistent with $\leq_\beta$.
(ii) ϕ_k does not contain any step which accesses y_{k-1}.
(iii) ψ_k does not contain any step which accesses y_k.

Then if $\beta' = \phi_1\psi_1\phi_2\psi_2 \cdots \phi_m\psi_m$, it follows that β' is consistent with $\leq_\beta$. However, β' contains at most $m \leq t/\lfloor \log_b n \rfloor + 1$ sessions, since each session must contain steps on both sides of some ϕ_k–ψ_k boundary. (If a sequence of steps were completely contained in $\psi_{k-1}\phi_k$, for example, then it would fail to contain a step accessing port y_{k-1}.)

It remains to show the existence of the required y_k, ϕ_k, and ψ_k, $1 \leq k \leq m$. We proceed by induction on k. Assume that y_{k-1} has been defined, and define y_k, ϕ_k, and ψ_k as follows. There are two cases. First, if there is some port which is not accessed by any step of β_k, then let y_k be that port, ϕ_k the null sequence, and $\psi_k = \beta_k$. Properties (i)–(iii) are easily seen to be true in this case. Otherwise, let τ_k be the first step in β_k which accesses y_{k-1}. We claim that there exists a port y_k such that

(iv) if σ is a step in β_k which accesses y_k, then it is false that $\tau_k \leq_\beta \sigma$.

Let σ_k be the last step in β_k which accesses y_k. Assuming (iv) and then adding the pair (σ_k, τ_k) to $\leq_\beta$ and closing under transitivity results in another partial order $\leq_k$. Choose any total ordering β'_k of the steps in β_k consistent with $\leq_k$. τ_k is the first step in β'_k which accesses y_{k-1}, since all steps accessing the same variable are totally ordered in $\leq_k$. Let ϕ_k be the prefix of β'_k up to but not including τ_k, and let ψ_k be the remainder. σ_k occurs in ϕ_k since $\sigma_k \leq_k \tau_k$. (This is all illustrated in Figure 1.) Then properties (i)–(iii) are easily seen to be true.

It remains to verify the claimed existence of y_k. We do this by proving a series of three lemmas about the restriction of $\leq_\beta$ to β_k. These lemmas and their proofs rely on only a few properties of the resulting partial order. We state the required properties explicitly and thereby make the lemmas and their proofs entirely self-contained.

Let R be a totally ordered set $\{1, \ldots, |R|\}$ (of "round" numbers), P a finite set (of "processes"), and X a set (of "variables"). Let D (the "steps") be a finite set having mappings *round*: $D \to R$, *proc*: $D \to P$, and *var*: $D \to X$. Assume that for every pair $(r, p) \in R \times P$ there is exactly one $\sigma \in D$ having round(σ) = r and proc(σ) = p. Let $\text{loc}(x) = \{\text{proc}(\sigma) : \sigma \in D \text{ and } \text{var}(\sigma) = x\}$. Let $b \geq 2$ and assume $|\text{loc}(x)| \leq b$ for all $x \in X$.

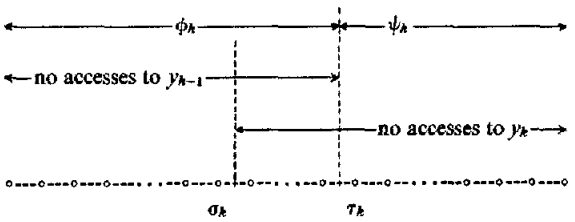

FIG. 1 A total ordering of steps in β_k consistent with $\leq_k$.

Let $\leq$ be a partial order on D, and write $\sigma <_1 \tau$ to indicate that $\sigma < \tau$ and there is no ρ with $\sigma < \rho < \tau$. Assume that $\leq$ has the following properties:

(i) If $\sigma <_1 \tau$, then either $\text{var}(\sigma) = \text{var}(\tau)$ or $\text{proc}(\sigma) = \text{proc}(\tau)$.
(ii) If either $\text{var}(\sigma) = \text{var}(\tau)$ or $\text{proc}(\sigma) = \text{proc}(\tau)$, then σ and τ are $\leq$-comparable.
(iii) If $\sigma \leq \tau$, then $\text{round}(\sigma) \leq \text{round}(\tau)$.

Finally, let $dep(\sigma) = \{\text{var}(\tau) : \sigma \leq \tau\}$.

LEMMA 1 (ANTIMONOTONICITY). *If $\sigma_1 \leq \sigma_2$, then $dep(\sigma_2) \subseteq dep(\sigma_1)$.*

PROOF. Obvious from the definition of dep. □

LEMMA 2. *Let $\sigma \in D$, $r = round(\sigma)$, and $x = var(\sigma)$. Let $C = \{\tau \in D : round(\tau) = r + 1$ and $proc(\tau) \in loc(x)\}$. Then $dep(\sigma) \subseteq \bigcup_{\tau \in C} dep(\tau) \cup \{x\}$.*

PROOF. Proof is by induction on $\leq$, beginning with maximal elements. Let $\sigma \in D$, and assume the lemma holds for all $\tau > \sigma$. Assume r, x, and C are defined from σ as in the statement of the lemma. If there exists $\sigma' \in D$ with $\text{var}(\sigma') = x$ and $\sigma' > \sigma$, then fix σ' as the smallest such member of D. (Property (ii) ensures that σ', if it exists, is defined uniquely.) Similarly, if there exists $\sigma'' \in D$ with $\text{proc}(\sigma'') = \text{proc}(\sigma)$ and $\sigma'' > \sigma$, then fix σ'' as the smallest such member of D. Define $B' = \text{dep}(\sigma')$ if σ' exists, ϕ otherwise, and $B'' = \text{dep}(\sigma'')$ if σ'' exists, ϕ otherwise. Then properties (i) and (ii) and antimonotonicity show that $\text{dep}(\sigma) \subseteq B' \cup B'' \cup \{x\}$. It suffices to show that $B' \cup B'' \subseteq \bigcup_{\tau \in C} \text{dep}(\tau) \cup \{x\}$.

We first consider B' and assume σ' exists. (If σ' does not exist, then there is nothing to prove.) By induction, $B' \subseteq \bigcup_{\tau' \in C'} \text{dep}(\tau') \cup \{x\}$, where $C' = \{\tau' \in D : \text{round}(\tau') = \text{round}(\sigma') + 1$ and $\text{proc}(\tau') \in \text{loc}(x)\}$. For every $\tau' \in C'$ there exists $\tau \in C$ with $\text{proc}(\tau) = \text{proc}(\tau')$, since every process takes a step in every round. Property (ii) shows that τ and τ' are $\leq$-comparable. But

$$\begin{aligned} \text{round}(\tau') &= \text{round}(\sigma') + 1 \\ &\geq \text{round}(\sigma) + 1 \qquad \text{by (iii)} \\ &= \text{round}(\tau). \end{aligned}$$

If $\tau \neq \tau'$, then $\text{round}(\tau') > \text{round}(\tau)$, since each process takes exactly one step in each round; in this case, (iii) implies that $\tau' > \tau$. Thus in any case it is true that $\tau \leq \tau'$. Antimonotonicity implies that $\text{dep}(\tau') \subseteq \text{dep}(\tau)$. Thus $B' \subseteq \bigcup_{\tau \in C} \text{dep}(\tau) \cup \{x\}$, as needed.

Finally, we consider B'' and assume σ'' exists. Then $\text{round}(\sigma'') = r + 1$, so that $\sigma'' \in C$. Thus $B'' \subseteq \bigcup_{\tau \in C} \text{dep}(\tau)$, as needed. □

LEMMA 3. *For each $\sigma \in D$ with $round(\sigma) = r$ it is the case that*

$$|dep(\sigma)| \leq \frac{b^{|R|-r+1} - 1}{b - 1}.$$

PROOF. We proceed by induction on r, starting with $r = |R|$ and working backward.

Basis: $r = |R|$. By Lemma 2, $\text{dep}(\sigma) \subseteq \{\text{var}(\sigma)\}$, so $|\text{dep}(\sigma)| \leq 1$, as needed.

Induction: $1 \leq r < |R|$. By Lemma 2 we have $|\text{dep}(\sigma)| \leq \sum_{\tau \in C} |\text{dep}(\tau)| + 1$, where C is defined as in Lemma 2. Each $\tau \in C$ has $\text{round}(\tau) = r + 1$, so by induction, $|\text{dep}(\tau)| \leq (b^{|R|-r} - 1)/(b - 1)$. Also, $|C| \leq b$. Hence,

$$|\text{dep}(\sigma)| \leq b \cdot \left\lceil \frac{b^{|R|-r} - 1}{b - 1} \right\rceil + 1 = \frac{b^{|R|-r+1} - 1}{b - 1},$$

as needed. □

We now return to the main proof and use Lemma 3 to show the existence of the needed y_k. We apply Lemma 3 to the subordering of $\leq_\beta$ defined by restriction to β_k. The set R of "round numbers" required for Lemma 3 is $\{1, \ldots, \lfloor \log_b n \rfloor\}$. The required mapping "rounds" is obtained by renumbering the round-robin rounds of β_k, preserving their previous order. Mappings "proc" and "var" are obtained from the mappings "process" and "variable," respectively. It is straightforward to see that the necessary properties of D and $\leq$ are satisfied. Then by Lemma 3, we see that $|\text{dep}(\tau_k)| \leq (b^{\lfloor \log_b n \rfloor - 1 + 1} - 1)/(b - 1) \leq n - 1$. Since there are n ports, this means that there must exist a port y_k satisfying the required property (iv). □

5. *Results for More General Models*

If the model is generalized by removing the bound on the number of processes which can access a shared variable, then a single communication variable shared by n port processes can be used to construct an $O(s)$ solution.

In fact, if the original model is only generalized slightly by allowing *one* of the shared variables to be *read* by an arbitrary number of processes (but only to be *changed* by one process), then an $O(s + \log n)$ solution is possible. In more detail, we use a shared variable, the *message board*, which every process can read but only one fixed process, the *supervisor*, can change. Each port has a corresponding port process, and there are additional communication processes whose job it is to pass messages through a tree network from the port processes back to the supervisor. The message board contains an integer which we call a "clock" value. The supervisor alternately increments the clock and reads the messages being sent back. Each port process repeatedly performs a cycle of reading the clock, performing a port event, and sending a message back to the supervisor, through the network, which contains the clock value just read and the port identifier.

If c_1 and c_2 are two successive clock values sent by port process i, then a port event must occur at port i sometime *after* the clock assumes value c_1 and *before* the clock assumes value $c_2 + 1$. By naturally combining this information about all ports, the supervisor can construct a sequence $0 = b_0 < b_1 < b_2 < \cdots < b_s$ such that for each j, a session is guaranteed to occur between the times when the clock first assumes values b_j and b_{j+1}. (Specifically, let $c_{i1}, c_{i2}, \ldots$ denote the successive clock values sent by port process i, $1 \leq i \leq n$. Then define $b_j = \max\{c_{i(k+1)} + 1 : 1 \leq i \leq n \text{ and } k \text{ is the smallest index such that } c_{ik} \geq b_{j-1}\}$, for each j, $1 \leq j \leq s$.) After the supervisor constructs this entire sequence, it knows that at least s sessions have in fact occurred, at which time it puts a STOP message on its message board. When the port processes read the STOP message, they stop performing port events.

It is easy to see that this construction solves the (s, n)-session problem. We argue that it satisfies the required $O(s + \log n)$ time bound.

First, we consider message transmission time. Since we are not assuming any upper bound on size of variables, the tree network can guarantee (by concatenating messages) that any message can be sent as soon as a process is ready to send it, and also that any message sent by time t is received by the supervisor by time $t + O(\log n)$.

Next, we claim that for each j, $1 \leq j \leq s$, the time that elapses from when the clock first assumes values b_{j-1} until it first assumes b_j is bounded above by a constant. For, from the time when the clock first assumes value b_{j-1}, it is at most a fixed constant amount of time before all port processes have read the clock, performed port events, sent messages containing clock values $\geq b_{j-1}$, and read the clock once again. Thereafter, it is at most one time unit before the clock is incremented again, thereby assuming value b_j.

Thus, the total elapsed time until the clock assumes value b_s is $O(s)$. Thereafter, within time $O(\log n)$, the supervisor has received all the needed messages and can deduce that s sessions have occurred and display the STOP message. Three time units later, all port processes will have read the STOP message and will have stopped performing port events.

6. *Conclusion*

We have demonstrated a particular situation in which asynchronous systems are provably less efficient than synchronous systems for solving a natural distributed problem. We expect that there are many other such situations.

It is quite pleasing to try to design distributed algorithms so that their logical correctness is timing-independent (i.e., so they are "completely asynchronous"), whereas their performance might depend on timing considerations. This paper suggest that this goal will not always be achievable: for some tasks, the only practical distributed solutions might be timing-dependent.

REFERENCES

1. LYNCH, N.A., AND FISCHER, M.J. On describing the behavior and implementation of distributed systems In *Theoretical Computer Science 13,* North-Holland, Amsterdam, 1981, pp. 17–43.

RECEIVED SEPTEMBER 1981; REVISED SEPTEMBER 1982; ACCEPTED SEPTEMBER 1982