Sharing Memory Robustly in Message-Passing Systems

HAGIT ATTIYA

The Technion, Haifa, Israel

AMOTZ BAR-NOY

IBM T. J. Watson Research Center, Yorktown Center, Yorktown Heights, New York

AND

DANNY DOLEV

IBM Almaden Research Center, San Jose, California and Hebrew University, Jerusalem, Israel

Abstract. Emulators that translate algorithms from the shared-memory model to two different message-passing models are presented. Both are achieved by implementing a wait-free, atomic, single-writer multi-reader register in unreliable, asynchronous networks. The two message-passing models considered are a complete network with processor failures and an arbitrary network with dynamic link failures.

These results make it possible to view the shared-memory model as a higher-level language for designing algorithms in asynchronous distributed systems. Any wait-free algorithm based on atomic, single-writer multi-reader registers can be automatically emulated in message-passing systems, provided that at least a majority of the processors are not faulty and remain connected. The overhead introduced by these emulations is polynomial in the number of processors in the system.

Immediate new results are obtained by applying the emulators to known shared-memory algorithms. These include, among others, protocols to solve the following problems in the message-passing model in the presence of processor or link failures: multi-writer multi-reader registers, concurrent time-stamp systems, l-exclusion, atomic snapshots, randomized consensus, and implementation of data structures.

A preliminary version of this paper appeared in *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing* (Quebec City, Quebec, Canada, Aug. 22–24). ACM, New York, 1990, pp. 363–376.

The research of H. Attiya was supported by grant no. 92-0233 from the United States–Israel Binational Science Foundation (BSF), Jerusalem, Israel. Part of the work was done while the author was at the Laboratory for Computer Science, MIT, supported by National Science Foundation (NSF) grant no. CCR 86-11442, by Office of Naval Research (ONR) Contract no. N00014-85-K-0168, and by DARPA Contracts no. N00014-83-k-0125 and N00014-89-J-1988.

Part of the work of A. Bar-Noy was done while the author was at the Computer Science Department, Stanford University, was supported in part by a Weizmann fellowship, by contract ONR N00014-88-k-0166, and a grant of Stanford's Center for Integrated Systems.

Authors addresses: H. Attiya, Department of Computer Science, The Technion, Haifa 32000, Israel: A. Bar-Noy, IBM T. J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598; D. Dolev, IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120, and the Computer Science Department, Hebrew University, Jerusalem 91904, Israel.

Categories and Subject Descriptors: C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)—*multiple-instruction-stream, multiple-data-stream processors (MIMD)*; C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*distributed networks*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems; D.4.1 [**Operating Systems**]: Process Management—*concurrency, multiprocessing/multiprogramming, synchronization*; F.1.1 [**Computation by Abstract Devices**]: Models of Computation—*relations among models*

General Terms: Algorithms, Theory, Reliability.

Additional Key Words and Phrases: Atomic registers, emulation, fault-tolerance, message passing, processor and link failures, shared memory, and wait-freedom

1. *Introduction*

Two major interprocessor communication models in distributed systems have attracted much attention and study: the *shared-memory* model and the *message-passing* model. In the shared-memory model, n processors communicate by writing and reading to shared registers. In the message-passing model, n processors are located at the nodes of a network and communicate by sending messages over communication links.

In both models, we consider asynchronous unreliable systems in which failures may occur. In the shared-memory model, processors may fail by stopping (and a slow processor cannot be distinguished from a failed processor). In the message-passing model, failures may occur in either of two ways. In the *complete network* model, processors may fail by stopping (without being detected). In the *arbitrary network* model, links fail and recover dynamically, possibly disconnecting the network for some periods.

The design of fault-tolerant algorithms in either of these models is a delicate and error-prone task. However, this task is somewhat easier in shared-memory systems, where processors enjoy a more global view of the system. A shared register guarantees that once a processor reads a particular value, then, unless the value of this register is changed by a write, every future read of this register is always available, regardless of processor slow-down or failure. These properties permit us to ignore issues that must be addressed in message-passing systems. For example, there are discrepancies in the local views of different processors that are not necessarily determined by the relative order at which processors execute their operations.

An interesting example is provided by the problem of achieving *randomized consensus*. Several solutions for this problem exist in the message-passing model, for example, Ben-Or [1983], Chor et al. [1985], and Dwork et al. [1986], and in the shared-memory model, for example, Abrahamson [1988], Aspnes and Herlihy [1990a], Attiya et al. [1989], and Chor et al. [1987]. However, the algorithm of Aspnes and Herlihy [1990a] is the first to have polynomial expected running time and still overcome an "omnipotent" adversary—one that has access to the outcomes of local coin-flips. The difficulty of overcoming messages' asynchrony in the message-passing model made it hard to come up with algorithms that tolerate such omnipotent adversary with polynomial expected running time.[1]

[1]The asynchronous message-passing algorithm of P. Feldman (personal communications) is resilient to Byzantine faults, but requires private communication links and thus is not resilient to an omnipotent adversary.

This paper presents *emulators* of shared-memory systems in message-passing systems (networks), in the presence of processor or link failures. Any wait-free algorithm in the shared-memory model that is based on atomic, single-writer multi-reader registers can be emulated in both message-passing models. The overhead for the emulations is polynomial in the number of processors. The complexity measures considered for each read or write operation are the number of messages, the size of messages, the execution time, and the local memory size.

Thus, shared-memory systems may serve as a "laboratory" for designing resilient algorithms. Once a problem is solved in the shared-memory model, it is automatically solved in the message-passing model, and only optimization issues remain to be addressed.

Among the immediate new results obtained by applying the emulators to existing shared-memory algorithms are network protocols that solve the following problems in the presence of processor or link failures:

—Atomic, multi-writer multi-reader registers [Peterson and Burns 1987; Vitanyi and Awerbuch 1986].
—Concurrent time-stamp systems [Dolev and Shavit 1989; Israeli and Li 1992].
—Variants of l-exclusion [Afek et al. 1990; Burns and Peterson 1989; Dolev et al. 1988].
—Atomic snapshot scan [Afek et al. 1993; Anderson 1993].
—Randomized consensus [Aspnes and Herlihy 1990a; Attiya et al. 1984].[2]
—Implementation of data structures [Aspnes and Herlihy 1990b; Herlihy 1991].

First, we introduce the basic communication primitive that is used in our algorithms. We then present an *unbounded* emulator (in messages and local memory) for the complete network model that tolerates processor failures. This implementation exposes some of the basic ideas underlying our constructions. In addition, part of the correctness proof for this emulator carries over to the other models. We then describe the modifications needed in order to obtain the *bounded* emulator for the complete network in the presence of processor failures. Finally, we modify this emulator to work in an arbitrary network in the presence of link failures. We present two ways to do so. The first modification is based on replacing each physical link of the complete network with a "virtual viable link" using an *end-to-end* protocol [Afek and Gafni 1988, 1991; Awerbuch et al. 1989]. The second modification results in a more efficient emulation. It is based on implementing our communication primitive as a diffusing computation using the *resynchronization* technique of Afek and Gafni [1991]. In both cases, the emulator for dynamic networks is bounded in the number and the size of messages, but not in the local memory overhead.

We consider systems that are completely asynchronous since this enables us to isolate the study from any model-dependent synchronization assumptions. Although many "real" shared-memory systems are at least partially syn-

[2] This result also follows from the transformation of Bar-Noy and Dolev [1989].

chronous, asynchrony allows us to provide an abstract treatment of systems in which different processors have different priorities.

We believe that bounded solutions are important, although in reality, 20-bit counters will not wrap around and thus will suffice for all practical purposes. The reason is because bounded solutions are much more resilient—traditional protocols fail if an error occurs and cause counters to grow without limit. An algorithm designed to handle bounded counters will be able to recover from such a situation and resume normal operation.

Wait-free protocols in shared-memory systems enable a processor to complete any operation regardless of the speed of other processors. In message-passing systems, it can be shown, following the proof in Attiya et al. [1989], that for many problems requiring global coordination, there is no solution that can prevail over a "strong" adversary—an adversary that can stop a majority of the processors or disconnect large portions of the network. Such an adversary can cause two groups of fewer than a majority of the processors to operate separately by suspending all the messages from one group to the other. For many global coordination problems, this leads to contradicting and inconsistent operations by the two groups. As mentioned in Attiya et al. [1989], similar arguments show that processors cannot halt after deciding. Thus, in our emulators, a processor that is disconnected (permanently) from a majority of the processors is considered *faulty* and is blocked.[3] A wait-free algorithm will run correctly under our emulators if at least a majority of the processors are non-faulty, and are connected to each other. Our solutions do not depend on connection with a *specific* majority at any time. Moreover, it might be that at no time does there exist a full connection to any party. The only condition is that messages will eventually reach some majority that will acknowledge them.

Although the difficult construction is the solution in the complete network with bounded-size messages, the unbounded construction is not straightforward. In both cases, to avoid problems resulting from processors having old values, we attach time-stamps to the values written by the writer. In the unbounded construction, the time-stamps are the integer numbers. In the bounded construction, we use a nontrivial method to let the writer keep track of old time-stamps that are still in the system. This allows us to employ a *bounded sequential time-stamp system* [Israeli and Li 1993].

Some of the previous research on dynamic networks (e.g., Afek et al. [1987] and Finn [1979]) assumed a "grace period" during which the network stabilizes for a long enough time in order to guarantee correctness. Our results do not rely on the existence of such a period, and follow the approach taken in, for example, Afek and Gafni [1988, 1991], Awerbuch et al. [1989], and Vishkin [1983].

There are two related studies on the relationships between shared-memory and message-passing systems. Bar-Noy and Dolev [1989] provide translations between protocols in the shared-memory and the message-passing models. These translations apply only to protocols that use a restricted form of communication. Chor and Moscovici [1989] present a hierarchy of resilience for problems in shared-memory systems and complete networks. They show that for some problems, the wait-free shared-memory model is not equivalent

[3]Such a processor will not be able to terminate its operation but will never produce erroneous results.

to the complete network model, where up to half of the processors may fail. This results, however, assumes that processors *halt* after deciding.

The rest of this paper is organized as follows: In Section 2, we describe the various models considered. In Section 3, we introduce the communication primitive. In Section 4, we present an unbounded implementation for the complete network model in the presence of processor failures. In Section 5, we present the modifications needed in order to obtain the bounded implementation for the complete network model in the presence of processor failures. In Section 6, we modify this emulator to work in an arbitrary network in the presence of link failures. We conclude, in Section 7, with a discussion of the results and some directions for future research.

2. *Preliminaries*

In this section, we discuss the models addressed in this paper. Our definitions follow Lamport [1986] for shared-memory systems, Fischer et al. [1985] for complete networks with processor failures, and Awerbuch et al. [1989] for arbitrary networks with link failures. In all models we consider, a system consists of n independent and asynchronous processors, numbered $1, \ldots, n$. Each processor i is a (possibly infinite) state machine, with a unique initial state, and a transition function.

2.1. Message-Passing Systems.

In a message-passing system, processors communicate by sending *messages* (taken from some alphabet $\mathcal{M}$) to each other. Processors are located at the nodes of a network (which we do not model explicitly) and can send messages only to their direct neighbors.

We model computations of the system as sequences of steps. Each step is either a *message delivery step*, representing the delivery of a message to a processor, or a *computation step* of a single processor.

In each message delivery step, a single message is placed in the incoming buffer of the processor. Formally, a message delivery step is a pair (i, m), where $m \in \mathcal{M}$. In each indivisible computation step, a processor receives all messages delivered to it since its last computation step, performs some local computation and sends some messages, and possibly changes its local state. Formally, a computation step of processor i is a tuple (i, s, s', M), where s and s' are the old and new states of i (respectively), and $M \subseteq \mathcal{M} \times \{1, \ldots, n\}$ is the set of messages sent by processor i. The set M specifies a set of *send events* that occur in a computation step. A message m is *delivered* to processor i when the step (i, m) happens; the message is *received* by i when i takes a computation step following the delivery of that message.

An *execution* is an infinite sequence of steps satisfying the following conditions:

(1) the old state in the first computation step of processor i is i's initial state,
(2) the old state of each subsequent computation step of processor i s the new state of the previous step,
(3) the new state of any computation step of processor i is obtained by applying the transition function of i to the old state and the messages delivered since the last computation step, and
(4) there is a one-to-one mapping from delivery steps to corresponding send events (with the same message).

The network is not explicitly modeled; however, the last condition guarantees that messages are not duplicated or corrupted. The network is allowed to not deliver some messages or to deliver them out of order.

In the complete network model, we assume that the network formed by the communication links is *complete*, and that processors might be *faulty*. A faulty processor simply stops operating. More formally, processor i is *nonfaulty* in an execution if the execution contains an infinite number of computation steps by i and all messages it sends are eventually delivered; otherwise, it is *faulty*. Note that messages will be delivered to a faulty processor, even after it stops taking steps.

In dynamic networks, communication links might become *non-operational.* A link is nonoperational, if, starting from some message and on, it does not deliver any further messages to the other endpoint. More precisely, if a specific message is not delivered, then all messages sent after it, will not be delivered. For those messages, the delay is considered to be infinite. Otherwise, the link is *operational.*[4] Processor i is *connected* to a processor j if there is a path of operational links between them; otherwise, i is *disconnected* from j. A processor that is disconnected from $\lceil n/2 \rceil$ processors or more is *faulty*.

2.2. Atomic Registers. An axiomatic definition of an atomic register can be found in Lamport [1986]. The definition presented here is an equivalent one (see [Lamport 1986, Proposition 3]) that is simpler to use. An *atomic, single-writer multi-reader register* is an abstract data structure. Each register is accessed by two procedures, $\mathsf{write}_w(v)$ that is executed only by some specific processor w, called the *writer*, and $\mathsf{read}_r(v)$ that may be executed by any processor r, $1 \leq r \leq n$, called a *reader*. We can associate computation steps with calling and returning from these procedures in a natural way. An operation *precedes* another if it returns before the other operation is called. Two operations are concurrent if neither of them precedes the other.

The values returned by these procedures, when applied to the same register, must satisfy the following two properties:[5]

(1) Every read operation returns either the value written by the most recent preceding write operation (the initial value if there is no such write) or a value written by a write operation that is concurrent with this read operation.
(2) If a read operation $\mathscr{R}_1$ reads a value from a write operation $\mathscr{W}_1$, and a read operation $\mathscr{R}_2$ reads a value from a write operation $\mathscr{W}_2$ and $\mathscr{R}_1$ precedes $\mathscr{R}_2$, then $\mathscr{W}_2$ does not precede $\mathscr{W}_1$.

2.3. Complexity Measures. The complexity measures we consider are the following:

(1) The number of messages sent in an execution of a write or read operation,
(2) the size of the messages,

[4] This model is called the *∞-delay-model* in Afek et al. [1987] and Afek and Gafni [1988]. Afek and Gafni [1988] point out that the standard model of dynamic mesage-passing systems, where communication links alternate between periods of being operational and non-operational, can be reduced to this model.

[5] We concentrate on the implementation of a single register; multiple copies of the emulator can be used to implement any finite number of registers.

(3) the time it takes to execute a write or read operation, under the assumption that any message is either received within one time unit, or never at all (cf. Awerbuch [1987]),
(4) the amount of the local memory used by a processor.

For all these measures, we are interested in the worst-case complexity.

3. *Procedure* communicate

In this section, we present the basic primitive used for communication in our algorithms, called communicate. This primitive operates in complete networks. It enables a processor to send a message and get acknowledgments (possibly carrying some information) from a majority of the processors.

Because of possible processors' failures, a processor cannot wait for acknowledgments from all the other processors or from any particular processor. However, at least a majority of the processors will not fail and thus a processor can wait to get acknowledgments from them. Notice that processors want to communicate with any majority of the processors, not necessarily the same majority each time. A processor uses the primitive to broadcast a message $\langle M \rangle$ to all the processors and then to collect a corresponding $\langle ACK \rangle$ message from a majority of them. In some cases, information will be added to the $\langle ACK \rangle$ messages.

For simplicity, we assume that each edge (i, j) is composed of two distinct "virtual" directed edges $\langle i, j \rangle$ and $\langle j, i \rangle$. The communication on $\langle i, j \rangle$ is independent of the communication on $\langle j, i \rangle$.

Procedure communicate uses a simple *ping-pong* mechanism. This mechanism ensures FIFO communication on each directed link in the network, and guarantees that at any time only one message is in transit on each link. Informally, this is achieved by the following rule: i sends the first message on $\langle i, j \rangle$ and then i and j alternate turns in sending further messages and acknowledgments on $\langle i, j \rangle$.

More precisely, the ping-pong on the directed edge $\langle i, j \rangle$ is managed by processor i. Processor i maintains a vector *turn* of length n, with an entry for each processor, that can get the values here or there. If $turn(j) = here$, then it is i's turn on $\langle i, j \rangle$ and only then i may send a message to j. If $turn(j) = there$, then either i's message is in transit, j's acknowledgment is in transit, or j received i's message and has not replied yet (it might be that j failed). Initially, $turn(j) = here$. Hereafter, we assume that the vector *turn* is updated automatically by the **send** and **receive** operations.[6] For simplicity, a processor also sends each message to itself and responds with the appropriate acknowledgment.

Procedure communicate gets as an input a message M and returns as an output a vector *info*, of length n. The jth entry in this vector contains information received with j's acknowledgment (or $\perp$ if no acknowledgment was received from j). To control the sending of messages, the procedure maintains a local vector *status*. The jth entry of this vector may obtain one of the following values: *notsent*, meaning M was not sent to j (since $turn(j) = there$); *notack*, meaning M was sent but not yet acknowledged by j; *ack*, meaning M was acknowledged by j. Additional local variables in Procedure

[6] The details of how this is done are omitted from the code.

```
Procedure communicate(⟨M⟩; info); (* for processor i *)
        #acks := 0;
        for all 1 ≤ j ≤ n do
                status(j) := notsent ,
                info(j) :=⊥ ;
        for all 1 ≤ j ≤ n s.t. turn(j) = here do
                send ⟨M⟩ to j ;
                status(j) := notack ,
        repeat until #acks ≥ ⌈(n+1)/2⌉
                upon receiving ⟨m⟩ from j:
                        if status(j) = notsent then
                        (* acknowledgment of an old message *)
                          send ⟨M⟩ to j ;
                          status(j) := notack;
                        else if status(j) = notack then
                          status(j) := ack ;
                          info(j) := m ;
                          #acks := #acks + 1 ,
end procedure communicate,
```

FIG. 1. *Procedure* communicate.

communicate are the vector *turn* and the integer counter, *#acks* that counts the number of acknowledgments received so far.

The pseudo-code for this procedure appears in Figure 1. We note that whenever this procedure is employed we also specify the information sent with the acknowledgment for each message and the local computation triggered by receiving a particular message.

The ping-pong mechanism guarantees the following two properties of the communicate procedure. First, the acknowledgments stored in the output vector *info* were indeed sent as acknowledgments to the message M, that is, at least $\lceil (n+1)/2 \rceil$ processors received the message M. Second, the number of messages sent during each execution of the procedure is at most $2n$. Also, it is not hard to see that the procedure terminates under our assumptions. The next lemma summarizes the properties and the complexity of Procedure communicate.

LEMMA 3.1. *The following all hold for each execution of Procedure* communicate *by processor i with message* $\langle M \rangle$:

(1) *if i is connected to at least a majority of the processors, then the execution terminates after at most two time units,*
(2) *at least* $\lceil (n+1)/2 \rceil$ *processors receive* $\langle M \rangle$ *and return the corresponding acknowledgment,*
(3) *at most* $2n$ *messages are sent during this execution, and*
(4) *the size of i's local memory is* $O(n)$ *times the size of the acknowledgments to* $\langle M \rangle$.

4. *The Unbounded Implementation—Complete Network*

Informally, in order to write a new value, the writer executes communicate to send its new value to a majority of the processors. It completes the write

operation only after receiving acknowledgments from a majority of the processors. The writer appends a label to every new value it writes. In the unbounded implementation, this is an integer. For simplicity, we ignore the value itself and identify it with its label.

In order to read a label, the reader sends a request to all processors and gets in return the latest labels known to a majority of the processors (using communicate). Then it adopts (returns) the maximal among them. Before finishing the read operation, the reader announces the label it intends to adopt to at least a majority of the processors (again by using communicate). Informally, this announcement is needed since, otherwise, it is possible for a read operation to return the label of a write operation that is concurrent with it, and for a later read operation to return an earlier label.

Processor i stores in its local memory a variable $label_i$, holding the most recent label of the register known to i. This label may be acquired either during i's read operations, from messages sent during other processors' read operations, or directly from the writer. In addition, i holds a vector of length n named *info*, containing the most recent labels sent to i by other processors as acknowledgments to i's request message. Letting $\mathscr{V}$ denote the number of bits needed to represent any label from the domain of all possible labels, we have:

PROPOSITION 4.1. *The size of the local memory at each processor is* $O(n\mathscr{V})$.

In the implementation, there are two procedures: read for the readers, write for the writer. In addition, we also specify acknowledgments to all types of messages, as required by communicate. The algorithm uses six types of messages, arranged in three pairs, each consisting of a message and a corresponding acknowledgment.

(1) The pair of write messages.

$\langle W, label \rangle$: sent by the writer in order to write *label* in its register.
$\langle ACK\text{-}W \rangle$: the corresponding acknowledgment.

(2) The first pair of read messages.

$\langle R_1 \rangle$: sent by the reader to request the recent label of the writer.
$\langle label \rangle$: the corresponding acknowledgment, contains the sender's most updated label of the register.

(3) The second pair of read messages.

$\langle R_2, label \rangle$: sent by the reader before terminating in order to announce that it is going to return *label* as the label of the register.
$\langle ACK\text{-}R_2 \rangle$: the corresponding acknowledgment.

The second pair of read messages is used by the reader to announce to other readers which label it is going to adopt. Clearly, we have:

PROPOSITION 4.2. *The maximum size of a message is* $O(\mathscr{V})$.

A pseudo-code for the algorithm appears in Figure 2. The bottom part instructs each processor how to acknowledge each message according to the template in Figure 1 (as explained in Section 3). We use *void* to say that the information sent with the acknowledgments to a particular message is ignored.

```
Procedure read_i(label_i); (* executed by processor i and returns label_i *)
        communicate(⟨R_1⟩, info);
        label_i := max_{1≤j≤n}{info(j) | info(j) ≠ ⊥};
        communicate(⟨R_2, label_i⟩, void),
end procedure read_i ,

Procedure write_w; (* executed by the writer w *)
        label_w = label_w + 1, (* the new label of the register *)
        communicate(⟨W, label_w⟩, void);
end procedure write_w;

(* acknowledgments sent by processor j *)
case received from w
    ⟨W, label_w⟩:  label_j := max{label_w, label_j} ;
                   send ⟨ACK-W⟩ to w;
case received from i
    ⟨R_1⟩:         send ⟨label_j⟩ to i;
    ⟨R_2, label_i⟩:  label_j := max{label_i, label_j} ;
                   send ⟨ACK-R_2⟩ to i;
```

FIG. 2. The unbounded emulator.

Since communication is done only by communicate, Lemma 3.1 (part 1) implies the following lemma.

LEMMA 4.3. *Each execution of a* read *operation or a* write *operation terminates.*

The label contained in the first write message and the second read message is called the label *communicated* by the communicate procedure execution. The maximum label among the labels contained in the acknowledgments of the first read message is called the label *acknowledged* by the communicate procedure execution. The following lemma deals with the ordering of these labels, and is the crux of the correctness proof.

LEMMA 4.4. *Assume a* communicate *procedure execution* $\mathscr{C}_1$ *communicated* x, *and a* communicate *procedure execution* $\mathscr{C}_2$ *acknowledged* y. *Assume that* $\mathscr{C}_1$ *has completed before* $\mathscr{C}_2$ *has started. Then* $y \geq x$.

PROOF. By Lemma 3.1 (part 2) and the code for acknowledgments, when $\mathscr{C}_1$ is completed at least a majority of the processors store a label that is greater or equal to x. Similarly, by Lemma 3.1 (part 2), in $\mathscr{C}_2$ acknowledgments were received from at least a majority of the processors. Thus, there must be at least one processor, say i, that stored a label $x'_i \geq x$ and acknowledged in $\mathscr{C}_2$. Since y is maximal among the labels contained in the acknowledgments of $\mathscr{C}_2$, it follows that $y \geq x'_i \geq x$. □

A write operation completes only after its communicate procedure completes. By Lemma 4.4, every read operation that will start after the write

operation completes will read a value with a greater than or equal timestamp. Since any earlier write operation has a smaller timestamp, it follows that:

LEMMA 4.5. *Assume a* read *operation, $\mathscr{R}$, returns the label y. Then y is either the label of the last* write *operation that was completed before $\mathscr{R}$ started or it is the label of a concurrent* write *operation.*

In a similar manner, since a read operation completes only after its second execution of communicate is completed, Lemma 4.4 implies the following lemma.

LEMMA 4.6. *Assume some* read *operation, $\mathscr{R}_1$, returns the label x, and that another* read *operation, $\mathscr{R}_2$, that started after $\mathscr{R}_1$ completed, returns y. Also, assume that some* write *operation, $\mathscr{W}_1$, wrote x and that another* write *operation, $\mathscr{W}_2$, wrote y. Then $\mathscr{W}_1$ precedes $\mathscr{W}_2$.*

Since processors communicate only by using procedure communicate, Lemma 3.1 (parts 3 and 4) implies the following complexity propositions.

PROPOSITION 4.7. *At most $4n$ messages are sent during each execution of a* read *operation. At most $2n$ messages are sent during each execution of a* write *operation.*

PROPOSITION 4.8. *Each execution of a* read *operation takes at most 4 time units. Each execution of a* write *operation takes at most 2 time units.*

The next theorem summarizes the above discussion.

THEOREM 4.9. *There exists an unbounded emulator of an atomic, single-writer multi-reader register in a complete network, in the presence of at most $\lfloor (n-1)/2 \rfloor$ processor failures. Each execution of a* read *operation or a* write *operation requires $O(n)$ messages and $O(1)$ time.*

5. *The Bounded Implementation—Complete Network*

5.1. INFORMAL DESCRIPTION. The only source of unboundedness in the emulation described in Section 4 is the integer labels used by the writer. In order to eliminate this, we use an idea that was employed previously in Awerbuch et al. [1989] and Israeli and Li [1993]. The integer labels are replaced by *bounded sequential time-stamp system* [Israeli and Li 1993], which is a finite domain $\mathscr{L}$ of label values together with a total order relation $\succ$. Whenever the writer needs a new label it produces a new one, larger (with respect to the $\succ$ order) than all the labels that exist in the system. Thus, instead of simply adding 1 to the label, as in the unbounded emulation, the writer invokes a special procedure called LABEL. The input for this procedure is a set of labels and the output is a new label that is greater than all the labels in this set. This can be achieved by the constructions presented in Dolev and Shavit [1987] and Israeli and Li [1993] for bounded sequential time-stamp systems. In addition, the readers need a special function, called MAX, which returns the maximum of two values according to the ordering $\succ$ on labels. Again, this can be achieved by the constructions of Dolev and Shavit [1987] and Israeli and Li [1993].

The main difficulty in implementing this idea in the message-passing model is in maintaining the set of labels existing in the system. Notice that in order to

assure correctness, it is sufficient to guarantee that the set of labels that exist in the system is contained in the input set of labels of Procedure LABEL.

To overcome this difficulty, the writer collects from other processors labels that are still in the system—labels that they have for the writer's label or the most recent labels that they have sent to other processors. The set of labels collected is then supplied to Procedure LABEL. The problem is that the writer can only expect answers from a majority of the processors. Therefore, each processor makes sure that this information is stored in the system, by sending it to a majority of the processors. Thus, any majority of the processors would be able to provide the writer with all the labels that are still in the system.

To this end, whenever a processor adopts a label as the maximum label of the writer, it *records* this label and all the recent labels it has sent to other processors. This is done by broadcasting a message including this information and waiting for acknowledgments from a majority of the processors (using communicate). A processor receiving a recording message stores it in its local memory. In response to a query from the writer about labels, a processor sends all the labels it has stored. This guarantees that labels do not get lost, as a majority of the processors have stored them.

To avoid a chain reaction, where a recording message causes other recording messages, processors ignore the labels carried by recording messages even if their value is greater than the value they have for the writer's label. In addition, a separate ping-pong mechanism is used for each type of message, and thus, for example, processor i may send a recording message to processor j although j did not acknowledge a read message of i.

5.2. Data Structures and Messages. To implement the recording process, each processor i maintains an $n \times n$ matrix L_i of labels. The ith row vector $L_i(i)$ is updated dynamically by i according to the messages i sends. The jth row vector $L_i(j)$ is updated by the messages i receives from j during a recording process initiated by j. Each entry, $L_i(i, k)$, is composed of two fields: *sent* and *ack*. The field $L_i(i, k).sent$ contains the last label i sent to k during the second phase of a read operation and the field $L_i(i, k).ack$ is the last label i sent to k as an acknowledgment to a read request of k. In particular, $L_i(i, i)$ is the current maximum label of the writer known to i. The writer starts each write operation by obtaining from a majority of the processors their most updated values for the matrix L (using communicate). The union of the labels that appear in its own matrix and these matrices is the input to Procedure LABEL.

Procedures read and write use five pairs of messages and corresponding acknowledgments.

(1) The first pair of write messages.

- $\langle W_1 \rangle$: sent by the writer at the beginning of its operation in order to collect information about existing labels.
- $\langle L \rangle$: the corresponding acknowledgment, L is the sender's updated value of the labels' matrix.

(2) The second pair of write messages, $\langle W_2, label \rangle$ and $\langle ACK\text{-}W_2 \rangle$, the first pair of read messages, $\langle R_1 \rangle$ and $\langle label \rangle$, and the second pair of read messages, $\langle R_2, label \rangle$ and $\langle ACK\text{-}R_2 \rangle$, are the same as the corresponding messages in the unbounded algorithm.

(3) The pair of recording messages.

$\langle REC, L(i)\rangle$: before adopting any new label for the register, processor i sends $L_t(i)$ to other processors. The vector $L_t(i)$ contains this new label and all the recent labels that i sent to other processors.

$\langle ACK\text{-}REC\rangle$: the corresponding acknowledgment.

Let $\mathscr{Y}$ denote the number of bits needed to represent any label from $\mathscr{L}$ (i.e., $\mathscr{Y} = \log|\mathscr{L}|$). Since the longest message is $\langle L\rangle$, we have,

PROPOSITION 5.2.1. *The maximum size of a message is* $O(n^2 \cdot \mathscr{Y})$.

Since the local memory of a reader contains one matrix and that of the writer contains n matrices we have,

PROPOSITION 5.2.2. *The size of the local memory of a reader is* $O(n^2 \cdot \mathscr{Y})$. *The size of the local memory of a writer is* $O(n^3 \cdot \mathscr{Y})$.

5.3. THE ALGORITHM. The pseudo-code for the algorithm appears in Figure 3. The code for Procedure read and Procedure write and the acknowledgment process are similar to the pseudo-code of the unbounded emulator (Figure 2). The first part of Procedure recording and the update commands dynamically update the vector $L_t(i)$.

Unlike a read or a write operation, a recording process is an event-driven process. Several recording processes, initiated by messages of different processors, may be in progress at the same time. The following are the two rules that govern concurrent recording processes. First, a recoding message is not communicated before the previous recording message was acknowledged by a majority of the processors (this is achieved by the **wait** command in the code of Procedure recording). Second, if multiple recordings are ready to begin at the same computation step (perhaps after waiting), then they are combined into a single recording for a vector $L_t(i)$ in which each component is taken to be the maximum among the corresponding components of all the vectors in the recording processes to be combined. These rules guarantee that at any time at most one recording process is in progress per processor.

5.4. CORRECTNESS AND COMPLEXITY. Atomicity of the bounded emulator follows from the same reasoning as in the unbounded case (Lemmas 4.5 and 4.6). The following lemma, Lemma 5.4.1 is the core of the correctness proof for the bounded emulator—it assures that the writer always obtains a superset of the labels that might be considered by some processor.

For this lemma, we need some definitions. A label x is *viable* in a system state after some finite execution prefix if (1) for some processor i, the current register's label in i is x, or (2) for some processors i and j, a nonrecording message containing x has been sent by i to j and is received in some execution extending this prefix. More precisely, either $L_i(i,i) = x$, $L_i(i,j).sent = x$, or $L_i(i,j).ack = x$. In these cases, we say that processor i is *responsible* for label x. Thus, by definition, a label is viable if and only if it has a processor that is responsible for it. Intuitively, a viable label is or could be a candidate for the current register's label for some processor. A label x is *recorded* in some state

```
Procedure read_i(label_i) ; (* executed by processor i and returns label_i *)
        communicate_R(⟨R_1⟩, info) ;
        label_i := MAX{L_i(i,i) , MAX_{1≤j≤n}{info(j) | info(j) ≠ ⊥}};
        if label_i ≻ L_i(i,i) then recording_i(label_i) ,
        communicate_R(⟨R_2, L_i(i,i)⟩, void) ,
end procedure read_i ;

Procedure write_w ; (* executed by the writer w *)
        communicate_W(⟨W_1⟩, L) ;
        L_w(w,w) := LABEL(⋃ L) ; (* all the non-empty entries in L *)
        communicate_W(⟨W_2, L_w(w,w)⟩, void) ;
end procedure write_w ;

Procedure recording_i(label_i) ; (* executed by processor i *)
        L_i(i,i) := label_i ;
        wait until the previous call for recording_i terminates ;
        communicate_REC(⟨REC, L_i(i)⟩, void) ,
end procedure recording_i ;

(* updates executed by processor i *)
upon sending label x to j in acknowledgement to j's R_1 message:
      L_i(i,j).ack := x ;
upon sending label x to j in i's R_2 message:
      L_i(i,j).sent := x ;


(* acknowledgments sent by processor j *)
case received from w
      ⟨W_1⟩:          send ⟨L_j⟩ to w;
      ⟨W_2, label_w⟩: if label_w ≻ L_j(j,j) then recording_j(label_w) ;
                      send ⟨ACK-W_2⟩ to w ;
case received from i
      ⟨R_1⟩:          send ⟨L_j(j,j)⟩ to i;
      ⟨R_2, label_i⟩: if label_i ≻ L_j(j,j) then recording_j(label_i) ;
                      send ⟨ACK-R_2⟩ to i;
      ⟨REC, L_i(i)⟩: L_j(i) := L_i(i) ;
                      send ⟨ACK-REC⟩ to i;
```

FIG. 3. The bounded emulator.

if it is stored either in the writer matrix or in the matrices of at least a majority of the processors.

LEMMA 5.4.1. *Every viable label is recorded.*

PROOF. Assume x is a viable label and let i be some processor that is responsible for x. Consider a simple path on which the label x has arrived at i, that is, a sequence $i_0, i_1, \ldots, i_m$, where i_0 is the writer and $i_m = i$. In this sequence, for any l, $1 \leq l \leq m$, processor i_l adopted x as a result of a message from i_{l-1}.

The claim is proved by induction on m, the length of this path. The base case, $m = 0$, occurs when i is the writer. Then, the code of the update process and Procedure write implies that x is stored in i's matrix. Assume that $m > 0$ and that the induction hypothesis holds for $m - 1 \geq 0$. That is, if processor j is

responsible for a viable label y and y has arrived at j via the sequence of processors $j_0, j_1, \ldots, j_{m-1}$ where j_0 is the writer, $j_{m-1} = j$, and for every 1, $l \leq l \leq m-1$ processor j_l adopted y as a result of a message from j_{l-1}, then label y is recorded.

We now prove the claim holds for m. Since processor i is responsible for x, it follows that either $L_i(i,i) = x$, $L_i(j,j).sent = x$, or $L_i(i,j).ack = x$. This implies that $L_i(i,i) = x$ at some state in the execution prefix leading to this state. Since $L_i(i,i)$ can be set to x only in a recording process, it follows that i has invoked a recording process for x. There are two cases:

(1) Processor i has not finished the recording process for x. Let $k = i_{m-1}$. We first show that $x \in L_k(k,i)$. If x arrives at i in an R_2 message from k, then, since i has not finished the recording process for x, i has not sent an $ACK\text{-}R_2$ to k. Clearly, k sets $L_k(k,i).sent = x$ when sending x to i. The claim follows since k will not send an R_2 message to i before receiving an $ACK\text{-}R_2$ message from i, and hence $L_k(k,i).sent = x$. Otherwise, x arrives at i in an $ACK\text{-}R_1$ from k. Since i has not finished the recording process for x, i will not send another R_1 message. Clearly, k sets $L_k(k,i).ack = x$ when sending x to i. The claim follows since k will not send an $ACK\text{-}R_1$ message to i before receiving an R_1 message from i, and hence $L_k(k,i).ack = x$.
In either case, $x \in L_k(k,i)$ and hence k is responsible for x. The claim follows from the induction hypothesis.

(2) Processor i has finished the recording process for x. Let $k_1, k_2, \ldots, k_{\lceil (n+1)/2 \rceil}$ be the processors that acknowledged the recording process either for x or for some $y \succ x$. If $L_i(i,i) = x$, then processors $k_1, k_2, \ldots, k_{\lceil (n+1)/2 \rceil}$ have x in their copy of $L(i,i)$. If $L_i(i,i) = y \succ x$, then either $L_i(i,j).sent = x$, or $L_i(i,j).ack = x$ (for some processor j). Since x was sent by i to j before the recording process for y started, it follwos that $x \in L_i(i,j)$ during the recording process for y. Thus, for any $k \in \{k_1, \ldots, k_{\lceil (n+1)/2 \rceil}\}$, either $L_k(i,i) = x$ or $x \in L_k(i,j)$. Consequently, a majority of the processors have x in their copy of $L(i)$, and therefore, x is recorded. □

LEMMA 5.4.2. *A label generated in the call to Procedure* LABEL *is greater than any viable label in the system.*

PROOF. The input of Procedure LABEL contains the labels that appear in the matrix of the writer or in the matrices of a majority of the processors, that is, all the recorded labels. By Lemma 5.4.1, it contains all the viable labels. The lemma follows from the existence of bounded sequential time-stamp systems [Israeli and Li 1993; Dolev and Shavit 1987]. □

The next lemma shows, in particular, that in the bounded implementation, processors do not reach a deadlock and terminate each operation they start.

LEMMA 5.4.3. *Each execution of a* read *or* write *operation takes at most* 12 *time units.*

PROOF. We describe a worst-case time scenario for a read operation. A similar, but shorter, scenario for the write operation is omitted.

Recall that the maximum delay of a message is 1. Suppose processor i sent an $\langle R_1 \rangle$ message at time t. By time $t+2$, all the acknowledgments for this message have arrived. Assume that as a result of the acknowledgments proces-

sor i must start a recording process for some label x. It could be the case that processor i cannot start this recording process immediately, since processor i has started another recording process before time $t + 2$. However, this other recording process will end no later than time $t + 4$. Thus, the recording process for x would start no later than time $t + 4$ and would terminate no later than time $t + 6$.

Hence, processor i sends an $\langle R_2, y \rangle$ message, for $y \geq x$, no later than time $t + 6$. By time $t + 7$, at least a majority of the processors receive this message and may need to start a recording process for y. Again, it is possible that they cannot start the recording process for y immediately since another recording process is in progress. However, like before, the recording of y would start no later than time $t + 9$ and would terminate before time $t + 11$. Thus, these processors can send the $\langle ACK - R_2 \rangle$ message before time $t + 11$, and processor i gets these messages no later than time $t + 12$ and finishes its read operation. □

The worst-case scenario for a read operation requires $O(n^2)$ messages, when $O(n)$ processors start recording in response to an R_2 message. The worst-case scenario for a write operation is similar. This implies the following proposition:

PROPOSITION 5.4.4. *At most $O(n^2)$ messages are sent during each execution of a* read *or a* write *operation.*

The constructions of bounded sequential time-stamp system [Israeli and Li 1992; Dolev and Shavit 1987] imply that a label can be represented using $O(n)$ bits (i.e., $|\mathscr{L}| = O(2^n)$). The next theorem summarizes the above discussion.

THEOREM 5.4.5. *There exists a bounded emulator of an atomic, single-writer multi-reader register in a complete network, in the presence of at most $\lfloor (n-1)/2 \rfloor$ processor failures. Each execution of a* read *operation or a* write *operation requires $O(n^2)$ messages each of size $O(n^3)$, $O(1)$ time, and $O(n^4)$ local memory.*

6. *The Bounded Implementation—Arbitrary Network*

In an arbitrary network, a processor is considered faulty if it cannot communicate with a majority of the processors, and a correctly functioning processor is guaranteed to eventually be in the same connected component with a majority of the processors. The first construction in this section is achieved by replacing every **send** operation from i to j by an execution of an end-to-end protocol between i and j. Several implementations of such a protocol are known (see Afek and Gafni [1988, 1991]; and Awerbuch et al. [1991]). An end-to-end protocol establishes traffic between i and j if there is eventually a path between them. We assume that at most $\lfloor (n-1)/2 \rfloor$ processors are faulty and therefore, at least $\lceil (n+1)/2 \rceil$ processors are connected to each other. Thus, eventually, there will be a path between any nonfaulty processor and a majority of the processors and the system behaves as in the case of complete network with processor failures.

Note that there can be labels in a disconnected part of the system that will not appear in the input of Procedure LABEL. However, these are not viable labels because the end-to-end protocol will prevent processors from adopting them as the writer's label and hence correctness is preserved.

The complexity claims in the next theorem are implied by the end-to-end protocol of Afek and Gafni [1989].

THEOREM 6.1. *There exists a bounded emulator of an atomic, single-writer multi-reader register in an arbitrary network in the presence of link failures that do not disconnect a majority of the processors. Each execution of a* read *operation or a* write *operation requires* $O(n^5)$ *messages, each of size* $O(n^3)$, *and* $O(n^2)$ *time.*

Instead of implementing each virtual link separately, we can achieve improved performance by implementing communicate directly. We make use of the fact that Afek and Gafni [1991] show how to resynchronize any diffusing computation, not only an end-to-end protocol.

More specifically, in a *diffusing computation* [Dijkstra and Scholten 1980], one node, the initiator, initiates the computation by sending messages to its neighbors. Following the reception of a message, a node may send messages to its neighbors. In this manner, the computation diffuses from the initiator. Procedure communicate can be written as a diffusing computation. The processor invoking Procedure communicate initiates the computation by sending the message to its neighbors; these, in turn, propagate the message to their neighbors. Acknowledgements are sent on the communication links that spanned the computation. Since a nonfaulty processor is eventually connected to at least $\lceil (n+1)/2 \rceil$ processors, a nonfaulty processor will receive $\lceil (n+1)/2 \rceil$ acknowledgments eventually.

The resulting implementation requires $O(n^3)$ messages per invocation of communicate. Thus, we have

THEOREM 6.2. *There exists a bounded emulator of an atomic, single-writer multi-reader register in an arbitrary network in the presence of link failures that do not disconnect a majority of the processors. Each execution of a* read *operation or a* write *operation requires* $O(n^4)$ *messages, each of size* $O(n^3)$, *and* $O(n^2)$ *time.*

Since the constructions of Afek and Gafni [1987; 1991] require unbounded local space the local space overhead of the algorithm is unbounded.

7. *Discussion and Further Research*

We have presented emulators of atomic, single-writer multi-reader registers in message-passing systems (networks), in the presence of processor or link failures. In the complete network, in the presence of processor failures, each operation to the register requires $O(n^2)$ messages, each of size $O(n^3)$, and constant time. In an arbitrary network, in the presence of link failures, each operation to the register requires $O(n^4)$ messages, each of size $O(n^3)$ and $O(n^2)$ time.

It is interesting to improve the complexity of the emulations, in either of the message-passing systems. Alternatively, it might be possible to prove lower bounds on the cost of such emulations.

An interesting direction is to emulate stronger shared memory primitives in message-passing systems in the presence of failures. Any primitive that can be implemented from wait-free, atomic, single-writer multi-reader registers, can also be implemented in message-passing systems using the emulators we have presented. This includes wait-free, atomic, multi-writer multi-reader registers, atomic snapshots, and many others. However, there are shared-memory data

structures that cannot be implemented from wait-free, atomic, single-writer multi-reader registers [Herlihy 1988]. Some of these primitives, such as **Read-Modify-Write**, can be used to solve consensus [Herlihy 1988], and thus any emulation of them in the presence of failures will imply a solution to consensus in the presence of failures. It is known [Fischer et al. 1985] that consensus cannot be solved in asynchronous systems even in the presence of one failure. Thus, we need to strengthen the message-passing model in order to emulate primitives such as **Read-Modify-Write**. Additional power can be added to the message-passing model considered in this paper by failure detector mechanisms or automatic acknowledgment mechanisms (cf. [Feldman and Nigam 1980]). We leave all of this as a subject for future work.

ACKNOWLEDGMENTS. We would also like to thank Baruch Awerbuch and Yishay Mansour for helpful discussions. We would also like to thank Tami Tamir and the anonymous referees for helpful comments.

REFERENCES

ABRAHAMSON, K. 1988. On achieving consensus using a shared memory. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing* (Toronto, Ont., Canada, Aug 15–17). ACM, New York, pp. 291–302.

AFEK, Y., ATTIYA, H., DOLEV, D., GAFNI, E., MERRITT, M., AND SHAVIT, N. 1993. Atomic snapshots of shared memory. *J. ACM 40*, 4 (Sept.), 873–890.

AFEK, Y., AWERBUCH, B., AND GAFNI, E. 1987. Applying static network protocols to dynamic networks. In *Proceedings of the 28th Symposium on Foundations of Computer Science*. IEEE, New York, pp. 358–369.

AFEK, Y., DOLEV, D., GAFNI, E., MERRITT, M., AND SHAVIT, N. 1990. A bounded first-in first-enabled-solution to the *l*-exclusion problem. In *Proceedings of the International Workshop on Distributed Algorithms*. Lecture Notes in Computer Science, vol. 486, Springer-Verlag, New York.

AFEK, Y. AND GAFNI, E. 1988. End-to-end communication in unreliable networks. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing* (Toronto, Ont., Canada, Aug. 15–17). ACM, New York, pp. 131–147.

AFEK, Y. AND GAFNI, E. 1991. Bootstrap network resynchronization. In *Proceedings of the 10th Annual ACM Symposium on Principles of Distributed Computing* (Montreal, Que., Canada, Aug. 19–21). ACM, New York, pp. 295–307.

ANDERSON, J. H. 1993. Composite registers. *Dist. Computing 6*, 3, 141–154.

ASPNES, J., AND MERLIHY, M. 1990a. Fast randomized consensus using shared memory. *J. Algorithms 15*, 1 (Sept.), 441–460.

ASPNES, J., AND HERLIHY, M. P. 1990b. Wait-free data structures in the asynchronous PRAM model. In *Proceedings of the 22nd Annual ACM Symposium on Parallel Algorithms and Architectures* (Island of Crete, Greece, July 2–6). ACM, New York, pp. 340–349.

ATTIYA, H., BAR-NOY, A., DOLEV, D., PELEG, D. AND REISCHUK, R. 1990. Renaming in an asynchronous environment. *J. ACM 37*, 3 (July), 524–548.

ATTIYA, H., DOLEV, D. AND SHAVIT, N. 1987. Bounded polynomial randomized consensus. In *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing* (Edmonton, Alb., Canada, Aug. 14–16). ACM, New York, pp. 281–293.

AWERBUCH, B. 1987. Optimal distributed algorithms for minimum weight spanning tree, counting, leader election and related problems. In *Proceedings of the 19th Symposium on Theory of Computing*. (New York, N.Y., May 25–27). ACM, New York, pp. 230–240.

AWERBUCH, B., MANSOUR, Y., AND SHAVIT, N. 1989. Polynomial end-to-end communication. In *Proceedings of the 30th Symposium of Computer Science*. IEEE, New York, pp. 358–363.

BAR-NOY, A. AND DOLEV, D. 1989. Shared-memory vs. message-passing in an asynchronous distributed environment. In *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing* (Edmonton, Alb., Canada, Aug. 14–16). ACM, New York, pp. 307–318.

BEN-OR, M. 1983. Another advantage of free choice: Completely asynchronous agreement protocols. In *Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing* (Montreal, Que., Canada, Aug. 17–19). ACM, New York, pp. 27–30.

BURNS, J. E., AND PETERSON, G. L. 1989. The ambiguity of choosing. In *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing* (Edmonton, Alb., Canada, Aug. 14–16). ACM, New York, pp. 145–157.

CHOR, B., ISRAELI, A., AND LI, M. 1987. On processor coordination using asynchronous hardware. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing* (Vancouver, B.C., Canada, Aug. 10–12). ACM, New York, pp. 86–97.

CHOR, B., MERRITT, M., AND SHMOYS, D. 1985. Simple constant-time consensus protocols in realistic failure models. In *Proceedings of the 4th Annual ACM Symposium on Principles of Distributed Computing* (Minaki, Ont., Canada, Aug. 5–7). ACM, New York, pp. 152–160.

CHOR, B., AND MOSCOVICI, L. 1989. Solvability in asynchronous environments. In *Proceedings of the 30th Symposium on Foundations of Computer Science*. IEEE, New York, pp. 422–427.

DIJKSTRA, E. W., AND SCHOLTEN, C. S. 1988. Termination detection for diffusing computations. *Inf. Proc. Lett.*, *1*, 1 (Aug)., 1–4.

DOLEV, D., GAFNI, E., AND SHAVIT, N. 1988. Toward a non-atomic era: *l*-exclusion as a test case. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing* (Chicago, Ill., May 2–4). ACM, New York, pp. 78–92.

DOLEV, D., AND SHAVIT, N. 1987. Unpublished manuscript. July. Cited in Awerbuch et al. [1989].

DOLEV, D., AND SHAVIT, N. 1989. Bounded concurrent time-stamp systems are constructible. *SIAM J. Computing*, to appear. Also: *Proceedings of the 21st Symposium on Theory of Computing* (Seattle, Wash., May 15–17), ACM, New York, pp. 454–466.

DWORK, C., SHMOYS, D., AND STOCKMEYER, L. 1986. Flipping persuasively in constant expected time. In *Proceedings of the 27th Symposium on Foundations of Computer Science*, IEEE, New York, pp. 222–232.

FELDMAN, J. A., AND NIGAM, A. 1980. A model and proof technique for message-based systems. *SIAM J. Computing 9*, 4 (Nov). 768–784.

FINN, S. G. 1979. Resynch procedures and a fail-safe network protocol. *IEEE Trans. Comm. COM-27*, 840–845.

FISCHER, M. J., LYNCH, N. A., AND PATERSON, M. S. 1985. Impossibility of distributed consensus with one faulty processor. *J. ACM 32*, 2 (Apr.), 374–382.

HERLIHY, M. P. 1988. Impossibility and universality results for wait-free synchronization. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing* (Toronto, Ont., Canada, Aug. 15–17). ACM, New York, pp. 276–290.

HERLIHY, M. P. 1991. Wait-free synchronization. *ACM Trans. Prog. Lang. Syst. 13*, 1 (Jan.), 124–149. (Earlier version appeared as: Randomized wait-free concurrent objects. In *Proceedings of the 10th Annual ACM Symposium on Principles of Distributed Computing* (Montreal, Que., Canada, Aug. 19–21). ACM, New York, pp. 11–21.

ISRAELI, A., AND LI, M. 1993. Bounded time-stamps. *Dist. Comput. 6*, 4, 205–209.

LAMPORT, L. 1986. On interprocess communication: Part I and II. *Dist. Comput. 1*, 77–101.

LI, M., TROMP, J. AND VITANYI, P. 1989. How to share concurrent wait-free variables. Report CS-R8916. CWI, Amsterdam, The Netherlands.

PETERSON, G. L., AND BURNS. J. E. 1987. Concurrent reading while writing II: The multiwriter case. In *Proceedings of the 28th Symposium on Foundations of Computer Science*. IEEE, New York, pp. 383–392.

VISHKIN, U. 1983. A distributed orientation algorithm. *IEEE Trans. Inf. Theory* (June).

VITANYI, P., AND AWERBUCH, B. 1986. Atomic shared register access by asynchronous hardware. In *Proceedings of the 27th Symposium on Foundations of Computer Science*. IEEE, New York, pp. 233–243.

RECEIVED MAY 1990; REVISED JANUARY 1994; ACCEPTED FEBRUARY 1994