Sharing Memory Robustly in Message-Passing Systems

Hagit Attiya[1] Amotz Bar-Noy[2] Danny Dolev[3]

February 16, 1990

[1]Laboratory for Computer Science, MIT, Cambridge, MA 02139. Supported by NSF grant no CCR-8611442, by ONR contract no N00014-85-K-0168, and by DARPA contracts no N00014-83-K-0125 and N00014-89-J-1988.

[2]IBM T. J. Watson Research Center, P. O. Box 704, Yorktown Heights, NY 10598. Part of the work was done while the author was at the Computer Science Department, Stanford University, Stanford, CA 94305. Supported in part by a Weizmann fellowship, by contract ONR N00014-88-K-0166 and a grant of Stanford's Center for Integrated Systems.

[3]IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120, and the Computer Science Department, Hebrew University, Jerusalem.

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. The overhead introduced by these emulations is polynomial in the number of processors in the systems.

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 a class of data structures.

Keywords: Message passing, shared memory, dynamic networks, fault tolerance, wait-free algorithms, emulations, atomic registers.

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 atomic 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 process 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 (or *wait-free*) 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 by any other processor will obtain the same value. Furthermore, the value of a shared 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, e.g., [16, 19, 25], and in the shared-memory model, e.g., [18, 1, 9, 12]. However, the algorithm of [9] 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]

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

[1]The asynchronous message-passing algorithm of [26] is resilient to Byzantine faults, but requires private communication links and thus is not resilient to an omnipotent adversary.

both message-passing models. The overhead for the emulations is polynomial in the number of processors. The complexity measures considered are the number of messages and their size, the time and the local memory size for each read or write operation.

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 ([36, 34]).
- Concurrent time-stamp systems ([31, 24]).
- Variants of ℓ-exclusion ([22, 17, 4]).
- Atomic snapshot scan ([2, 7, 8]).
- Randomized consensus ([9, 12]).[2]
- Implementation of a class of data structures ([10]).

First we introduce the basic communication primitive which is used in our algorithms. We then present an *unbounded* emulator for the complete network in the presence of processor failures. This implementation exposes some of the basic ideas underlying our constructions. Moreover, part of the correctness proof for this emulator can be carried 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 ([5, 14, 6]). 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 [6].

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

[2]This result also follows from the transformation of [15].

We believe that bounded solutions are important, although in reality, 20 bits 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 [11], 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 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 [11], similar arguments show that processors cannot halt after deciding. Thus, in our emulators a processor which is disconnected (permanently) from a majority of the processors is considered *faulty* and is blocked.[3] Our solutions do not depend on connection with a *specific* majority at any time. Moreover, it might be that at no time there exists a full connection to any party. The only condition is that messages will eventually reach some majority which 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* ([31]).

Some of the previous research on dynamic networks (e.g., [28, 3]) assumed a "grace period" during which the network stabilizes for 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, e.g., [35, 5, 14, 6].

There are two related studies on the relationships between shared-memory and message-passing systems. Bar-Noy and Dolev ([15]) provide translations between protocols in the shared-memory and the message-passing models. These translations apply only to protocols that use a very restricted form of communication. Chor and Moscovici ([20]) present a hierarchy of resiliency for problems in shared-memory systems and complete networks, and show that

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

for some problems, the wait-free shared-memory model is not equivalent to complete network, where up to half of the processors may fail. Their result, however, assumes that processors *halt* after deciding.

The rest of this paper is organized as follows. In Section 2, we briefly describe the various models considered. In Section 3, we introduce the communication primitive. In Section 4, we present an unbounded implementation for complete network 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 in the presence of link 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 [32] for shared-memory systems, [29] for complete networks with processor failures, and [14] for arbitrary networks with link failures. In all models we consider, a system consists of n independent and asynchronous processors, which we number $1, \ldots, n$.

A formal definition of an atomic register can be found in [32], the definition presented here is an equivalent one (see [32, Proposition 3]) which is simpler to use. An atomic, single-writer multi-reader register is an abstract data structure. Each register is accessed by two procedures, $\mathsf{write}_w(v)$ which is executed only by some specific processor w, called the *writer*, and $\mathsf{read}_r(v)$ which may be executed by any processor $1 \leq r \leq n$, called a *reader*. It is assumed that the values of these procedures satisfy the following two properties:

1. Every read operation returns either the last value written or a value that is written concurrently with this read.
2. If a read operation $\mathcal{R}_2$ started after a read operation $\mathcal{R}_1$ has finished, then the value $\mathcal{R}_2$ returns cannot be older than the value returned by $\mathcal{R}_1$.

In message-passing systems, processors are located at the nodes of a network and communicate by sending messages along communication links. Communication is completely *asynchronous* and messages may incur an unknown delay. At each atomic step, a processor may receive some set of messages that were sent to it, perform some local computation and send some messages.

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. A *nonfaulty* processor is one that takes an infinite number of steps, and all of its messages are delivered after a finite delay. We assume that at most $\lfloor\frac{n-1}{2}\rfloor$ processors are faulty in any execution of the system.

In dynamic networks communication links might become *non-viable*. A link is non-viable, if, starting from some message and on, it will not deliver any further messages to the other end-point. For those messages the delay is considered to be infinite. Otherwise, the link is *viable*. This model is called the ∞-*delay model* in [5]. Afek and Gafni ([5]) point out that the standard model of dynamic message-passing systems, where communication links alternate between periods of operation and non-operation, can be reduced to this model. A processor that is permanently disconnected from $\lceil\frac{n}{2}\rceil$ processors or more is considered *faulty*. We assume there are $\lceil\frac{n+1}{2}\rceil$ processors that are eventually in the same connected component. Thus, at most $\lfloor\frac{n-1}{2}\rfloor$ processors are faulty.

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,
3. the time it takes to execute a write or read operation, under the assumption that any message is either delivered within one time unit, or never at all (cf. [13]), and
4. the amount of the overhead 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 acknowledgements (possibly carrying some information) from a majority of the processors.

Because of possible processors' crash failures, a processor cannot wait for acknowledgements from all the other processors or from any particular processor. However, at least a majority of the processors will not crash and thus a processor can wait to get acknowledgements from them. Notice that processors want to communicate with any majority of the processors, not necessarily the same majority each time. A processor utilizes 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 acknowledgements 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 *my* or *his*. If $turn(j) = my$ then it is i's turn on $\langle i, j \rangle$ and only then i may send a message to j. If $turn(j) = his$ then either i's message is in transit, j's acknowledgement is in transit, or j received i's message and has not replied yet (it might be that j crashed). Initially, $turn(j) = my$. Hereafter, we assume that the vector *turn* is updated automatically by the **send** and **receive** operations.[4] For simplicity, a processor sends each message also to itself and responds with the appropriate acknowledgement.

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 acknowledgement (or $\perp$ if no acknowledgement 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 $turn(j) = his$); *notack*, meaning M was sent but not yet acknowledged by j; *ack*, meaning M was acknowledged by j. Additional local variables in procedure **communicate** are the vector *turn* and the integer counter *#acks* which counts the number of acknowledgements received so far.

The pseudo-code for this procedure appears in Figure 1. We note that whenever this procedure is employed we also specify its companion procedure, **ack**, which specifies the information sent with the acknowledgement 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 acknowledgements stored in the output vector *info* were indeed sent as acknowledgements to the message M, i.e., at least $\lceil \frac{n+1}{2} \rceil$ processors received the message M.

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

```
Procedure communicate($\langle M\rangle; info$); (* for processor $i$ *)
        $\#acks := 0$;
        for all $1 \leq j \leq n$ do
            $status(j) := notsent$ ;
            $info(j) := \perp$ ;
        for all $1 \leq j \leq n$ s.t. $turn(j) = my$ do
            send $\langle M\rangle$ to $j$ ;
            $status(j) := notack$ ;
        repeat until $\#acks \geq \lceil \frac{n+1}{2} \rceil$
            upon receiving $\langle m\rangle$ from $j$:
                if $status(j) = notsent$ then
                (* acknowledgement of an old message *)
                  send $\langle M\rangle$ to $j$ ;
                  $status(j) := notack$;
                else if $status(j) = notack$ then
                  $status(j) := ack$ ;
                  $info(j) := m$ ;
                  $\#acks := \#acks + 1$ ;
end procedure communicate;
```

Figure 1: *The procedure* communicate.

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 the message $\langle M \rangle$:*

1. *if i is connected to at least a majority of the processors then the execution terminates,*
2. *at least $\lceil \frac{n+1}{2} \rceil$ processors receive $\langle M \rangle$ and return the corresponding acknowledgement,*
3. *at most $2n$ messages are sent during this execution,*
4. *the procedure terminates after at most two time units, and*
5. *the size of i's local memory is $O(n)$ times the size of the acknowledgements 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 acknowledgements from a majority of the processors. In order to read a value, the reader sends a request to all processors and gets in return the latest values 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 value it intends to adopt to at least a majority of the processors (again by using communicate).

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 the label.

Processor i stores in its local memory a variable val_i, holding the most recent value of the register known to i. This value 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 of the most recent values of the register sent to i by other processors. Let $\mathcal{V}$ denote the number of bits needed to represent any value from the domain of all possible values, we have

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

In the implementation, there are three procedures: **read** for the reader, **write** for the writer, and **ack**, used by all processors to respond to messages. These procedures utilize six types of messages, arranged in three pairs, each consisting of a message and a corresponding acknowledgement.

1. The pair of write messages.

   $\langle W, val\rangle$: sent by the writer in order to write *val* in its register.

   $\langle ACK\text{-}W\rangle$: the corresponding acknowledgement.

2. The first pair of read messages.

   $\langle R_1\rangle$: sent by the reader to request the recent value of the writer.

   $\langle val\rangle$: the corresponding acknowledgement, contains the sender's most updated value of the register.

3. The second pair of read messages.

   $\langle R_2, val\rangle$: sent by the reader before terminating in order to announce that it is going to return *val* as the value of the register.

   $\langle ACK\text{-}R_2\rangle$: the corresponding acknowledgement.

Clearly, we have

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

The descriptions of procedures **write**, **read** and **ack** appear in Figure 2. Procedure **ack** instructs each processor what to do upon receiving a message according to the template in Figure 1 (as explained in Section 3). We use *void* to say that the information sent with the acknowledgements to a particular message is ignored. Since communication is done only by **communicate**, Lemma 3.1 (part 1) implies

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

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

```
Procedure read_i($val_i$); (* executed by processor $i$ and returns $val_i$ *)
        communicate($\langle R_1 \rangle$, $info$);
        $val_i := \max_{1 \le j \le n}\{info(j) \mid info(j) \neq \perp\}$;
        communicate($\langle R_2, val_i \rangle$, $void$);
end procedure read_i ;

Procedure write_w; (* for the writer $w$ *)
        $val_w := val_w + 1$; (* the new value of the register *)
        communicate($\langle W, val_w \rangle$, $void$);
end procedure write_w;

Procedure ack_j; (* executed by processor $j$ *)
        case received from $w$
            $\langle W, val_w \rangle$:  $val_j := \max\{val_w, val_j\}$ ;
                          send $\langle ACK\text{-}W \rangle$ to $w$;
        case received from $i$
            $\langle R_1 \rangle$:        send $\langle val_j \rangle$ to $i$;
            $\langle R_2, val_i \rangle$:  $val_j := \max\{val_i, val_j\}$ ;
                          send $\langle ACK\text{-}R_2 \rangle$ to $i$;
end procedure ack_j;
```

Figure 2: *The* read, write *and* ack *procedures of the unbounded emulator.*

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

Proof: By Lemma 3.1 (part 2) and the code for ack, when $\mathcal{C}_1$ is completed at least majority of the processors store x', such that $x' \geq x$. Similarly, by Lemma 3.1 (part 2), in $\mathcal{C}_2$ acknowledgements were received from at least a majority of the processors. Thus, there must be at least one processor that stored a value $x' \geq x$ and acknowledged in $\mathcal{C}_2$. Since y is maximal among the values contained in the acknowledgements of $\mathcal{C}_2$, it follows that $y \geq x' \geq x$. ■

Since a write operation completes only after its communicate procedure completes, Lemma 4.4 implies

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

Lemma 4.6 *Assume some* read *operation, $\mathcal{R}_1$, returns the value x, and that another* read *operation, $\mathcal{R}_2$, that started after $\mathcal{R}_1$ completed, returns y. Then $x \leq y$.*

Since processors communicate only by using the communicate procedure, 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 $\lfloor \frac{n-1}{2} \rfloor$ 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 above emulation is the integer labels utilized by the writer. In order to eliminate this, we use an idea which was employed previously in [31, 14]. The integer labels are replaced by *bounded sequential time-stamp system* ([31]), which is a finite domain $\mathcal{L}$ of label values together with a total order relation $\prec$. Whenever the writer needs a new label it produces a new one, larger (with respect to the $\prec$ order) than all the labels that exist in the system. Thus, instead of just adding one to the label, as in the unbounded emulation, here 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 which is greater than all the labels in this set. This can be achieved by the constructions presented in [31, 23] for bounded sequential time-stamp systems.

The main difficulty in carrying this idea over to the message-passing model is in maintaining the set of labels existing in the system, a task which need not be addressed in the shared-memory model (cf. [31, 33]). Notice that in order to assure correctness, it suffices to guarantee that the set of labels that exist in the system is contained in the input set of labels of procedure LABEL. The key idea is as follows.

Whenever a processor adopts a label (as the maximum value of the writer it knows about), it *records* this fact in the system. This is done by broadcasting an appropriate message and waiting for acknowledgements from a majority of the processors (using communicate). Upon receiving a recording message, a processor stores the information it contains in its local memory, but ignores the values it carries. This process guarantees that labels do not get lost as a majority of the processors have recorded them.

To avoid inconsistencies that might occur, a processor blocks all computation that is related to new labels during the recording process. It does not adopt new labels and does not send nonrecording messages containing new labels. An independent ping-pong mechanism is employed for each type of messages, e.g., i may send a recording message to j although j did not acknowledge a read message of i. Since recording messages do not cause a processor to adopt a label, deadlock is avoided.

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 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 and the field $L_i(i,k).ack$ is the last label i sent to k as an acknowledgement 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 acknowledgements.

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 acknowledgement, L is the sender's updated value of the labels' matrix.

2. The second pair of write messages, $\langle W_2, val \rangle$ and $\langle ACK\text{-}W_2 \rangle$, the first pair of read messages, $\langle R_1 \rangle$ and $\langle val \rangle$, and the second pair of read messages, $\langle R_2, val \rangle$ 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 value for the register, processor i sends $L_i(i)$ to other processors. The vector $L_i(i)$ contains this new value and all the recent values that i sent on its links to other processors.

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

The longest message is $\langle L \rangle$, denote $\mathcal{V} = \log |\mathcal{L}|$. we have,

Proposition 5.1 *The maximum size of a message is $O(n^2 \cdot \mathcal{V})$.*

Recall that during the recording process, processors do not reply to nonrecording messages. Therefore, messages are accumulated in the local memory of the processor and are ordered in a queue. As soon as the recording process ends, the processor first handles the messages on the queue.[5] Due to the ping-pong mechanism the length of this queue is at most $O(n)$. As each message on the queue contains (at most) a vector of n labels and the matrix L_i is $O(n^2 \cdot \mathcal{V})$, we have,

[5]The details of how this queue is handled are omitted.

Proposition 5.2 *The size of the local memory of a reader is $O(n^2 \cdot \mathcal{V})$. The size of the local memory of a writer is $O(n^3 \cdot \mathcal{V})$.*

5.3 The Algorithm

The pseudo-code for the algorithm appears in Figure 3. Procedure update and the first part of procedure recording update dynamically the vector $L_i(i)$. Therefore, in procedure read, it is enough to take val_i as $L_i(i,i)$. The flag *blocked* is set to *true* during the recording process and prevents the processor from receiving or sending some messages as described in procedure ack. As mentioned before, in order to prevent deadlocks a separate ping-pong mechanism is employed for each type of message. In order to distinguish between the different mechanisms, calls to communicate are subscripted with the message type.

5.4 Correctness and Complexity

Atomicity of the bounded emulator follow from the same reasoning as in the unbounded case (Lemma 4.5 and Lemma 4.6). The following lemma is the core of the correctness proof for the bounded emulator—it assures that the writer always obtain a superset of the labels that might be adopted as the register's value by some processor. We call a label x *viable*, if in some system state, at some possible extension from this state, for some processor i, $val_i = x$. Intuitively, a viable label is held by some processor as the current register's value or it will become the current register's value for some processor.

Lemma 5.3 *Each viable label is stored either in the writer matrix or in the matrices of at least a majority of the processors.*

Proof: We say that processor i is *responsible* for label x, if x is stored in $L_i(i)$, i.e., if either $L_i(i,i) = x$, $L_i(i,j).sent = x$ or $L_i(i,j).ack = x$. We first claim that for any viable label there exists a processor that is responsible for it. Assume that x is a label that is held by i as the current register's value, then by the code of the algorithm $L_i(i,i) = x$ and by definition i is responsible for x. Assume x will become the current register's value for processor j in the future, then it must be that some processor i has sent it to j (either by R_2 (W_2) messages of i or in response to an R_1 request message by j) thus $x \in L_i(i,j)$.

Now assume that i is responsible for x. Look at a simple path on which the label x has arrived at i, i.e., 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, $1 \leq \ell \leq m$, processor i_ℓ adopted x as a result of a message from $i_{\ell-1}$.

```
Procedure read$_i$($val_i$) ; (* executed by processor $i$ and returns $val_i$ *)
        communicate$_R$($\langle R_1 \rangle$, $info$) ;
        $val_i := L_i(i,i)$ ;
        communicate$_R$($\langle R_2, val_i \rangle$, $void$) ;
end procedure read$_i$ ;

Procedure write$_w$; (* for the writer $w$ *)
        communicate$_W$($\langle W_1 \rangle$, $L$) ;
        $L_w(w,w) :=$ LABEL($\bigcup L$) ; (* all the non-empty entries in $L$ *)
        communicate$_W$($\langle W_2, L_w(w,w) \rangle$, $void$) ;
end procedure write$_w$;

Procedure recording$_i$ ; (* executed by processor $i$ *)
        upon receiving new label $x > L_i(i,i)$:
            $blocked := true$ ;
            $L_i(i,i) := x$;
            communicate$_{REC}$($\langle REC, L_i(i) \rangle$, $void$) ;
            $blocked := false$ ;
end procedure recording$_i$ ;

Procedure update$_i$ ; (* executed by processor $i$ *)
        upon sending label $x$ to $j$ in $i$'s read operation:
        $L_i(i,j).sent := x$ ;
        upon sending label $x$ to $j$ in $j$'s read operation:
        $L_i(i,j).ack := x$ ;
end procedure update$_i$ ;

Procedure ack$_j$; (* executed by processor $j$ *)
        case received from $w$
            $\langle W_1 \rangle$:              send $\langle L_j \rangle$ to $w$;
            $\langle W_2, val_w \rangle$:       if $val_w > L_j(j,j)$ then wait until $blocked = false$ ;
                                 send $\langle ACK\text{-}W_2 \rangle$ to $w$ ;
        case received from $i$
            $\langle R_1 \rangle$:              wait until $blocked = false$ ;
                                 send $\langle L_j(j,j) \rangle$ to $i$;
            $\langle R_2, val_i \rangle$:       if $val_i > L_j(j,j)$ then wait until $blocked = false$ ;
                                 send $\langle ACK\text{-}R_2 \rangle$ to $i$;
            $\langle REC, L_i(i) \rangle$:      $L_j(i) := L_i(i)$ ;
                                 send $\langle ACK\text{-}REC \rangle$ to $i$;
end procedure ack$_j$;
```

Figure 3: *The* read, write, recording, update *and* ack *procedures of the bounded emulator.*

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 codes of procedures **update** and **write** imply that x is stored in i's matrix. For the induction step, assume that $m > 0$, and that the induction hypothesis holds for any ℓ, $0 \leq \ell < m$. We have two cases.

1. The first case is when i has not finished the recording process for x. It follows from the code of procedure **recording** that $L_i(i,i) = x$. We show that $k = i_{m-1}$ is responsible for x, and the lemma follows from the induction hypothesis.

   If i received x from k through an R_2 (W_2) message, then since i is blocked during the recording process it would not reply until the recording process of x is done. Consequently, $L_k(k,i).sent = x$.

   If i received x from k through an $ACK\text{-}R_1$ message, then since i would not terminate a **read** operation until it finishes the recording process of x, it would not start a new **read** operation. Consequently, $L_k(k,i).ack = x$.

2. The second case is when i has finished the recording process for x. If $L_i(i,i) = x$, i.e., x is still the current value that i holds, then the code for procedure **record**, and the properties of procedure **communicate** (Lemma 3.1, part 2) imply that x is stored in the matrices of at least a majority of the processors.

   If $L_i(i,i) \neq x$, then since i is responsible for x there must exist a j such that $x \in L_i(i,j)$. Furthermore, since i has a more recent value for the register it must be that $L_i(i,i) = y \succ x$. By the code for procedure **recording** and the properties of procedure **communicate** (Lemma 3.1), at the end of the recording process for x, x is stored as $L(i,i)$ in the matrices of at least a majority of the processors. Let k be some processor that recorded x for i, i.e., such that $L_k(i,i) = x$ at the end of the recording process for x.

   If currently, $L_k(i,i) = z \neq x$ then it must be that $x \prec z$. Since forwarding a new value is blocked during the recording process, it must be that x was sent by i to j before the recording process for z started. Thus $x \in L_i(i,j)$ during the recording process for z, and consequently $x \in L_k(i,j)$. Therefore, x appears in the matrices of a majority of the processors.

■

Lemma 5.3 and the constructions of bounded sequential time-stamp systems of [31, 23] imply

Corollary 5.4 *The new label generated by procedure* LABEL *is greater than any viable label in the system.*

Recording messages are acknowledged immediately and are never blocked. Thus, a processor never deadlocks during a recording process and will eventually acknowledge all the messages it receives. The next lemma follows since during a read or a write operation, at most $2n$ recording processes could occur.

Lemma 5.5 *Each execution of a* read *operation or a* write *operation terminates.*

Each acknowledgement the reader receives might cause it to initiate a recording process. By Lemma 3.1, part 3, at most $2n$ messages are sent during each of these recording processes. In addition, each message of type W_2 or R_2 might cause other processors to initiate a recording process. Thus, at most $O(n^2)$ messages are sent during each execution of an operation, and it takes at most $O(1)$ time units. Thus we have

Proposition 5.6 *At most $O(n^2)$ messages are sent during each execution of a* read *or a* write *operation.*

Proposition 5.7 *Each execution of a* read *or* write *operation takes at most 6 time units.*

The constructions of bounded sequential time-stamp system ([31, 23]) imply that a label can be represented using $O(n)$ bits. The next theorem summarizes the above discussion.

Theorem 5.8 *There exists a bounded emulator of an atomic, single-writer multi-reader register in a complete network, in the presence of at most $\lfloor \frac{n-1}{2} \rfloor$ processor failures. Each execution of a* read *operation or a* write *operation requires $O(n^2)$ messages each of size $O(n)$, $O(1)$ 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 be eventually 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. Implementations of such a protocol are known (see [5, 14, 6]). An end-to-end protocol establishes traffic between i and j if there is eventually a path between them. In our case, eventually there will be a path between any nonfaulty processor and a majority of the processors, thus the system behaves as in the case of complete network with processor failures.

Note that there are labels in 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 [6].[6]

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 the 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 $O(n)$, 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 ([6]) show how to resynchronize *any diffusing computation* ([21]), not only an end-to-end protocol. Although the task achieved by communicate is not exactly a diffusing computation, we can modify the algorithm of [6], by "piggybacking" acknowledgement information. 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 the 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 $O(n)$, and $O(n^2)$ time.*

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 $O(n)$, 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 $O(n)$, and $O(n^3)$ 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.

[6]Any improvement in the complexity of the end-to-end protocol will immediately result in an improvement to the complexity of our implementation.

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 be also 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 ([30]). Some of these primitives, such as Read-Modify-Write, can be used to solve consensus ([30]), 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 ([29]) 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 primitive such as Read-Modify-Write. Additional power can be added to the message-passing model considered in this paper by, e.g., failure detection mechanisms or automatic acknowledgement mechanisms (cf. [27]). We leave all of this as a subject for future work.

Acknowledgements:

We would like to thank Baruch Awerbuch and Yishay Mansour for helpful discussions.

References

[1] K. Abrahamson, On Achieving Consensus Using a Shared Memory, *Proc. 7th ACM Symp. on Principles of Dist. Computing*, pp. 291–302, 1988.

[2] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit, Atomic Snapshots of Shared Memory, *manuscript*.

[3] Y. Afek, B. Awerbuch, and E. Gafni, Applying Static Network Protocols to Dynamic Networks, *Proc. 28th Symp. on Foundations of Comp. Science*, pp. 358–369, 1987.

[4] Y. Afek, D. Dolev, E. Gafni, M. Merritt and N. Shavit, A Bounded First-In First-Enabled-Solution to the ℓ-Exclusion Problem, *manuscript*.

[5] Y. Afek, and E. Gafni, End-to-End Communication in Unreliable Networks, *Proc. 7th ACM Symp. on Principles of Dist. Computing*, pp. 131–147, 1983.

[6] Y. Afek, and E. Gafni, Bootstrap Network Resynchronization: An Efficient Technique for End-to-End Communication, *manuscript*.

[7] J. H. Anderson, Composite Registers, TR-89-25, Department of Computer Science, The University of Texas at Austin, September 1989.

[8] J. H. Anderson, Multiple-Writer Composite Registers, TR-89-26, Department of Computer Science, The University of Texas at Austin, September 1989.

[9] J. Aspnes, and M. Herlihy, Fast Randomized consensus Using Shared Memory, *Technical Report*, Computer Science Department, Carnegie Mellon University, 1988.

[10] J. Aspnes, and M. P. Herlihy, Wait-Free Data Structures in the Asynchronous PRAM model, *private communication*.

[11] H. Attiya, A. Bar-Noy, D. Dolev, D. Koller, D. Peleg, and R. Reischuk, Achievable Cases in an Asynchronous Environment, *Proc. 28th Symp. on Foundations of Comp. Science*, pp. 337–346, 1987.

[12] H. Attiya, D. Dolev and N. Shavit, Bounded Polynomial Randomized Consensus, *Proc. 8th ACM Symp. on Principles of Dist. Computing*, pp. 281–293, 1989.

[13] B. Awerbuch, Optimal Distributed Algorithms for Minimum Weight Spanning Tree, Counting, Leader Election and Related Problems, *Proc. 19th ACM SIGACT Symp. on Theory of Computing*, pp. 230–240, 1987.

[14] B. Awerbuch, Y. Mansour, and N. Shavit, Polynomial End-To-End Communication, *Proc. 30th Symp. on Foundations of Comp. Science*, pp. 358–363, 1989.

[15] A. Bar-Noy, and D. Dolev, Shared-Memory vs. Message-Passing in an Asynchronous Distributed Environment, *Proc. 8th ACM Symp. on Principles of Dist. Computing*, pp. 307–318, 1989.

[16] M. Ben-Or, Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols, *Proc. 2nd ACM Symp. on Principles of Dist. Computing*, pp. 27–30, 1983.

[17] J. E. Burns and G. L. Peterson, The Ambiguity of Choosing, *Proc. 8th ACM Symp. on Principles of Dist. Computing*, pp. 145–157, 1989.

[18] B. Chor, A. Israeli, and M. Li, On Processor Coordination Using Asynchronous Hardware, *Proc. 6th ACM Symp. on Principles of Dist. Computing*, pp. 86–97, 1987.

[19] B. Chor, M. Merritt, and D. Shmoys, Simple Constant-Time Consensus Protocols in Realistic Failure Models, *Proc. 4th ACM Symp. on Principles of Dist. Computing*, pp. 152–160, 1985.

[20] B. Chor, and L. Moscovici, Solvability in Asynchronous Environments, *Proc. 30th Symp. on Foundations of Comp. Science*, pp. 422–427, 1989.

[21] E. W. Dijkstra and C. S. Scholten, Termination Detection for Diffusing Computations, *Information Processing Letters*, Vol. 1, No. 1 (August 1980), pp. 1–4.

[22] D. Dolev and E. Gafni and N. Shavit, Toward a Non-Atomic Era: ℓ-Exclusion as a Test Case, *Proc. 20th ACM Symp. on Theory of Computation*, pp. 78–92, 1988.

[23] D. Dolev, and N. Shavit, *unpublished manuscript*, July 1987. Appears in [14].

[24] D. Dolev, and N. Shavit, Bounded Concurrent Time-Stamp Systems are Constructible, *Proc. 21st ACM SIGACT Symp. on Theory of Computing*, pp. 454–466, 1989.

[25] C. Dwork, D. Shmoys, and L. Stockmeyer, Flipping Persuasively in Constant Expected Time, *Proc. 27th Symp. on Foundations of Computer Science*, pp. 222–232, 1986.

[26] P. Feldman, *private communication*.

[27] J. A. Feldman and A. Nigam, A Model and Proof Technique for Message-Based Systems, *SIAM J. on Computing*, Vol. 9, No. 4 (November 1980), pp. 768–784.

[28] S. G. Finn, Resynch Procedures and a Fail-Safe Network Protocol, *IEEE Trans. Comm.*, COM-27 pp. 840–845, 1979.

[29] M.J. Fischer, N.A. Lynch, and M.S. Paterson, Impossibility of Distributed Consensus with one Faulty Processor, *Journal of the ACM*, **32** pp. 374–382, 1985.

[30] M. P. Herlihy, Wait Free Implementations of Concurrent Objects, *Proc. 7th ACM Symp. on Principles of Distributed Computing*, 1988, pp. 276–290.

[31] A. Israeli, and M. Li, Bounded Time-stamps, *Proc. 28th Symp. on Foundations of Comp. Science*, pp. 371–382, 1987.

[32] L. Lamport, On Interprocess Communication, Part I and II, *Distributed Computing* **1** pp. 77–101, 1986.

[33] M. Li, J. Tromp and P. Vitanyi, How to Share Concurrent Wait-Free Variables, Report CS-R8916, CWI, Amsterdam, April 1989. Preliminary version in *ICALP*, 1989.

[34] G. L. Peterson and James E. Burns, Concurrent Reading While Writing II: The Multi-writer Case, *Proc. 28th Symp. on Foundations of Comp. Science*, pp. 383–392, 1987.

[35] U. Vishkin, A Distributed Orientation Algorithm, *IEEE Trans. on Information Theory*, June 1983.

[36] P. Vitanyi, and B. Awerbuch, Atomic Shared Register Access by Asynchronous Hardware, *Proc. 27th Symp. on Foundations of Comp. Science*, pp. 233–243, 1986.