Renaming in an Asynchronous Environment

HAGIT ATTIYA, AMOTZ BAR-NOY, AND DANNY DOLEV

Hebrew University, Jerusalem, Israel

DAVID PELEG

Stanford University, Stanford, California

AND

RÜDIGER REISCHUK

Technische Hochschule Darmstadt, W. Germany

Abstract. This paper is concerned with the solvability of the problem of processor renaming in unreliable, completely asynchronous distributed systems. Fischer et al. prove in [8] that "nontrivial consensus" cannot be attained in such systems, even when only a single, benign processor failure is possible. In contrast, this paper shows that problems of processor renaming can be solved even in the presence of up to $t < n/2$ faulty processors, contradicting the widely held belief that no nontrivial problem can be solved in such a system. The problems deal with *renaming processors* so as to reduce the size of the initial name space. When only uniqueness of the new names is required, we present a lower bound of $n + 1$ on the size of the new name space, and a renaming algorithm that establishes an upper bound on $n + t$. If the new names are required also to preserve the original order, a tight bound of $2^t(n - t + 1) - 1$ is obtained.

Categories and Subject Descriptors: C.2.2 [**Computer-Communication Networks**]: Network protocols

General Terms: Algorithms, Reliability

Additional Key Words and Phrases: Asynchrony, asynchronous environment, lower bounds, processor renaming, symmetry breaking, unreliable systems

1. *Introduction*

The problem of reaching agreement in unreliable distributed systems has been the subject of extensive research. In the fundamental paper [8], Fischer et al. prove that deterministic processors cannot attain "nontrivial consensus" in any asynchronous system in which processors may fail. Their result holds even when it is

Part of this work was done while A. Bar-Noy, D. Dolev, and D. Peleg were visiting the IBM Almaden Research Center in San Jose, California. H. Attiya was supported by a doctoral fellowship from the Liebniz Center. D. Peleg was supported in part by a Weizmann fellowship and by contract of the Office of Naval Research N00014-85-C-0731.

Authors' current addresses: H. Attiya, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139; 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; D. Peleg, Department of Applied Mathematics, The Weizmann Institute, Rehovot 76100, Israel; R. Reischuk, Technische Hochschule Darmstadt, West Germany.

guaranteed that at most *one* processor may fail, and even when this failure is a benign *failstop* failure (i.e., a processor can fail only by stopping). This negative result, and later stronger versions of it [6], created the widely held impression that in the presence of faults one cannot expect to solve asynchronously *any* nontrivial problem that requires participation and coordination of several processors. Attention turned to the design of probabilistic algorithms, thus relaxing the determinism [3, 4, 12], and to attaining consensus in semi-synchronous systems, thus restricting the asynchrony [1, 7].

Contradicting this widely held belief, we show that there are nontrivial problems that can be solved in unreliable asynchronous systems. These new results, along with the observations of [9] and the recent results of [5] may well lead to the revival of the original research trend, namely, exploring and charting the border between the attainable and the unattainable in a completely asynchronous environment in the presence of faults.

The problem we examine is processor renaming. The *renaming problem* can be informally defined as follows. Each of the n processors initially has a (distinct) name taken from an unbounded ordered domain. (Throughout we identify the processors with their old names, and denote them by p, q, etc.) We want an algorithm that allows each correct (nonfaulty) processor to choose (irreversibly) a new name from a space of size N, so one of the following requirements hold:

(1) *Uniqueness.* Every two new names are distinct.
(2) *Order-preserving.* If $p < q$, then the new name of p is smaller than the new name of q.

It is required that N does not depend on the original names, but only on n and on an upper bound t on number of possible faults. Naturally, we are interested in having an algorithm requiring the smallest possible N.

We refer to the version of the problem requiring the first property as the *uniqueness* problem, and to the version with the second property as the *order-preserving* problem. (Note that since the original names of processors are distinct, the order-preserving condition implies the uniqueness property.)

There are several reasons for studying this particular problem. The basic difficulties in solving the renaming problem are overcoming uncertainties and breaking symmetries. Both difficulties are inherent in many distributed problems. Intuitively, the reason one can solve the renaming problem is the room left for maneuvering; a "dead" or "seemingly dead" processor can affect two reserved names, while the rest of the processors share the others. In the consensus problem we do not have that, and every single processor may affect the result in some runs of the algorithm.

One motivation for studying the renaming problem, is the fact that the complexity (in both communication and time) of a distributed algorithm depends, sometimes, on the size of the name-space of the processors (cf. [11]). Using shorter names as processor identifiers in messages results in shorter messages and hence smaller message complexity. In particular, replacing names taken from an unbounded domain by bounded-length names lets one to effectively bound the message complexity of algorithms.

It is important to notice that in an asynchronous environment processors should be careful about dropping their old names and starting to use the new ones. This is especially important in cases where names are used for (continuous) identification. In such cases, there are several points that need to be taken care of. To begin with, there should be clear distinction between old and new names (say, by using a different field length for names), to prevent the possibility of confusing the old name of one processor with the new name of another. Secondly, even after a processor p has decided a new name x and sent all other processors messages announcing x, it cannot start signing its messages to another processor q by x until it receives from q an acknowledgment for the announcement message. Luckily, this is guaranteed to happen within finite time after renaming, if both are nonfaulty, and practically will happen by the next communication between them. A third requirement is that all processors must keep in memory both the old and the new names of all processors, and may erase the old name of a processor p only after observing that all other processors already know of the new name (which, of course, may never happen). Hence, renaming may not necessarily reduce memory space requirements. Note, however, that assuming a FIFO ordering on the arrival of messages is sufficient to eliminate this difficulty and enable processors to drop (and forget) their old names immediately after announcing their new choice.

Sometimes, names serve a more restricted function than identification. For instance, processors may use their names only as "tags" for marking their presence (say, in some priority queue). Other processors do not need to know exactly which processor has placed the tag, if they know it is not their own. For such applications, a processor may start using its new name as a tag immediately after deciding it, without having to make sure that other processors are aware of the change. (Note that for such applications, the renaming algorithm *does* need to ensure that no two processors select the same new name, and also that new names are distinct from the old ones.)

In fact, this type of application can be carried even farther. The methods developed here for the renaming problem can be combined with the idea of using processors' names as tags to yield a fault-tolerant critical section management algorithm. Suppose that the system contains some k copies of a global resource (say printers). One would like to have an algorithm managing the access of processors to the resource, with at most one processor using a copy at a time, according to some safety and fairness rules. This can be done by introducing "permits" numbered 1 to k. A processor has to get hold of a permit to enter the critical section and use the appropriate resource. The problem of managing the circulation of the permits can be viewed as a "repetitive" variant of the renaming problem in which processors decide new names (or equivalently, obtain permits), use the resource and then release their permits, and so on. This approach to the fault-tolerant critical section problem is presented and developed further in [2].

To summarize, the inherent complexity of the renaming problem and the methods used in obtaining the algorithms are basic and are useful both in their own right and in attacking other problems in asynchronous message passing environment. Throughout the rest of the paper, we concentrate only on the initial stage of selecting the new names and deciding them, and ignore the issue of possible applications or the question of how old names are dropped and replaced by new ones.

Note that the original names of the participating processors are not known to all processors in advance. Otherwise, there exists a trivial solution: Assuming the old names are $p_1 < p_2 < \cdots < p_n$, let p_i choose i as its new name. This meets both requirements with $N = n$. We further assume that once a processor *decides* on a name, it will not change it. Otherwise, one can again present a trivial solution: Each processor selects its ordinal number among the original names known to it as a new name. Later, the processor changes its choice in the appropriate way whenever it learns of an original name previously unknown to it. To rule out the trivial solution of hardcoding new processors' names into the algorithm, we assume that the algorithm executed by a processor depends only on its original name.

For the uniqueness problem, our algorithm yields new names from a space of size $n + t$. A lower bound of $n + 1$ is proven. The proof generalizes to a class of coordination problems. For the order-preserving problem, we get a new name space of size $2^t(n - t + 1) - 1$, with a tightly matching lower bound.

We remark that a simple *probabilistic* version of our algorithm yields names from a space of size n, which is clearly optimal (we omit any further description or analysis since this is done by standard methods). The gap between our bounds for deterministic algorithms is still an open question. To narrow this gap, a better understanding of the effect of multiple faults is required.

The rest of the paper is organized as follows. In Section 2, we give a formal description of the model. The lower bounds for the uniqueness problem, for the order-preserving problem, and for the relation between n and t appear in Sections 3, 4, and 5, respectively. A simple but inefficient algorithm for the uniqueness problem is described in Section 6, and serves to demonstrate some of the ideas behind our later algorithms. The two algorithms for the uniqueness problem and the order-preserving problem are described in Sections 7 and 8, respectively.

2. *The Computation Model*

The definitions follow the ones introduced in [6] and [8] with slight modifications. For convenience of presentation we number the processors $p_1, \ldots, p_n$. To avoid trivial solutions to the renaming problem, this numbering is external and is unknown to the processors themselves. The names by which the processors identify themselves are considered to be part of their inputs. (We sometimes, especially in the upper bounds, ignore this distinction and refer by p_i both to the ith processor and to its original name.)

Let I be a set of input values, and D a set of decision (output) values. Assume that processors have special *input* and *decision* registers, and that decision registers are *write-once* and are initialized with a "$*$" (meaning that a decision has not yet been made). An *input vector* $\vec{I} = \langle i_1, \ldots, i_n \rangle \in I^n$ is an assignment of some input values $i_1, \ldots, i_n \in I$ to the input registers of the processors $p_1, \ldots, p_n$, respectively. A *decision vector* $\vec{D} = \langle d_1, \ldots, d_n \rangle$ is the concatenation of the values $d_1, \ldots, d_n \in D \cup \{*\}$ stored in the decision registers of the processors $p_1, \ldots, p_n$, respectively. A decision vector $\vec{D}_2$ *extends* another decision vector $\vec{D}_1$ if some coordinates that contain "$*$" in $\vec{D}_1$ contain values from D in $\vec{D}_2$.

A *coordination problem* Φ is a mapping from I^n to subsets of D^n, defining the permitted decisions for any input vector. If a problem maps some input vector to the empty set, then this is not a *legal* input vector for that problem. Denote by $\mathcal{I}$ the set of legal input vectors, that is, $\vec{I} \in I^n$ is in $\mathcal{I}$ if and only if $\Phi(\vec{I}) \neq \emptyset$. In the renaming problem, for example, I is the set of natural numbers, while $D = \{1, \ldots, N\}$ for some bound N, and $\mathcal{I}$, the set of legal input vectors, includes those vectors in I^n in which all numbers are distinct. Denote by $\bar{\mathcal{D}}$ the set of all vectors in D^n in which no decision value appears twice. The uniqueness problem maps any legal input vector to the set $\bar{\mathcal{D}}$. Similarly, the order-preserving problem maps any legal input vector $\vec{I}$ to the set of vectors in $\bar{\mathcal{D}}$ that have the same order pattern as $\vec{I}$. (In both cases, all illegal input vectors are mapped to the empty set.)

An *algorithm* is a system of n ($n \geq 2$) processors $P = \{p_1, \ldots, p_n\}$, modeled as infinite-state machines with state set Z. To denote the initial value of the input register, we assume that the set of input values I is mapped on the set of initial states in Z. Each processor starts in an initial state, and follows a deterministic algorithm involving receiving and sending messages. The messages are drawn from an infinite set M. Each processor has a *message buffer*, holding the messages that have been sent to it but not yet received. Each buffer is modeled as an unordered set of messages, and is assumed to be initially empty. A message is sent to a processor by adding it to its message buffer.

A configuration consists of the state of every processor and the contents of every message buffer. In an initial configuration, every processor is in an initial state, and every message buffer is empty. An *event* is a pair (p, m) where $p \in P$ and $m \in M \cup \{\perp\}$, where $\perp$ denotes "no message." An event (p, m) is *applicable* to a configuration C if p's message buffer contains the message m. Applying the event $e = (p, m)$ means that p receives the message by m (by removing it from its message buffer), carries out local computations, changes its state and sends messages (as specified by the algorithm) in a single atomic step. In particular, the event $(p, \perp)$ in which p is active but receives no message is always applicable. (This corresponds to the situation in which all messages sent to p, that is, placed in p's buffer, are still "in transit" and have not arrived yet.)

A *run* of the algorithm is finite (possibly empty) or infinite sequence of events. The *underlying schedule* of a run R is the sequence of processors of the events in R. The run R is *applicable* to a configuration C if the events in it can be applied in turn starting from C. If R is finite, the resulting configuration is denoted (C, R) and is said to be *reachable* from C. A configuration is *legal* if it is reachable from an initial configuration. (In particular, any initial configuration is legal.) A configuration *has decision value* $\vec{D}$ if the decision registers of the processors form the decision vector $\vec{D}$. We say that a decision vector $\vec{D}$ is *reachable* from a configuration C if there exists a configuration C' reachable from C with decision value $\vec{D}$.

There are no restrictions on the order in which processors take steps or the order in which messages are received, except that each nonfaulty processor takes an infinite number of steps during any infinite run, and every message sent to a nonfaulty processor should eventually be delivered. More formally, for $0 \leq t \leq n - 1$, an infinite run R is said to be *t-admissible* if it is applicable to a legal configuration $C = (\vec{I}, T)$, where $\vec{I}$ is an initial configuration and T a run, and if there is a set P' of size at least $n - t$ with the property that every processor in P' appears infinitely often in the underlying schedule of R, and receives all messages sent to it in $T \cdot R$. The maximal such P' is denoted $NF(R)$, and it is the set of *nonfaulty* processors in R.

An algorithm is a *t-resilient* algorithm for solving a coordination problem Φ if for any legal input vector $\vec{I}$, and for any t-admissible run R from $\vec{I}$, there exists a finite prefix R' of R in which all processors in $NF(R)$ decide, and moreover, for every finite prefix R'' of R, the decision value of $(\vec{I}, R'')$ can be extended to a vector in $\Phi(\vec{I})$. Notice that since processors fail only by stopping, and since processors cannot distinguish a failed processor from a very slow processor, we cannot "declare" a processor that did not participate in a finite run to be faulty. Hence, the decisions made by any of the processors should be extendible by decisions of the slow processors.

When proving the lower bounds it is convenient to model the situation by an "adversary" that has the power to control the order of execution of processors' steps, to manipulate the arrival of messages and to distribute the input names. The notion of a run incorporates these aspects. However, for clarity of presentation we sometimes prefer to divide the adversarial powers among three "players"—the "scheduler," the "postmaster" and the "distributor." The *scheduler* determines the underlying schedule of the run, that is, decides which processor will be active in each step. The *postmaster* controls the policy of message delivery, that is, decides which of the messages currently in the active processor's buffer (if any) is to be received at that step. The *distributor* creates the input vector. The adversaries may coordinate their decisions, which are done dynamically. In these terms, an algorithm solves a given problem in a completely asynchronous system if it correctly solves the problem for arbitrary scheduler, postmaster and distributor that generate a t-admissible run.

Our lower-bound proofs use mainly post policies of the following structure. For some set $P' \subseteq P$, $|P'| \geq n - t$, the postmaster delivers all messages sent among processors in P' whenever possible (subject to the scheduler policy) and in the right order, while suspending all messages sent to and from processors in $P - P'$. We refer to this policy as an *immediate post policy for P'*. (In proving our lower bounds, we sometimes apply such a policy to set P' that is only a *subset* of the nonfaulty processors, but in each such case we may apply the policy only for a finite number of steps, since using it forever constitutes a violation of the "fairness" rule requiring all correct processors to eventually receive their mail.)

To reflect the fact that processors' numbering is external, that is, that the renaming protocol can depend only on processors' initial names, we make the following assumption on the algorithms we consider:

Anonymity Assumption. Let π be a permutation of $\{1, \ldots, n\}$, let J and J' be two initial configurations, such that for any $1 \leq i \leq n$ processor p_i in J and processor $p_{\pi(i)}$ in J' are in the same state. Let R be a run applicable to J and let $\pi(R)$ be the run with the permutation π applied to the underlying schedule of R, then p_i in (J, R) and $p_{\pi(i)}$ in $(J', \pi(R))$ are in the same state.

3. *The $n + 1$ Lower Bound*

The lemmas given in this section all concern t-resilient algorithms and their runs and describe some of their basic properties. Throughout this section, we assume that $t \geq 1$ and $n \geq 2$. The first lemma implies some *commutativity* of schedules, and was proved in [8]. From the deterministic nature of the algorithm and the asynchronous nature of the system it follows that if the events in two finite runs do not "interact" (in a sense to be made precise by the lemma), then the order in which they are applied does not matter.

LEMMA 3.1 [8]. *Let C be a legal configuration and let R_1 and R_2 be runs applicable to C such that the set of processors occurring in the run R_1 is disjoint from the set of processors in R_2. Let $C_1 = (C, R_1)$ and $C_2 = (C, R_2)$, then R_1 is applicable to C_2, R_2 is applicable to C_1, and $(C_1, R_2) = (C_2, R_1)$.*

The next simple lemma is the key to our proofs. Let $p \in P$, two legal configurations C_1, C_2 are called *p-apart* if there exist two legal configurations C_1', C_2' that differ in at most p's state, and two runs R_1, R_2 in which only p takes steps, such that $C_j = (C_j', R_j)$, for $j = 1, 2$.

LEMMA 3.2. *Let C_1 and C_2 be any two legal configurations that are p_i-apart. Then there are decision values $\vec{D}_1$ and $\vec{D}_2$ reachable from C_1 and C_2, respectively, that differ in at most the ith coordinate.*

PROOF. By definition $C_j = (C_j', R_j)$, for $j = 1, 2$, C_1', C_2' differ in at most p_i's state, and only p_i take steps in R_j, $j = 1, 2$. Let R be some infinite run in which p_i

takes no steps, with an immediate post policy for $P - \{p_i\}$. Apply this run to C_1 and to C_2. Since $R_1 \cdot R$ is a 1-admissible run of a t-resilient algorithm, and by hypothesis $t \geq 1$, there is a finite prefix R' of R in which all processors in $NF(R) = P - \{p_i\}$ decide in (C_1, R'). Since no messages from p_i arrive during this run, the states of all processors in $NF(R)$ are the same in (C_1, R'') and (C_2, R''), for any finite prefix R'' of R. In particular, the states of all processors in $NF(R)$ are the same in (C_1, R') and (C_2, R'); hence, all processors in $NF(R)$ decide in (C_2, R').

Now consider two t-admissible runs T_1 and T_2 of the algorithm from (C_1, R') and from (C_2, R'), respectively, with the property that in T_j, $j = 1, 2$, all processors (including p_i) take steps infinitely often and all messages sent are received. Since the algorithm is t-resilient, in both runs p_i decides. Denote the decision vectors thus reached by $\vec{D}_1$ and $\vec{D}_2$. Since processors in $P - \{p_i\}$ cannot change their decision, and have the same decisions in both configurations (C_1, R') and (C_2, R'), $\vec{D}_1$ and $\vec{D}_2$ differ in at most the ith coordinate. $\square$

This lemma can be generalized by replacing p_i with any set $F \subseteq P$, $|F| \leq t$ (and the proof is similar).

Two vectors $\vec{v}$ and $\vec{w}$ (of equal length) are called *adjacent* if they differ in at most one coordinate. Given a set $\mathscr{E}$ of vectors, this adjacency relation induces a graph structure whose vertices are the vectors in $\mathscr{E}$, with an edge between two vertices iff the corresponding vectors are adjacent. Denote this graph by $\mathscr{G}(\mathscr{E})$.

Look at some configuration C, and let $\mathscr{D}(C)$ be the set of decision vectors reachable from C (by a certain algorithm). A legal configuration C is called *indecisive* (for a particular algorithm) if the graph $\mathscr{G}(\mathscr{D}(\mathscr{E}))$ is disconnected (i.e., contains more than one connected component). The following is a generalization of Lemma 3 from [8].

LEMMA 3.3. *Let C be a legal indecisive configuration of a t-resilient algorithm, and let $e = (p, m)$ be an event applicable to C. Then there is some finite run R such that $(C, R \cdot e)$ is legal and indecisive.*

PROOF. Look at all the configurations reachable from C by finite runs that do not include e. Since messages can be delayed, e can be applied to all of these configurations.

Assume, by way of contradiction, that all legal configurations of the form $(C, R \cdot e)$ are decisive (i.e., all the decision vectors reachable from them are connected). Since C is indecisive, there are decision vectors in at least two different connected components that are reachable from C. Without loss of generality, there are configurations E_1 and E_2 reachable from C such that (E_i, e) is legal and has decision vectors in the ith connected component. In particular, the decision vectors reachable from E_1 are disjoint from the decision vectors reachable from E_2.

Using standard techniques (as in the proof of Lemma 3 in [8]), it can be shown that there are two configurations E_1 and E_2, reachable from C, with the above properties, and such that one is obtained in a single step from the other, that is, $E_2 = (E_1, e')$, where $e' = (p', m')$.

Case 1, $p \neq p'$. By Lemma 3.1, $(E_2, e) = ((E_1, e'), e) = (E_1, e' \cdot e) = (E_1, e \cdot e')$, which contradicts the assumption that decision vectors from separate connected components are reached from each configuration.

Case 2, $p = p'$. Then (E_1, e) and $(E_2, e) = (E_1, e' \cdot e)$ are p-apart. By Lemma 3.2, there are decision vectors reachable from both, that are adjacent (differ in p's decision only). Since neither (E_1, e) nor (E_2, e) is indecisive, all the decision vectors reachable from both are connected, that is, belong to the same connected component. This contradicts the way E_1 and E_2 were selected.) □

Recall that the set $\mathscr{D}(\vec{I})$ is the set of the decision values that are reachable by a particular algorithm when starting from an initial legal configuration $\vec{I}$, and denote by $\mathscr{D}(\mathscr{I})$ the union of $\mathscr{D}(\vec{I})$ over $\vec{I} \in \mathscr{I}$. An algorithm is *splitting* if there exist two legal inputs $\vec{I}_1, \vec{I}_2$ that are connected in $\mathscr{G}(\mathscr{I})$, and two decision vectors $D_i \in \mathscr{D}(\vec{I}_i)$, such that D_1 is not connected with D_2 in $\mathscr{G}(\mathscr{D}(\mathscr{I}))$, that is, the algorithm maps connected initial configurations to disconnected decision vectors. Using the techniques of [8], we have the following lemma.

LEMMA 3.4. *For any* 1*-resilient splitting algorithm that solves a coordination problem, there exists an initial legal indecisive configuration.*

PROOF. Assume to the contrary that for any initial legal configuration $\vec{I}$, $\mathscr{G}(\mathscr{D}(\vec{I}))$ is connected. By the splitting property there exist two initial configurations $\vec{I}$ and $\vec{J}$ such that $\mathscr{D}(\vec{I})$ and $\mathscr{D}(\vec{J})$ contain elements from two different connected components. By induction on the distance between $\mathscr{I}$ and $\mathscr{J}$ in $\mathscr{G}(\mathscr{I})$, it can be shown that there are two *adjacent* initial configurations $\vec{I}_1$ and $\vec{I}_2$ with this property. Let p be the processor in which $\vec{I}_1$ and $\vec{I}_2$ differ. $\vec{I}_1$ and $\vec{I}_2$ are p-apart, and hence, by Lemma 3.2, there exist two *adjacent* decision vectors, $\vec{D}_1$ and $\vec{D}_2$ reachable from $\vec{I}_1$ and $\vec{I}_2$ (respectively). Since $\mathscr{G}(\mathscr{D}(\vec{I}_1))$ is connected $\vec{D}_1$ is in this connected component, and in the same way $\vec{D}_2$ is connected to all vectors in $\mathscr{D}(\vec{I}_2)$. However, $\vec{D}_1$ and $\vec{D}_2$ are adjacent, hence all vectors in $\mathscr{D}(\vec{I}_1)$ are connected to all vectors in $\mathscr{D}(\vec{I}_2)$ (in $\mathscr{G}(\mathscr{D}(\mathscr{I}))$); a contradiction. □

LEMMA 3.5. *There is no* 1*-resilient splitting algorithm for solving a coordination problem in a completely asynchronous system.*

PROOF. The proof follows the technique introduced in [8] (and appearing in Section 3 of [6]). Using Lemma 3.3 and 3.4, we construct an infinite 0-admissible run in which all configurations are indecisive, as follows: Let B_1 be an initial indecisive configuration, whose existence is guaranteed by Lemma 3.4. Usually, if B_j is indecisive, let $p = p_i$, where $i = (j \bmod n)$, and let m be the oldest message in p's buffer (not yet delivered). Apply Lemma 3.3 to B_j and $e = (p, m)$, and denote by B_{j+1} the resulting indecisive configuration. The resulting run is 0-admissible and contains only indecisive configurations.

In a 0-admissible run, all processors should eventually decide. This means that there is a configuration reachable after some finite prefix of the run that has only *one* possible decision vector. This implies that following configurations cannot be indecisive. A contradiction. □

Similar results, for the case of distinguishable processors, were proved by Moran and Wolfstahl [10] and by Taubenfeld [13]. Note that in our model distinct processors could be modeled by appending i to processor p_i's input.

In the renaming problem with $N = n$, there are no connected decision vectors. This implies that for proving that any algorithm is splitting it suffices to show it must reach at least two decision vectors. This is done in the proof of the next lemma.

LEMMA 3.6. *Any algorithm that solves the renaming problem with $N = n$ is a splitting algorithm.*

PROOF. Let $\mathscr{A}$ be some renaming algorithm and assume to the contrary that in all runs of $\mathscr{A}$ only one decision vector is reached. Therefore, each processor always decides the same value. Assume p_i always decides x_i, in all possible runs. By the anonymity assumption, we can permute the inputs between p_i and p_j, for any j, $1 \leq j \leq n$, which implies that $x_i = x_j$. It follows that all x_is are identical, thus violating the uniqueness requirement. □

THEOREM 3.7. *There is no* 1*-resilient algorithm for solving the uniqueness problem with* $N = n$ *in a completely asynchronous system.*

The remainder of this section deals with coordination problems in general and is not essential to the results in the rest of the paper.

As before, $\Phi(\vec{I})$ is the set of all decision vectors that are possible when starting from an initial configuration $\vec{I}$. Notice that an algorithm for solving Φ need not reach *all* the decision vectors in $\Phi(\vec{I})$. However, the set of decision vectors reachable in all runs of a particular algorithm solving a problem should *cover* Φ in the following sense: A set $\mathscr{C} \subseteq D^n$ *covers* a coordination problem Φ if for any $\vec{I} \in \mathscr{I}$, $\Phi(\vec{I}) \cap \mathscr{C} \neq \varnothing$. A cover $\mathscr{C}$ is a *minimal cover* for Φ if no proper subset of it covers Φ. Note that the set of decision vectors reachable in all runs of an algorithm solving Φ must include a minimal cover for Φ.

Theorem 3.7 can be proved for a class of nontrivial coordination problems. A coordination problem Φ is *nontrivial* if no minimal cover of it is a singleton, that is, every algorithm for solving Φ must reach at least two decision vectors. A coordination problem Φ is called *splitting* if $\mathscr{G}(\mathscr{C})$ is disconnected for every minimal cover $\mathscr{C}$ of Φ, every algorithm for solving Φ must reach decision vectors in at least two connected components. The details of the proofs below are omitted, because of being almost identical to those above.

LEMMA 3.8. *Any algorithm for solving a splitting coordination problem is splitting.*

LEMMA 3.9. *There is no* 1*-resilient algorithm for solving a splitting coordination problem in a completely asynchronous system.*

A problem is called a *complete coordination* problem if the decisions of any $n - 1$ processors define *uniquely* the decision of the nth processor. Thus, we have the following simple observation.

Observation. Let Φ be a complete coordination problem and let $\vec{D}_1, \vec{D}_2 \in \Phi(\vec{I})$ such that $\vec{D}_1 \neq \vec{D}_2$. Then they differ in at least two coordinates.

It follows that no two (different) decision vectors are adjacent. This immediately implies that any *nontrivial* complete coordination problem is splitting, and thus by Lemma 3.9.

COROLLARY 3.10. *There is no* 1*-resilient algorithm for solving a nontrivial complete coordination problem in a completely asynchronous system.*

4. *The Order-Preserving Lower Bound*

In this section we prove that every t-resilient algorithm for solving the order-preserving problem requires $N \geq 2^t(n - t + 1) - 1$. The proof goes through even with a mild form of asynchrony: processors may operate in lock-step synchrony, and messages are delivered immediately, but processors do not start synchronously. Intuitively, we exploit this behavior by looking at runs that activate p_i

(for $n - t + 1 \leq i \leq n$) only after $p_1, \ldots, p_{i-1}$ have already decided. This policy forces these "early waking" processors to keep enough "space" between their new names because the processors that will wake up later might have names that fall between any two of them.

For convenience, we assume in this proof that the original name space is unbounded; however, the lower bound presented holds even if the name space is bounded, provided that its size is sufficiently large.

For any schedule σ, denote by σ^j the concatenation of j copies of σ (for some integer i).

For the sake of this lower bound, we assume the immediate post policy in all runs.

More formally, for any k, $0 \leq k \leq t$, define $\mathcal{S}_k$ to be the collection of all schedules σ with the properties that only the processors $p_1, \ldots, p_{n-k}$ appear in σ and each of these processors appears infinitely often. Define $\mathcal{I}_k$ to be all the input vectors $(a_1, \ldots, a_n)$ where for every i and j, $1 \leq i < j \leq n - k$, $2^k < |a_i - a_j|$ and $2^k < a_i$.

LEMMA 4.1. *Let A be an algorithm for the order-preserving problem with the new name space* $1, \ldots, N$. *For any* $0 \leq k \leq t$, *consider any* $\sigma \in \mathcal{S}_k$ *and* $I \in \mathcal{I}_k$ *and suppose that* $x_1, \ldots, x_{n-k}$ *are the new names chosen by* $p_1, \ldots, p_{n-k}$, *respectively, while running A with σ on I. Then* $|x_i - x_j| > 2^k - 1$ *for every i and j*, $1 \leq i \neq j \leq n - k$. *Also* $x_i > 2^k - 1$ *and* $(N + 1) - x_i > 2^k - 1$ *for every* $1 \leq i \leq n - k$.

PROOF. By induction on k. For $k = 0$ the claim follows from the requirement that processors choose distinct names and since $2^k - 1 = 0$.

For the inductive step, assume the hypothesis is true for $0 \leq k < t$, and assume to the contrary that the claim does not hold for $k + 1$. To be more specific, assume the existence of a schedule $\sigma \in \mathcal{S}_{k+1}$ and an input vector $I = (a_1, \ldots, a_n) \in \mathcal{I}_{k+1}$ and a run R with σ on I. For the new names $x_1, \ldots, x_{n-(k+1)}$ of the processors $p_1, \ldots, p_{n-(k+1)}$, respectively, obtained in this run, one of the following three cases happens:

(1) There exist i and j, $1 \leq i \neq j \leq n - (k + 1)$, such that $|x_i - x_j| \leq 2^{k+1} - 1$, or
(2) there exists i, $1 \leq i \leq n - (k + 1)$, such that $x_i \leq 2^{k+1} - 1$, or
(3) there exists i, $1 \leq i \leq n - (k + 1)$, such that $(N + 1) - x_i \leq 2^{k+1} - 1$.

By the definition of a t-resilient algorithm, there exists a prefix σ' of σ such that after applying σ' the processors $p_1, \ldots, p_{n-(k+1)}$ decide on the new names $x_1, \ldots, x_{n-(k+1)}$, respectively.

Suppose now that the first case, out of the three mentioned above, takes place (the other two cases are similar). Moreover, assume that i and j are the indices so $|x_i - x_j|$ is minimal and $x_j \leq x_i$. Let $\sigma'' = \sigma' \cdot (p_1, \ldots, p_{n-k})^\infty$ and let $I'' = (a_1, \ldots, a_{n-(k+1)}, \lceil (a_i - a_j)/2 \rceil, \ldots, a_n)$. Clearly, $\sigma'' \in \mathcal{S}_k$. The minimality of $|x_i - x_j|$ implies that there is no a_m, $1 \leq m \leq n - (k + 1)$, such that $a_i < a_m < a_j$; therefore, $a_{n-k} = \lceil (a_i + a_j)/2 \rceil$ is in the right distance from all the other. Thus, also $I'' \in \mathcal{I}_k$.

Run algorithm A with the schedule σ'' on I'' and let x_{n-k} be the name chosen by p_{n-k} in this run. By the order-preserving condition, $x_j < x_{n-k} < x_i$. Since $x_i - x_j \leq 2^{k+1} - 1$, either $x_{n-k} - x_j \leq 2^k - 1$ or $x_i - x_{n-k} \leq 2^k - 1$, contradicting the inductive hypothesis. □

Remark. We cannot prove Lemma 4.1 on any input vector, since if p_i and p_j have original names a_i and a_j, that are close to each other they do not have to leave a large gap between the new names they choose.

THEOREM 4.2. *Every t-resilient algorithm for solving the order-preserving problem requires $N \geq 2^t(n - t + 1) - 1$.*

PROOF. Apply Lemma 4.1 with $k = t$ and assume without loss of generality that $a_1 < a_2 < \cdots < a_{n-t}$. Then

$$N + 1 = (N + 1 - x_{n-t}) + \sum_{i=2}^{n-t} (x_i - x_{i-1}) + x_1 \geq 2^t(n - t + 1),$$

hence,

$$N \geq 2^t(n - t + 1) - 1. \qquad \square$$

5. *The $n > 2t$ Lower Bound*

In this section, we prove two impossibility results. First, we show that there is no solution for either of the renaming problems if $n \leq 2t$. Then, we show that there is no solution to these problems when $2 \leq t$ if every processor stops after choosing a new name.

THEOREM 5.1. *For any N, $N \geq n + 1$, if $n \leq 2t$, then there is no t-resilient algorithm for solving either of the renaming problems with a new name space of size N in a completely asynchronous system.*

PROOF. Assume to the contrary that $n \leq 2t$ and there is an algorithm A that solves any of the renaming problems using some new name space of size N. We prove the claim by demonstrating the existence of some input vector that fails the algorithm. Without loss of generality, assume that $n = 2t$.

Run algorithm A with the schedule $\sigma = (p_1, \ldots, p_t)^\infty$, with any post policy, and on all possible t-tuples of original names for these processors. Since $n = 2t$, the definition of a t-resilient algorithm implies that for each such input I there is some finite prefix σ_I of σ in which the t operating processors choose distinct names. Since the original name space is unbounded and N is fixed, there exist two disjoint sets of processors' names, say $\{a_1, \ldots, a_t\}$ and $\{b_1, \ldots, b_t\}$, such that p_1 decides the same new name when running A with σ on the inputs $I_a = (a_1, \ldots, a_t, \ldots)$ and $I_b = (b_1, \ldots, b_t, \ldots)$. Let σ_a (respectively, σ_b) be the finite prefix of σ in which processors with originals names I_a (respectively, I_b) decide, and let σ_b' be σ_b where p_i is replaced by p_{t+j} for $1 \leq i \leq t$.

Now run the algorithm on the input vector $(a_1, \ldots, a_t, b_1, \ldots, b_t)$ with the schedule $\sigma' = \sigma_a \cdot \sigma_b'$ and a post policy by which all messages sent between the two groups of processors are delayed until the decision of all the processors. By the anonymity assumption, for processor p_1 (respectively, p_{t+1}) the schedule σ_a (respectively, σ_b') is indistinguishable from the subschedule $\sigma = \sigma_a \cdot \sigma_b'$. Since the post policy is legal they both choose the same name; a contradiction. $\square$

THEOREM 5.2. *For any $N \geq n + 1$ and for any $t \geq 2$, if every processor is required to stop running after deciding its new name, then there is no t-resilient algorithm for solving either of the renaming problems with a new name space of size N in a completely asynchronous system.*

PROOF. As in the previous proof we assume, toward a contradiction, that there exists an algorithm A solving the problem under the assumption of the theorem and demonstrate the existence of some input vector that fails the algorithm.

Run algorithm A with the schedule $\sigma = (p_1, \ldots, p_{n-t})^\infty$, with any post policy and on the input vector $(a_1, \ldots, a_n)$ for arbitrary a_i, $1 \leq i \leq n$. The definition of a t-resilient algorithm implies that there is some finite prefix σ' of σ in which the $n - t$ operating processors choose distinct names. Let $\sigma_q = \sigma' \cdot (q)^\infty$. Since the original name space is unbounded and N is fixed, there exist two processor names a and a' so q decides the same new name when running algorithm A on the inputs $(a_1, \ldots, a_{n-t}, a, \ldots, a_n)$ and $(a_1, \ldots, a_{n-t}, a', \ldots, a_n)$ with the schedule σ_q and with any post policy (actually, q is p_{n-t+1}).

Now run algorithm A with the schedule $\sigma_{qq'} = \sigma' \cdot (q, q')^\infty$, with the post policy that delays all messages sent between q and q', and on the input vector $(a_1, \ldots, a_{n-t}, a, a', \ldots, a_n)$ (here, q and q' are p_{n-t+1} and p_{n-t+2}). At the end of the subschedule σ' of $\sigma_{qq'}$ all the processors $p_1, \ldots, p_{n-t}$ stop running before knowing about q and q'. Therefore, by the anonymity assumption, for processor q (respectively, q') the schedule σ_q (respectively, $\sigma_{q'}$) is indistinguishable from the schedule $\sigma_{qq'}$. Hence, they both choose the same name; a contradiction. □

Hence, in our algorithms we assume that $2t + 1 \leq n$ and that every processor continues to cooperate after it chooses its name, to help the others.

6. *A Simple Uniqueness Algorithm*

In this section we present a simple algorithm for the uniqueness problem, which requires a name space of size $N = (n - t/2)(t + 1)$. A later algorithm improves this bound to $n + t$. Still, this algorithm illustrates some of the ideas used in the later algorithms.

Throughout the algorithm, each processor p attempts to learn of as many initial names as possible, keeping the initial names it *does* know of in a vector (i.e., an ordered set) V. As a rule, sets of (old or new) names are always taken to be ordered in increasing order, and the *rank* of a name in a set of names refers to its position in this order (with the smallest having rank 1). The rank of a processor in the vector V is defined to be the rank of its name. These ranks play an important role in all later algorithms.

Initially, $V = \{p\}$. Thereafter, processors continually exchange their sets, each processor updating its own set as it learns of new initial names. Each processor also maintains a counter c that is the number of processors that have claimed having the same set V as itself.

Since t processors might be faulty, a processor cannot expect to get more than $n - t$ messages from different processors (including itself). Thus, after having $c = n - t$ identical messages (or sets V), it makes no sense to wait for more messages (which might never arrive), and the processor should take some action. This observation generalizes to every data structure the processors might try to "stablize" by exchanging messages: one cannot expect to get more than $n - t$ identical copies, since at this stage there may exist a correct schedule in which no processor can add any information to the others.

The above remarks lead us to define the basic notion of a *stable* vector. (Here and in the sequel, we sometimes abuse terminology by using the term *vector* and the notation V to refer to some specific *value* or content of the vector.)

Definition 6.1. Consider an algorithm A in which processors exchange (the contents of) a vector V of a certain type. The vector V is *stable with respect to a processor p* in a given run of A if p has received $n - t - 1$ messages containing identical copies of V. The vector V is *stable* if it is stable with respect to some processor p.

Here and in later algorithms, we need a (partial) ordering on the vectors V, reflecting the accumulation of knowledge in the processors. Intuitively, $V > V'$ means that V is more updated than V'. For this section, we simply take as our partial order the containment relation on sets V (i.e, $V > V'$ iff $V \supset V'$).

Definition 6.2. Consider an algorithm A in which processors exchange a vector V. We say that A is *locally proper* (with respect to the vector ordering) if, for every run of A and for every processor p, the set of values of the vector V held by p during the run is totally ordered.

In fact, in all cases considered here, the ordering on the vectors held by a processor during a run corresponds directly to their order in time, that is, the value of the vector V held by a processor at some point in time is no smaller than any value of V held earlier by this processor.

LEMMA 6.1. *For every algorithm A, if A is locally proper with respect to the vector ordering, then in any run of A the set of stable vectors is totally ordered.*

PROOF. Assuming V and V' are two incomparable stable vectors obtained in the same run. By definition V is stable with respect to some processor, so this processor received copies of V from at least $n - t$ distinct processors. the same holds for V'. Since $n \geq 2t + 1$, there is some processor p that sent both V and V'. Hence, p held both values in its vector at different stages of the run, which contradicts the assumption that A is locally proper. □

We now give a formal description of the algorithm.

```
The Simple Algorithm S   /* for a processor $p$ */
 0. $V \leftarrow \{p\}$.
 1. /* sending a new set */
    a. Send $V$ to every other processor.
    b. $c \leftarrow 1$.
 2. Wait until you receive a message $V'$.
    a. If $V' \subset V$ then   /* $V'$ contains old information */
                 goto 2.
    b. If $V' - V \neq \emptyset$ then:  /* $V'$ contains new information, update $V$ and restart */
                 $V \leftarrow V \cup V'$.
                 Goto 1.
    c. If $V = V'$ then:  /* an identical copy */
                 $c \leftarrow c + 1$.
                 If $c < n - t$ then goto 2 else goto 3.
 3. /* $V$ is a stable vector—decide */
    a. $v \leftarrow |V|$.
    b. Let $r$ be your rank in the vector $V$.
    c. Choose your new name to be the pair $\langle v, r \rangle$.
 4. /* echo stage – helping other processors */
    Continue forever; whenever you receive a message $V'$:
    a. $V \leftarrow V \cup V'$.
    b. Send $V$ to every other processor.
```

To prove the correctness of the algorithm we first make the following direct observation.

LEMMA 6.2. *The algorithm S is locally proper with respect to the set ordering.*

COROLLARY 6.3. *In any run of the algorithm S, the collection of stable vectors V is totally ordered.*

PROOF. Immediate from Lemmas 6.1 and 6.2. □

COROLLARY 6.4. *In any run of the algorithm S, if V and V' are stable sets of the same size, then $V = V'$.*

PROOF. If $V \neq V'$, then V and V' are not comparable, contradicting Corollary 6.3. □

LEMMA 6.5. *In every run of the algorithm S, each correct processor p eventually obtains a stable set V.*

PROOF. Given a nonfaulty processor p, there is a time τ in which p sets its set to V and never changes it again (since p can update its set at most $n - 1$ times). This time τ has the property that

(1) p does not receive the set V before (or p would have set its set to V sooner), and
(2) any other set V' that p receives throughout the run satisfies $V' \subseteq V$ (since V is the final value of p's set).

At this point, p sends the set V to all processors. For each nonfaulty processor p', let V' be the value of the set of p' when p' receives V from p. Processor p' has sent its set to p on the first time it set it to V'. Therefore, by property (2), $V' \subseteq V$. If $V' = V$, then by property (1), V' arrives after time τ. If $V' \subset V$, then on receiving V from p, p' sets its set to V and sends it to p. In both cases, p eventually gets a supporting copy for V from p' after time τ.

So after p sets its set to V, it receives $n - t$ copies of V and leaves the loop at Step 2. □

LEMMA 6.6. *In every run of the algorithm S, the names chosen by the processors are distinct.*

PROOF. Assume that processors p and q choose the names $\langle v_p, r_p \rangle$ and $\langle v_q, r_q \rangle$, respectively. If $v_p \neq v_q$, then the claim is immediate. Otherwise, the stable sets held by p and q at the moment of decision were of the same size, and by Corollary 6.4 they are identical. Since both p and q are in that set and $p \neq q$, necessarily $r_p \neq r_q$. □

LEMMA 6.7. *The number of different pairs $\langle v, r \rangle$, over all possible runs of the algorithm S is $N = (n - t/2)(t + 1)$.*

PROOF. The first component, v, satisfies, $n - t \leq v \leq n$. For every possible v, the second component may assume exactly v different values. This is because, in Step 3 of the algorithm, a deciding processor p is always in the list V. Therefore, the total number of possible pairs is

$$N = \sum_{v=n-t}^{n} v = \left(n - \frac{t}{2}\right)(t + 1). \qquad \square$$

Note. We can define a mapping from the pairs $\langle v, r \rangle$ to the range $[1, N]$. Using this mapping, the processors can choose their names to be integers in this range.

THEOREM 6.8. *There is an algorithm for the uniqueness problem that uses the names $\{1, \ldots, (n - t/2)(t + 1)\}$.*

7. *An Algorithm for the Uniqueness Problem*

In this section we describe a more involved algorithm for the uniqueness problem and prove its correctness.

7.1 OVERVIEW. Let us first give an outline of the algorithm. The new name space is $UNS = \{1, \dots, n + t\}$. As with the simple algorithm of Section 6, each processor p maintains and constantly updates a vector V containing information about the processors of which it knows, and a counter c of the number of processors that have claimed having the same information.

The algorithm is based on the following general strategy. A processor is required to reach a stable vector V, and then to *suggest* a name based on V. Then the processor exchanges information once more with the other processors, until it reaches a stable vector again. Now the processor has to review its suggestion, by checking whether it is currently valid. If the new information validates the suggestion, the processor now *decides* on its name. Otherwise (e.g., if the same name was suggested by some other processor simultaneously), the processor has to make a new suggestion and repeat the process.

7.2 VECTORS. The information maintained by processors during the execution of the algorithm consists of vectors V containing an entry for each known processor. Each entry contains several components, including the following:

(1) p, the old name of the processor.
(2) x^p, a new name suggested by p.
(3) J^p, a counter of the name suggestions made by p.
(4) b^p, a "decision" bit, which is 1 if p already decided on a name and 0 otherwise.

Throughout the rest of the paper we use the following notations with respect to vectors V. Denote by $UNDECIDED(V)$ the set of processors p for which $b^p = 0$ in V (i.e., the processors that have not yet decided on a name). Let $PROCS(V)$ be the set of processors (old names) in V. Let $FREE(V)$ be the set of new names from UNS that do not appear as name suggestions in V.

The number of entries in V ($= |PROCS(V)|$) is denoted by $|V|$. We use $p \in V$ as a shorthand for $p \in PROCS(V)$, and similarly $x \in V$ says that x occurs as a name suggestion in V.

The entries of each vector V are ordered internally by increasing order of the old names p (i.e., this ordering determines the rank of the entry of p in V). Initially, the vector held by p contains only one entry, corresponding to p itself, with all components except the first set to zero.

Again, the algorithm uses a partial ordering among the vectors themselves, reflecting the extent to which the vectors are up-to-date. This ordering is defined as follows:

Definition 7.1. For two vectors V, V', we say that $V \leq V'$ iff, for every processor p, $p \in V$ implies both $p \in V'$ and the value of J^p in V' is no smaller than the value of J^p in V.

Suppose that a processor holds the vector V, and gets a processor V' such that $V' \nleq V$. This means that there is some $q \in V'$ such that $q \notin V$ or J^q in V is smaller than in V'. In such a case, the processor can *update* its vector V based on V'. This operation consists of including all entries from V' that are missing in V, and replacing the x^q, J^q, and b^q components of entries in V for which V' is more recent (i.e., with larger J^q).

7.3 THE ALGORITHM. We now give a formal description of the algorithm.

Algorithm A

```
   /* For a processor $p$. Counter $c$ counts the number of identical copies of $V$ $p$ gets. */
0. Construct an initial $V$ with a single entry, for $p$, setting $x^p$, $J^p$, $b^p = 0$.
1. /* sending a new vector */
   a. Send $V$ to every other processor.
   b. $c \leftarrow 1$.
2. Wait until you receive a message $V'$.
   a. If $V' < V$ then goto 2.
   b. If $V' \not\leq V$ then:
                     /* there is some $q \in V'$ such that $q \notin V$ or $J^q$ in $V$ is smaller
                     than in $V'$ */
                  update $V$ as described above and goto 1.
   c. If $V = V'$ then:   /* received another identical copy */
                    $c \leftarrow c + 1$.
                    If $c < n - t$ then goto 2 else goto 3.
3. /* $V$ is a stable vector */
   If previously suggested a name (i.e., $x^p \neq 0$), and $x^p \neq x^q$ for any other suggested name
   $x^q \in V$, then:
   a. Decide $x^p$.
   b. $b^p \leftarrow 1$.
   c. Send $V$ to every other processor.
   d. Goto 5.
4. /* Never suggested a name before, or previous suggestion $x^p$ collides with other
   suggestion(s): Needs to suggest a new name based on the stable vector $V$ */
   a. Set $r$ to be the rank of $p$ in $UNDECIDED(V)$.
   b. If $r > t + 1$ then goto 2.   /* do not suggest any name */
   c. Suggest a name by setting $x^p \leftarrow FREE(V)(r)$, the $r$th free name in $V$.
   d. Insert $x^p$ to $V$.
   e. $J^p \leftarrow J^p + 1$.
   f. Goto 1.
5. /* echo stage—helping other processors */
   continue forever; whenever you receive a message $V'$:
   a. Update $V$ if necessary.
   b. Send $V$ to every other processor.
```

7.4 CORRECTNESS OF ALGORITHM A. We need some lemmas concerning the order relation on vectors. The first trivial observation follows directly from the description of the algorithm and Definitions 6.2 and 7.1.

LEMMA 7.1. *Algorithm A is locally proper with respect to our vector ordering.*

COROLLARY 7.2. *In any run of A, the set of stable vectors is totally ordered.*

PROOF. Immediate from Lemmas 6.1 and 7.1. □

The partial correctness of the algorithm is captured by the following claim.

LEMMA 7.3. *If p and p' have decided on new names x and x', respectively, then $x \neq x'$.*

PROOF. Assume to the contrary that $x = x'$. Let V (respectively, V') be the stable vector that p (respectively, p') held when deciding its new name. Corollary 7.2 implies that in any run of the algorithm the set of stable vectors is totally ordered; therefore, V and V' are comparable.

If $V < V'$, then the name x must appear also in V', since p never changes its suggested name after decision. Therefore, the choice of x' by p' is invalid, as it conflicts the requirement of Step 3. A similar contradiction follows from assuming $V' < V$ or $V = V'$. □

We also need to prove that no processor "gets stuck" when it has to suggest a name.

LEMMA 7.4. *Whenever a processor p has to suggest a name (in Step 4(c)), there is an available name in FREE(V).*

PROOF. We need to show that whenever a processor p gets to execute Step 4(c), $|FREE(V)| \geq t + 1$. Note that p reaches this step only when either $x^p = 0$ or x^p collides with some other suggestion in V. Therefore, the number of distinct suggestions currently appearing in V is at most $n - 1$. Given that $N \geq n - t$, there must be at least $t + 1$ free names. □

It remains to prove termination. This requires us to show that every correct processor eventually gets to decide a new name. We prove this by assuming the opposite and deriving a contradiction.

Assume the existence of a run in which some of the correct processors p continue running forever with $b^p = 0$. Let us introduce the following notation. Denote by *DECIDED* the set of all processors p that *decide* on a name along the run (i.e., that switch to $b^p = 1$ and stop increasing J^p at some point), and let *UNDECIDED* = P − *DECIDED*. Our hypothesis is that *UNDECIDED* contains at least one correct processor. Denote by *STUCK*, *STUCK* ⊆ *UNDECIDED*, the set of all processors p that get *stuck*, i.e., stop increasing J^p (but remain with $b^p = 0$) from some point on, and let *TRY* = *UNDECIDED* − *STUCK*. *TRY* contains those (correct) processors that continuously try to suggest new names forever but keep colliding, and never get to decide. See Figure 1.

LEMMA 7.5. *TRY* ≠ ∅.

PROOF. Assume to the contrary that *TRY* = ∅, or *STUCK* = *UNDECIDED*. Consider the point of time by which all processors reached their final J. The information exchanged from that point on does not change, so at some later point all correct processors obtain a stable vector. Consider the smallest correct processor in *UNDECIDED* (there is such a processor by our general hypothesis). Its rank in *UNDECIDED* is at most $t + 1$ because at most t processors are faulty. Therefore, by the rules of the algorithm it will suggest a new name and increase its J; a contradiction. □

Consequently, let p_0 be the smallest processor (old name) in *TRY*.

Since *TRY* ≠ ∅, the set of stable vectors obtained during the run is infinite. From some point on, all these vectors V satisfy the following properties:

(1) their length is $|V| = k$ for some $k \geq n - t$ (which does not change afterwards), and
(2) all processors in *DECIDED* ∪ *STUCK* have reached their final J value (hence, they do not suggest any new names afterwards, and in particular, all processors in *DECIDED* already have $b = 1$).

Let V_L be the first stable vector with the above properties (where by "first" we mean smallest according to the ordering defined earlier for vectors). Hereafter, we refer to every stable vector $V \geq V_L$ as a *limit vector*. Note that for all limit vectors V, the set of processors in it, *UNDECIDED*(V), is exactly *UNDECIDED*.

Denote the rank of p_0 in the set *UNDECIDED* by r_0. By the rules of the algorithm, for every $p \in$ *TRY*, the rank of p in the set *UNDECIDED* is at most $t + 1$. In particular,

LEMMA 7.6. $r_0 \leq t + 1$.

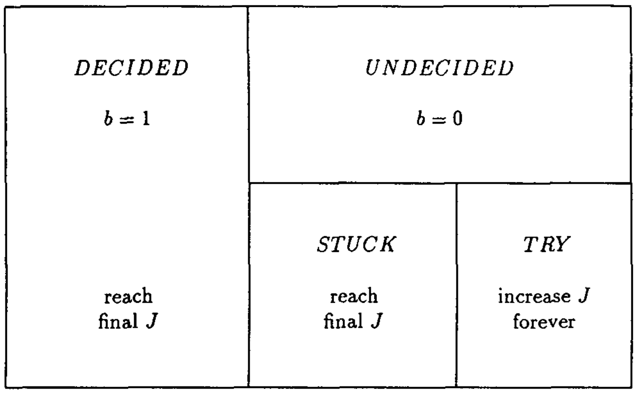

FIG. 1. The partition of processors.

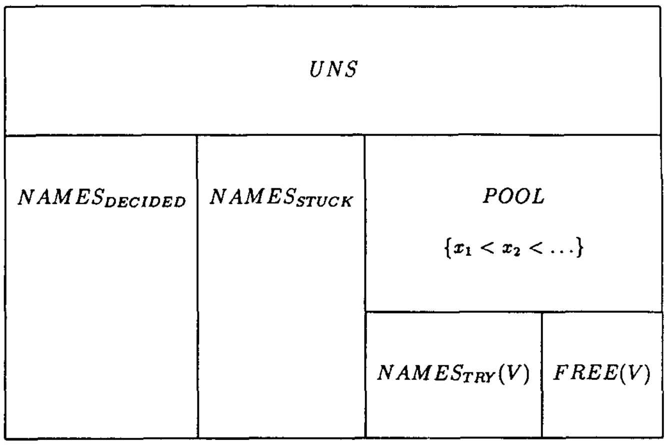

FIG. 2. The partition of new names.

Let us now classify the names in *UNS* as follows. Let $NAMES_{DECIDED}$ denote the set of new names decided by processors in *DECIDED*, let $NAMES_{STUCK}$ denote the set of new names last suggested by processors in *STUCK* and let $POOL = UNS - (NAMES_{DECIDED} \cup NAMES_{STUCK})$. (See Figure 2.) Intuitively, *POOL* is the set from which the processors of *TRY* continuously attempt to choose names. For every stable limit vector V let $NAMES_{TRY}(V)$ denote the set of new names appearing in V as suggestions x^q of processors $q \in TRY$. Note that for every limit vector V, $POOL = FREE(V) \cup NAMES_{TRY}(V)$. Assume that *POOL* is ordered, and let $POOL = \{x_1 \leq x_2 \leq \cdots\}$. For every stable limit vector V and for every name $x \in FREE(V)$, denote by $f(x, V)$ its rank in $FREE(V)$. Clearly $f(x_i, V) \leq i$.

There is a later point in time after which every processor in *TRY* has already suggested a name based on a limit vector. Hence, there is a future point in which p_0 holds a stable vector V'_L with the property that for every name x^q suggested in $NAMES_{TRY}(V'_L)$, the vector that was used to suggest x^q was a limit vector.

LEMMA 7.7. *In every stable vector $V \geq V'_L$, either $x_{r_0} \in FREE(V)$ or x_{r_0} is suggested only by p_0.*

PROOF. Assume to the contrary that x_{r_0} appears in V as a name suggestion. Since $x_{r_0} \in POOL$ (or, $x_{r_0} \notin NAMES_{DECIDED} \cup NAMES_{STUCK}$), it is suggested by some $q \in TRY$, $q \neq p_0$. Then q suggested x_{r_0} according to some stable limit vector V'. Let r be the rank of q in *UNDECIDED*. Then, $r_0 < r$ by definition of p_0. Since q must suggest the rth name in $FREE(V')$ based on V', and since x_{r_0} is *not* the rth element of $FREE(V')$ (as $f(x_{r_0}, V') \leq r_0 < r$), q could not have suggested it; a contradiction. □

Therefore, on seeing V'_L, p_0 either decides immediately on x_{r_0} as its name (if x_{r_0} appears as its suggested name in V'_L) or it suggests x_{r_0} now and decides on it upon obtaining the next stable vector.

It follows that p_0 does not decide a new name, so $p_0 \in DECIDED$, contradicting the fact that $p_0 \in UNDECIDED$. Since this fact follows from our general hypothesis that *UNDECIDED* contains some correct processors, we get

LEMMA 7.8. *In every run of the algorithm, all the correct processors eventually decide new names.*

THEOREM 7.9. *There is an algorithm for the uniqueness problem that uses the names* $\{1, \ldots, n + t\}$.

An extreme case in which $n + t$ names are actually needed by the algorithm is when the t processors with the smallest initial names suggest the names 1 through t and then fail and stop functioning. Thus, the names $\{1, \ldots, t\}$ are now occupied, and the collection of names from which the rest of the processors get to select their names is $POOL = \{t + 1, \ldots, n + t\}$. The correct processors then select their names sequentially in increasing order of original names. Each correct processor that gets to select a name views itself as the $t + 1$st contestant, and therefore has to select the $t + 1$st free name. Therefore, the correct processors end up selecting the names $\{2t + 1, \ldots, n + t\}$. The names $\{1, \ldots, t\}$ remain as suggestions made by the faulty processors, and the names $\{t + 1, \ldots, 2t\}$ remain free forever.

We remark that one can construct a simplified *probabilistic* version of this algorithm, in which the new name space is of size n, and processors suggest new names at random from the set $FREE(V)$ with equal probability. For such an algorithm, partial correctness is handled just as before, and (expected) termination can be proved by standard probabilistic arguments. As the realm of this paper does not concern efficiency, we do not enter the issue of expected number of rounds and related questions.

8. *The Order-Preserving Problem*

8.1 THE ALGORITHM. In this section we present an algorithm for the order-preserving problem that matches the lower bound of Section 4. We first give a simplified version of the algorithm, which makes the proof easier but yields a larger name space than is implied by the lower bound. Later, we show how to shrink the name space further, to match the lower bound.

Let us start with an informal description of the algorithm. The general structure is similar to the simple algorithm for the uniqueness problem (described in Section 6). During the execution of the algorithm, the processors store some information in a vector V. A processor is required to repeatedly exchange its information until reaching a stable vector. Then it can decide on its new name.

The vector held by processor p is of the form $V = \langle L, L_{n-t}, \ldots, L_n \rangle$, where each of the components is an ordered list of (old) processor names. The first list,

L, consists of all the processors known to p. The role of the other lists is slightly more complex. Later in this section, it is shown that, just as in the previous algorithms, at most $t + 1$ different sets of processors may occur as sets L in stable vectors during any run. Furthermore, these sets may be of size i for each $n - t \leq i \leq n$, and they are totally ordered by the subset relation. During the run, every processor p learns of some of these lists, and stores them in its vector V. Thus, each of the lists L_i in V is either empty or contains a set of exactly i processors, which are the processors occurring in the list L of some stable vector.

Note that processor p may learn of sets L_i as above in two different ways. When p obtains its stable vector V, the list of all processors known to p, L, is itself such a list. Besides, p may learn of such lists L_i from vectors it receives from other processors.

It turns out that for our purposes, the relative position of a processor's name is fully characterized by its ranks in these sets. Specifically, the name x chosen by processor p once obtaining a stable vector $V = \langle L, L_{n-t}, \ldots, L_n \rangle$ consists of a list of *rank numbers* $\langle x_{n-t}, x_{n-t+1}, \ldots, x_n \rangle$, where x_i is the rank of p in the list $L_i \cup \{p\}$, if $L_i \neq \emptyset$ in V, and 0, otherwise. Note that $0 \leq x_i \leq i + 1$; the case $x_i = i + 1$ occurs whenever $p \notin L_i$ and p is larger than any processor $q \in L_i$. The list L is henceforth called the *choice list* of x.

Whenever p gets a vector V' from one of the other processors, it has to check whether this vector contains some processors q or some lists L_i which p did not know of so far. In such a case, p has to update its vector, send it to all other processors and restart its attempt to obtain a stable vector. After obtaining a stable vector V, p can decide its name. The precise definition of the ordering on the new name space is designed to ensure that new names chosen in this way will preserve the original order.

Let us now proceed with more formal definitions.

Definition 8.1. The new name space, *OPNS*, contains names of the following form. Each name x consists of a sequence of $t + 1$ numbers, $\langle x_{n-t}, x_{n-t+1}, \ldots, x_n \rangle$, where $0 \leq x_i \leq i + 1$ and $0 \leq x_j - x_i \leq j - i$ for all $j > i$ such that both x_j and x_i are nonzero. We refer to these names as *rank sequences*.

Definition 8.2. Given a name $x = \langle x_{n-t}, \ldots, x_n \rangle$, the *choice length* of x, denoted $\mathrm{cl}(x)$, is the index of the last nonzero entry in x, that is,

(1) $x_{cl(x)} > 0$, and
(2) for every $i > \mathrm{cl}(x)$, $x_i = 0$.

The reason for this choice of terminology has to do with the fact that whenever a processor decides a new name x based on a choice list L we have $cl(x) = |L|$, that is, the choice length of x equals the length of its choice list.

The null value in *OPNS* is the sequence $\langle 0, \ldots, 0 \rangle$ (denoted 0 for simplicity) and its choice length is 0. We now define a partial order on the name space *OPNS*, as follows.

Definition 8.3. Let $x = \langle x_{n-t}, \ldots, x_l, 0, \ldots, 0 \rangle \neq 0$ and $x' = \langle x'_{n-t}, \ldots, x'_m, 0, \ldots, 0 \rangle \neq 0$ be two names in *OPNS* where $n - t \leq l = cl(x) \leq n$ and $n - t \leq m = cl(x') \leq n$ (note that by Definition 8.2 $x_l \neq 0$ and $x'_m \neq 0$). Without loss of generality assume that $l \leq m$. Then, x and x' are comparable only if

(a) $x'_l > 0$ and
(b) $m = l$ implies that $x'_m \neq x_l$.

Their ordering is determined as follows:

(1) case $l = m$, then

(1.1) case $x_l < x'_m$, then $x < x'$,
(1.2) case $x_l = x'_m$, then x and x' are incomparable (see (b) above),
(1.3) case $x_l > x'_m$, then $x > x'$,

(2) case $l < m$ then

(2.1) case $x_l < x'_l$, then $x < x'$,
(2.2) case $x_l = x'_l$, then $x > x'$,
(2.3) case $x_l > x'_l$, then $x > x'$.

This somewhat involved ordering is needed to ensure that whenever processors p and p' decide new names, if these names are at all *comparable*, then their relative order is the same as that of p and p' (see the proofs of Lemmas 8.6 and 8.10). Note that the ordering is not defined for every pair of names in *OPNS*. However, the algorithm guarantees that names that are actually chosen in any specific run are always comparable.

Again we need a partial ordering for vectors too.

Definition 8.4. For two vectors $V = \langle L, L_{n-t}, \ldots, L_n \rangle$ and $V' = \langle L', L'_{n-t}, \ldots, L'_n \rangle$, we say that $V \leq V'$ iff $L \subseteq L'$ and for every i, $n - t \leq i \leq n$, $L_i \subseteq L'_i$. (In fact, it follows from previous discussion that either $L_i = L'_i$ or $L_i = \emptyset$.)

Suppose that a processor holds the vector V, and gets a vector V' so $V' \nleq V$. This means that either L' contains some processors not in L, or there is some nonempty list L'_i in V' so $L_i = \emptyset$. In such a case, the processor can *update* its vector V based on V'. This operation consists of including any newly found processor in L, and setting $L_i \leftarrow L'_i$ for any i for which V' was more updated.

Let us now give a precise description of the algorithm for the order-preserving problem.

Algorithm B

```
0. Construct an initial $V$ with $L = \{p\}$ and $L_i = \emptyset$ for every $n - t \leq i \leq n$.
1. /* sending a new vector */
   a. Send $V$ to every other processor.
   b. $c \leftarrow 1$.
2. Wait until you receive a message $V'$.
   a. **If** $V' < V$ **then goto** 2.
   b. **If** $V' \nleq V$ **then** update $V$ according to $V'$ and **goto** 1.
   c. **If** $V = V'$ **then**:   /* received another identical copy */
                  $c \leftarrow c + 1$.
                  **If** $c < n - t$ **then goto** 2 **else goto** 3.
3. /* $V$ is a stable vector: decide a name */
   a. $l \leftarrow |L|$.
   b. $L_l \leftarrow L$.
   c. Send $V$ to every other processor.
   d. $x^p \leftarrow 0$.
   e. **For** every $L_i \neq \emptyset$ in $V$, **set** $x_i^p$ to be the rank of $p$ in the list $L_i \cup \{p\}$.
   f. Decide $x^p$.
4. /* echo stage—helping other processors */
   Continue forever; whenever you receive a message $V'$:
   a. Update $V$ if necessary.
   b. Send $V$ to every other processor.
```

8.2 Correctness of Algorithm B. As in the previous section we immediately have from Definitions 6.1 and 8.4 and from Lemma 6.1

COROLLARY 8.1. *Algorithm B is locally proper with respect to our vector ordering.*

COROLLARY 8.2. *In any run of algorithm B, the set of stable vectors is totally ordered.*

The above corollary implies

COROLLARY 8.3. *If $V = \langle L, L_{n-t}, \dots, L_n \rangle$ and $V' = \langle L', L'_{n-t}, \dots, L'_n \rangle$ are stable vectors and $|L| = |L'|$, then $L = L'$.*

COROLLARY 8.4. *If two processors p, p' hold nonempty lists, L_i, L'_i respectively, for some $n - t \le i \le n$, then $L_i = L'_i$.*

We now prove that every two new names preserve the order of the old names.

LEMMA 8.5. *If processors p and p' have decided on new names x and x', respectively, then x and x' are comparable.*

PROOF. Let $V = \langle L, L_{n-t}, \dots, L_n \rangle$ (respectively, $V' = \langle L', L'_{n-t}, \dots, L'_n \rangle$) be the stable vector which p (respectively, p') held when deciding its new name. Corollary 8.2 implies that in any run of the algorithm the set of stable vectors is totally ordered; therefore, V and V' are comparable.

Let $l = cl(x)$, $l' = cl(x')$, and without loss of generality assume that $l \le l'$. If $l = l'$, then p and p' are in L_l as a processor always belongs to its choice list. Consequently, their rank is not the same, that is, $x_l \neq x'_l$. By Definition 8.3(b), x and x' are comparable.

Otherwise, $l < l'$. Now, assume to the contrary that x and x' are not comparable. By Definition 8.3(a), this assumption implies that $x'_l = 0$. Contradiction is now derived from the fact that V and V' are comparable together with the following two observations:

(1) Since $x'_{l'} > 0$ (Definition 8.2(1)) and $x_{l'} = 0$ (Definition 8.2(2)), necessarily $V' \not\le V$.
(2) Since $x'_l = 0$ (by assumption) and $x_l > 0$ (Definition 8.2(1)), necessarily $V \not\le V'$. □

LEMMA 8.6. *If processors p and p' have decided on new names x and x', respectively, then these names preserve the original ordering.*

PROOF. By the previous lemma, x and x' are comparable. Let $l = cl(x)$, $l' = cl(x')$ and without loss of generality assume that $l \le l'$.

By Corollary 8.4, there is only one list L_l of length l occurring in stable vectors. By the choice of new names (in Step 3 of the algorithm), x_l is the rank of p in $L_l \cup \{p\}$. Since L_l is the choice list of x, necessarily $p \in L_l$, so x_l is the rank of p in L_l as well. Similarly, x'_l is the rank of p' in $L_l \cup \{p'\}$.

There are two cases to consider. If $p < p'$, then x'_l is strictly larger than x_l; hence, $x < x'$ by Definition 8.3 (1.1 and 2.1). So assume $p > p'$. If $x'_l < x_l$, then $x' < x$ by Definition 8.3 (1.3 and 2.3). The only remaining case is when $p' \notin L_l$ and $x'_l = x_l$. This may happen, for instance, when p and p' are adjacent in the original list of processor names, because x'_l is the rank of p' in $L_l \cup \{p'\}$ rather than L_l. However, Definition 8.3 (2.2) ensures that even here the right ordering is preserved. □

This completes the proof that algorithm B is partially correct for the order-preserving problem. We now prove the termination of algorithm B.

LEMMA 8.7. *In every run of the algorithm, every correct processor eventually decides a new name.*

PROOF. We need to establish that every correct processor eventually leaves the loop at Step 2.

Each processor p updates its vector V by either including a new processor in the set L or setting one of its lists L_i. The set L can be updated at most $n - 1$ times. Each list L_i is set at most once, so there are at most $t + 1$ updates of this type. This bounds the number of times a processor returns to Step 1 and retransmits its vector.

The rest of the proof is straightforward and follows Lemma 6.5. □

8.3 REDUCING THE NAME SPACE. A direct calculation shows that the algorithm as presented requires a new name space of size at most $(n + 1)^{t+1}$. Here we slightly modify the algorithm in two stages. A careful analysis of the second modification implies that the modified solution uses a name space of size $2^t(n - t + 1) - 1$, tightly matching the lower bound.

Definition 8.5. A *complete rank sequence*, or *CRS*, is a rank sequence $\langle x_{n-t}, \ldots, x_l, 0, \ldots, 0\rangle$ such that $x_i \neq 0$ (or, $1 \leq x_i \leq i + 1$) for all $n - t \leq i \leq l$.

Note that in a CRS, $0 \leq x_i - x_{i-1} \leq 1$ for every $n - t < i \leq l$.

LEMMA 8.8. *Every rank sequence $\langle x_{n-t}, \ldots, x_l, 0, \ldots, 0\rangle$ can be extended into a CRS $\langle x'_{n-t}, \ldots, x'_l, 0, \ldots, 0\rangle$ such that, if $x_j \neq 0$, then $x'_j = x_j$.*

Definition 8.6. For a rank sequence $x = \langle x_{n-t}, \ldots, x_l, 0, \ldots, 0\rangle$, define the CRS $f(x)$ to be the largest (lexicographically) CRS that extends x.

Note that a CRS x can be extended only to itself and therefore $f(x) = x$.

First Modification of Algorithm B. Modify Step 3f of the algorithm to be: decide $f(x^p)$.

LEMMA 8.9. *The modified algorithm is still correct.*

PROOF. We have to show that for every two processors p and q, the CRS-extensions $y^p = f(x^p)$ and $y^q = f(x^q)$ preserve the same ordering of x^p and x^q. The choice lengths of x^p and x^q remain the same in y^p and y^q, as no entries to their right are changed. Let $l = \min\{cl(x^p), \mathrm{cl}(x^q)\}$. It follows that x_l^p and x_l^q are nonzero. Therefore, y_l^p and y_l^q are nonzero and equal to x_l^p and x_l^q. Thus, the same ordering rule from Definition 8.3 is applied. □

We comment that this modification already reduces the size of the new name space to less than twice the value of the lower bound given in Section 4. We now proceed to reduce this size further and match it with the lower bound. This is done by showing that one can actually do with only about half the above sequences.

Definition 8.7

(1) The class C is the class of all CRSs $x = \langle x_{n-t}, \ldots, x_l, 0, \ldots, 0\rangle$ such that $l > n - t$ and $x_{l-1} < x_l$.

(2) For every CRS, $x \in C$, let $m(x)$ be the maximal index $n - t < m \leq l$ such that $x_{m-1} = x_m$, if exists, and $m(x) = n - t$, otherwise.

(3) For every CRS $x = \langle x_{n-t}, \ldots, x_l, 0, \ldots, 0\rangle$, if $x \in C$ then

$$g(x) = \langle x_{n-t}, \ldots, x_{m-1}, x_m + 1, x_{m+1} + 1, \ldots, x_{l-1} + 1, x_l, 0, \ldots, 0\rangle,$$

where $m = m(x)$, else $g(x) = x$.

Note that in Part (2) of the definition the only case where there is no such index is when $p \in L_{n-t}$ and all the processors in $L_l - L_{n-t}$ have (old) names strictly smaller than that of p. For Part (3) note that $g(x)$ is a CRS and $g(x) \notin C$.

Second Modification of Algorithm B. Modify Step 3f of the algorithm to be: decide $g(f(x))$.

LEMMA 8.10. *The modified algorithm is still correct.*

PROOF. We have to show that for every two processors p and q, $y^p = g(f(x^p))$ and $y^q = g(f(x^q))$ preserve the same ordering of x^p and x^q. Note that the choice lengths of x^p and x^q remain the same in y^p and y^q and without loss of generality assume that $l = cl(x^p) \leq m = cl(x^q)$. It follows that x_l^p and x_l^q are nonzero.

If $l = m$, then $x_l^p = y_l^p$ and $x_l^q = y_l^q$. Therefore, y^p and y^q preserve the order of x^p and x^q.

Now assume that $l < m$ and hence $x_l^p = y_l^p$. The following case analysis completes the proof.

(1) $x^p < x^q$. By Definition 8.3 (2.1) $x_l^p < x_l^q$ and by Definition 8.7 (3) $x_l^q \leq y_l^q$. Consequently $y_l^p = x_l^p < x_l^q < y_l^q$, which implies (Definition 8.3 (2.1)) that $y^p < y^q$ and the order is preserved.

(2) $x^p > x^q$. By Definition 8.3 (2.2 and 2.3) $x_l^p \geq x_l^q$.
   - 2.1 $x_l^p > x_l^q$. As $y_l^q \leq x_l^q + 1$ (Definition 8.7 (3)) it follows that $y_l^p = x_l^p \geq y_l^q$. By Definition 8.3 (2.2 and 2.3), we get that $y^p > y^q$ and the order is preserved.
   - 2.2 $x_l^p = x_l^q$. This can happen only if q is not in the list L_l (see the proof of Lemma 8.6). Let L_j, $l < j$, be the first list such that $q \in L_j$ and $x_j^q \neq 0$ (i.e., q knows about the list L_j). It follows from the choice of $f(x^q)$ (Definition 8.6) that $x_{j-1}^q = x_j^q$ and hence if $f(x^q) \in C$ then $m(f(x^q)) > l$ (Definition 8.7 (2)). According to Definition 8.7, $y_l^q = x_l^q$, as only indices that are larger than $m(f(x^q))$ may be changed (Definition 8.7 (3)). Combining all together we get $y_l^p = x_l^p = x_l^q = y_l^q$ that implies (Definition 8.3 (2.2)) that $y^p > y^q$ and the order is preserved. □

Let us now analyze the size of the resulting name space. Every CRS $x = \langle x_{n-t}, \ldots, x_l, 0, \ldots, 0 \rangle$ can be encoded by a sequence $\langle x_{n-t}, \epsilon_1, \ldots, \epsilon_k \rangle$, where $\epsilon_i \in \{0, 1\}$ and $k = l - (n - t)$. This encoding is obtained by taking $\epsilon_i = x_{n-t+i} - x_{n-t+i-1}$. It follows from Definition 8.5 that $\epsilon_i \in \{0, 1\}$.

Let x be a name chosen by the processor p according to the second modification of the algorithm and let $k = cl(x) - (n - t)$. Using the encoding described above, there are two cases:

(1) If $k = 0$, then there are $n - t$ possible CRSs of the form $\langle x_{n-t}, 0, \ldots, 0 \rangle$ for $1 \leq x_{n-t} \leq n - t$, since p must appear in its choice list L_{n-t}.
(2) If $k \geq 1$, then the last modification implies that $\epsilon_k = 0$, and by Definition 8.5 there are $(n - t + 1)2^{k-1}$ possible CRSs.

The size of the new name space is thus

$$N = (n - t) + (n - t + 1) \sum_{k=1}^{t} 2^{k-1} = 2^t(n - t + 1) - 1.$$

THEOREM 8.11. *There is an algorithm for the order-preserving problem that uses the names* $\{1, \ldots, 2^t(n - t + 1) - 1\}$.

PROOF. Follows from the lemmas of this section and by a simple bijection from *OPNS* to $\{1, \ldots, 2^t(n - t + 1) - 1\}$. □

ACKNOWLEDGMENTS. We are grateful to Michael Ben-Or, Nati Linial, Mark Tuttle, and Lenore Zuck for helpful discussions and comments. Thanks are also due to two anonymous referees for their careful reading of earlier versions of this paper, and their detailed comments that greatly improved the presentation.

REFERENCES

1. ATTIYA, H., DOLEV, D., AND GIL, J. Asynchronous Byzantine consensus. In *Proceedings of the 3rd ACM Symposium of Principles of Distributed Computing* (Vancouver, B.C., Canada, Aug. 27–29). ACM, New York, 1984, pp. 119–133.
2. BAR-NOY, A., DOLEV, D., KOLLER, D., AND PELEG, D. Fault-tolerant critical section management in asynchronous environments. In *Distributed Algorithms, 3rd International Workshop* (Nice, France, Sept.) Lecture Notes in Computer Science, No. 392. Springer-Verlag, New York, 1989, pp. 13–23. *Inf. Comput.*, to appear.
3. BEN-OR, M. Another advantage of free choice: Completely asynchronous agreement protocols. In *Proceedings of the 2nd ACM Symposium of Principles of Distributed Computing* (Montreal, Que., Canada, Aug. 17–19). ACM, New York, 1983, pp. 27–30.
4. BRACHA, G. An $O(\log n)$ expected rounds randomized Byzantine generals protocol. In *Proceedings of the 17th Annual ACM Symposium on Theory of Computing* (Providence, R.I., May 6–8). ACM, New York, 1985, pp. 316–326.
5. BRIDGLAND, M. F., AND WATRO, R. J. Fault-tolerant decision making in totally asynchronous distributed systems. In *Proceedings of the 6th ACM Symposium of Principles of Distributed Computing* (Vancouver, B.C., Canada, Aug. 10–12). ACM, New York, 1987, pp. 52–63.
6. DOLEV, D., DWORK, C., AND STOCKMEYER, L. On the minimal synchronism needed for distributed consensus. *J. ACM 34*, 1 (Jan. 1987), 77–97.
7. DWORK, C., LYNCH, N., AND STOCKMEYER, L. Consensus in the presence of partial synchrony. *J. ACM 35*, 2 (Apr. 1988), 288–323.
8. FISCHER, M. J., LYNCH, N. A., AND PATERSON, M. S. Impossibility of distributed consensus with one faulty processor. *J. ACM 32*, 2 (Apr. 1985), 374–382.
9. KOLLER, D. Token survival: Resilient token algorithms. M.Sc. Thesis, Hebrew University, Jerusalem, Israel, 1986.
10. MORAN, S., AND WOLFSTAHL, Y. Extended impossibility results for asynchronous complete networks. *Inf. Proc. Lett. 26* (Nov. 1987), 145–151.
11. RABIN, M. O. The choice coordination problem. *Acta Inf. 17* (1982), 121–134.
12. RABIN, M. O. Randomized Byzantine generals. In *Proceedings of the 24th Symposium of Foundations of Computer Science* (Nov.). IEEE, New York, 1983, pp. 403–409.
13. TAUBENFELD, G. Impossibility results for decision protocols. Tech. Rep. #445. Technion, Haifa, Israel, Jan. 1987.

RECEIVED MAY 1987; REVISED AUGUST 1988 AND JULY 1989; ACCEPTED JULY 1989