September 1986

Report No. STAN-CS-86-1131

Processor Renaming in Asynchronous Environments

by

Amotz Bar-Noy and David Peleg

Department of Computer Science

Stanford University
Stanford, CA 94305

PROCESSOR RENAMING IN ASYNCHRONOUS ENVIRONMENTS §

Amotz ***Bar-Noy***
Hebrew University, Jerusalem, Israel
and
David Peleg ‡
Stanford University, Stanford, California

September 86

Abstract

Fischer, Lynch and Paterson [FLP] proved that in a completely asynchronous system "weak agreement" cannot be achieved even in the presence of a single "benign" fault. Following the direction proposed in [ABDK], we demonstrate the interesting fact that some weaker forms of processor cooperation are still achievable in such a situation, and in fact, even in the presence of up to $t < n/2$ such faulty processors. In particular, we show that n processors, each having a distinct name taken from an unbounded ordered domain, can individually choose new distinct names from a space of size $n + t$ (where n is an obvious lower bound). In case the new names are required also to preserve the original order, we give an algorithm in which the space of new names is of size $2^t(n - t + 1) - 1$, which is tight.

§ Part of this work was done while the authors were visiting IBM Almaden Research Center, San-Jose.

‡ Supported in part by a Weizmann fellowship and by grant ONR N00014-85-C-0731

1. Introduction

The problem of reaching agreement in distributed systems was the focus of many studies in the last few years. One aspect of this research is understanding the limitations of asynchronicity. In a fundamental paper, Fischer, Lynch, and Paterson [FLP] proved that in a completely asynchronous system n processors cannot achieve "weak consensus" in the presence of faults. This holds even if at most *one* processor may become faulty, and even in the mild form of failstop fault, i.e., when a processor may not start or may suddenly stop functioning.

This negative result and later stronger versions of it [DDS] left the impression that in the presence of faults one cannot expect to reach any nontrivial goal that requires participation of several processors. In this paper we follow the observations of [ABDK], indicating that there are some nontrivial goal functions which may be achieved in a completely asynchronous system, even in the presence of up to $t < n/2$ such faulty processors. We present an algorithm for the problem of reducing the name space of the processors, raised in [ABDK].

This problem can be described as follows. Each of the n processors initially has a unique name taken from an unbounded ordered domain (w.l.o.g., the integers). Throughout we identify the processors with their old names, and denote them by p, q, etc. We want 'an algorithm that enables each correct processor to choose (within a finite number of steps) a new name from a space of size N, where N depends only on n and t, such that one of the following requirements hold:

a. ***Uniqueness:*** every two new names are distinct.

b. Order ***preserving:*** If $p < q$ then the new name of p is smaller than the new name of q.

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.

These problems have also some practical significance, as the complexity of some distributed algorithms depends on the size of the names of the processors. One such example is the algorithm for the Choice Coordination Problem [R].

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 $p_1 < p_2 < \ldots < p_n$, 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.

We adopt a model similar to that of [FLP]. The distributed system is composed of n asynchronous processors $\boldsymbol{P}$ connected through a completely asynchronous communication network. Every processor can directly send messages to all the others, i.e., the network is complete. In an atomic step a processor either sends a message or reads a message. Every message arrives its destination within a finite but unbounded amount of time, and it arrives exactly as it was sent. The communication system does not maintain the original order of the messages, so a message m_2 may arrive before message m_1, even if m_1 was sent before m_2.

During the run of the system, processors might become faulty and stop working. We make the standard assumption that at most t processors may become faulty, i.e., perform only a finite number of atomic steps, and the rest (called correct) must continue forever. The two parameters n and t are known in advance to all of the processors.

Our results hold also for the case of "omission" faults, and are conceivably extendable, via standard techniques, to the case of malicious faults with authentication and the general malicious case (with appropriate restrictions on n vs. t).

We represent the asynchronous behaviour of the system using the concepts of a "scheduler" and a "postman". For every run, the scheduler creates an infinite schedule $S = (p_{j_1}, p_{j_2}, \ldots)$. The processor p_{j_k} makes the k's atomic step in the system. The postman holds all the messages that were sent, and must deliver them within a finite number of steps. Whenever a processor tries to read a message the postman decides whether or not to deliver it a message, and which message to deliver from among the messages that were sent to this processor. We say that an algorithm solves a given problem in a completely asynchronous system if the algorithm correctly solves the problem for an arbitrary scheduler and postman.

The main results of this paper are algorithms for the two versions of the renaming problem. For the uniqueness problem, our algorithm yields new names from a space of size $n + t$ (where n is an obvious lower bound). This improves the algorithm presented in [ABDK], which results in a space of size nt. A simple *probabilistic* version of our algorithm yields a name space of size n. For the order preserving problem, we get a new name space of size $2^t(n - t + 1) - 1$, which tightly matches the lower bound.

The rest of the paper is organized as follows. In section 2 we give some lower bounds concerning the two problems. Both algorithms for the uniqueness problem and for the order preserving problem use the same schematic frame which is described in section 3. This form of presentation enables us to separate the discussion of issues of partial correctness from those of termination. The specific details of the algorithms for the uniqueness problem and the order preserving problem are described in sections 4 and 5, respectively.

2. Lower Bounds

Theorem 2.1: ***There is no algorithm for solving the naming problems in a completely asynchronous system, if n $\leq 2t$.***

Proof: Assume to the contrary that $n = 2t$ and there is an algorithm A which solves one of the above problems. Look at the schedule $S = (p_1, \ldots p_t, p_1, \ldots, p_t, \ldots)$. which is legal because only t processors $p_{t+1}, \ldots, p_n$ are faulty (perform a finite number of steps). Algorithm A should let the t operating processors choose distinct names for every set of initial names. Since the original name space is unbounded and the space of the new names is finite, there exist two sets of processors' names, say $p_1, \ldots. p_t$ and $q_1, \ldots q_t$ where $p_1 \neq q_1$ such that A assigns p_1 and q_1 the same new name when the corresponding sets of processors participate with the appropriate schedules, $S_p = (p_1, \ldots. p_t, p_1, \ldots. p_t, \ldots)$ and $S_q = (q_1, \ldots. q_t, q_1, \ldots. q_t, \ldots)$. Now look at the schedule $S = (p_1, \ldots p_t, p_1, \ldots, p_t, \ldots, q_1, \ldots q_t, q_1, \ldots q_t, \ldots)$, in which the first appearance of q_1 is after $p_1, \ldots, p_t$ have chosen their names (this must happen within a finite number of steps). The postman delays all messages sent between the two sets of t processors until all the processors choose names (and again this occurs after a finite number of steps). For processor q_1 (resp., p_1) the schedule S_q (resp., S_p) is indistinguishable from the schedule S. Hence, they both choose the same name; a contradiction. ▮

An argument similar to the proof of Lemma 2.1 implies that one cannot design an algorithm in which processors stop after they choose new names, because once n $- 2t$ processors choose new names, the last t processors, which may start running after that decision, cannot choose correctly.

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

Theorem *2.2:* ***Every algorithm*** *for* ***solving the uniqueness problem requires N*** $\geq$ ***n.***

Proof: : We must have n different names for the n possibly correct processors that may want a new name. ▮

Theorem 2.3 [ABDK]: ***Every algorithm*** *for* ***solving the order preserving problem*** *requires* ***N*** $\geq 2^t(n-t+1)-1$. ▮

Remark: The lower bound of Thm. 2.3 holds also if the system is synchronous, but we do not know when a processor makes its first step.

3. The Schematic Algorithm

In this section we describe a schematic algorithm for the renaming problem. For the time-being we do not specify the structure of new names, the order relation on new names or the criteria for choosing a name, but rather give a general strategy for the process of selecting names and study its properties.

Throughout the algorithm each processor ***p*** maintains an ordered vector ***V*** containing information about the processors (old names) of which it knows. The processors dynamically update their vectors ***V*** by exchanging them with all the others. Each processor also maintains a counter c which is the number of processors that have indicated having the same set ***V.***

Since t processors might be faulty, a processor cannot expect to get more than $n - t$ messages from different processors (including itself). Thus, after receiving $c = n - t$

identical copies of the vector ***V,*** it makes no sense to wait for more information (which might never arrive), and the processor should take some action. This observation leads us to defining the basic notion of a ***stable*** vector.

Definition 3.1: A vector ***V*** is ***stable with respect to a processor p*** in a given run of the algorithm if ***p*** has received $n-t$ messages containing identical copies of ***V*** (including its own copy). ***V*** is ***stable*** if it is stable w.r.t. some processor ***p.***

The algorithm is based on the following general strategy. A processor is requires 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. In case 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, or some other problem arises), the processor has to make a new suggestion and repeat the process.

The issues that arise in this type of an algorithm are ***partial correctness,*** namely whether the algorithm guarantees that any names which are actually chosen obey the requirements of the problem, and ***termination,*** namely whether all correct processors eventually get to choose a new name. The schematic algorithm presented here is partially correct in this sense, but its termination properties depend on the specific details of the name structure and name selection procedures.

Data structures

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.
4. b_p, a "decision" bit, which is 1 if p already decided on a name and 0 otherwise.
5. ***L,,*** the list of ***old*** names appearing in the vector by which x_p was suggested.

We should comment that each of the specific algorithms derived for our problems uses only part of the above information. In particular, the algorithm for the uniqueness problem does not need the L_p component, and the algorithm for the order preserving problem can do without the x_p, J_p and b_p components. However we include all of them in the schematic algorithm for unified presentation.

Throughout the rest of the paper we use the following notation with respect to vectors ***V.*** Denote by $D_0(V)$ the subvector containing exactly the entries in which $b = 0$ (i.e., the entries of processors that have not yet decided on a name). Let ***P(V)*** be the set of processors (old names) in ***V,*** and let X(V) be the set of new names in ***V.*** Denoting the space of new names by NS, let ***free(V)*** = NS $-$ ***X(V)*** be the set of free (unsuggested) names in ***V.*** As a rule, sets of (old or new) names are always taken to be ordered. We informally refer to the number of entries in $V\ (=\mid P(V)\mid)$ as $\mid V \mid$, and say $p \in V$ or $x \in V$ as a shorthand for $p \in P(V)$ or $x \in X(V)$, respectively.

Suppose a processor ***p*** suggests a name x_p on the basis of a stable vector ***V.*** This vector is henceforth referred to as the ***choice vector*** of x_p, and the list ***P(V)*** of ***old*** names (to be inserted into future vectors as L_p) is called the ***choice list*** of x_p. Similarly, $\mid V \mid$ and the index of p in ***V*** are referred to ***as*** the choice ***length*** and the ***choice index*** of x_p, respectively.

The vectors are ordered by increasing order of the old names ***p.*** Initially, the vector held by ***p*** contains only one column corresponding to ***p*** itself, with all components except the first set to the appropriate null values.

We need a certain partial ordering on vectors. This ordering reflects the accumulation of knowledge in the processors. Intuitively, $V > V'$ means that ***V*** is more updated than V'. The ordering is defined as follows.

Definition 3.2: For- two vectors V_1, V_2, we say that $V_1 \leq V_2$ iff for every processor ***p,*** $p \in V_1 \Rightarrow (p \in V_2$ and J_p in V_2 is no smaller than J_p in $V_1)$.

The current validity requirement

Suppose a processor p has suggested a name x_p for itself, exchanged information with the other processors and reached a stable vector ***V.*** This name x_p has to satisfy the requirement that it is currently valid. This requirement, later referred to as the requirement of current validity, or ***requirement*** CV, translates into the following conditions:

In the uniqueness case: The name x_p is different from any other suggested name $x_q \in X(V)$.

In the order preserving case: The name x_p preserves the ordering with respect to any other entry in the vector. I.e., for every ***q,*** if $x_q \in X(V)$, then $x_p < x_q$ iff $p < q$. (Note that these may be different name spaces and order relations.)

The schematic algorithm A

```
/* For a processor p. Counter c counts the number of identical copies of V p gets. */
0. Construct an initial V (with one entry - for p).
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 /* V' contains old information */
                return to 2
    b. if $V' \not\leq V$ (i.e., there is some $q \in V'$ s.t. $q \not\in V$ or $J_q$ in V is smaller than in V)
       then
                update V
                return to 1
    c. if V = $V'$ then /* heard of another identical copy */
                $c \leftarrow c + 1$
                if $c < n - t$ then return to 2 else goto 3.
3. /* V is a stable vector */
   If previously suggested a name $x_p$, and this name satisfies requirement CV then
    a. decide on $x_p$
    b. $b_p \leftarrow 1$
    c. send V to every other processor
    d. goto 5.
4. /* needs to suggest a new name */
    a. test for condition $C_A$ to decide whether to suggest a name. If "no" return to 2.
    b. suggest a name $x_p$ using procedure $P_A$.
    c. insert $x_p$ to V
    d. $J_p \leftarrow J_p + 1$
    e. update $L_p$ if necessary.
    f. return to 1.
5. /* echo stage - helping other processors */
   continue for ever:
    a. read a message V'
    b. update V if necessary.
    c. send V to every other processor.
```

Note that condition C_A and procedure P_A are left unspecified.

Partial correctness

We need some lemmas concerning the order relation on vectors. The first trivial observation follows directly from the definition of the algorithm ***A.***

Lemma 3.1: ***FOT every processor p and for every run of A, the set of vectors held by p during the run is totally ordered.*** ▮

In fact, this ordering corresponds directly to the time ordering in which these vectors were held by ***p.***

Lemma *3.2:* ***In any run of A, the set of stable vectors is totally ordered.***

Proof: Assume ***V*** and ***V'*** are two incomparable stable vectors obtained in the same run. ***V*** is stable w.r.t. 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 vectors at different stages of the run, which contradicts Lemma 3.1. ▮

We now prove the following partial correctness claim with respect to the schematic algorithm.

Lemma 3.3: ***If p and q have decided on new names x_p and x_q respectively, then these names satisfy the requirements of the problem.***

Proof: Assume that x_p and x_q do not satisfy the requirement of the problem. Let V_p (resp. V_q) be the stable vector which ***p*** (resp. ***q***) held when deciding upon its new name.

If $V_p < V_q$ then the name x_p must appear also in V_q, since ***p*** never changes its suggested name after decision. Therefore the choice of x_q by ***q*** is invalid, as it conflicts requirement CV. A similar contradiction follows from assuming $V_q < V_p$ or $V_p = V_q$. ▮

4. The Uniqueness Problem

In this section we describe an algorithm for the uniqueness problem and prove its correctness.

The new name space is UNS = $\{1, \ldots, n + t\}$, and we use an instance of the schematic algorithm. We have to specify the details of step *4* of the algorithm, particularly condition C_A and procedure P_A.

Suppose ***p*** obtains a stable vector V and let ***r*** be the index of ***p*** in $D_0(V)$.

Condition C_A: If $r > t + 1$ then "no" /* do not suggest any name */

Procedure P_A: Let $x_p \leftarrow$ (free(v))(r).

Partial correctness follows from the proof of the schematic algorithm. It is also clear that whenever a process has to suggest a name, such a name is available, since always

$|$ ***free(V)*** $| \geq t + 1$, hence no processor "gets stuck". It remains to prove termination. This requires us to show that every correct processor eventually gets to decide on a new name. We prove this by assuming the opposite and deriving a contradiction.

Assume the existence of a run (scheduler plus postman) in which some of the correct processors p continue running forever with $b_p = 0$. Let us introduce the following notation. Denote by D_1 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 $D_0 = P - D_1$. Our hypothesis is that D_0 contains at least one correct processor. Denote by F, $F \subseteq D_0$, the set of all processors p that stop increasing J_p (but remain with $b_p = 0$) from some point on, and let $W = D_0 - F$.

Lemma 4.1: $W \neq \emptyset$.

Proof: Assume to the contrary that $W = \emptyset$, or $F = D_0$. 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 D_0 (there is such a processor by our general hypothesis). Its index in D_0 is at most $t + 1$, so 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 W.

Since $W \neq \emptyset$, 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 $D_1 \cup F$ have reached their final J value (hence they do not suggest any new names afterwards, and in particular, all processors in D_1 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 $V > V_L$ as a ***limit vector.*** Note that for all limit vectors V the subvector ***Do(V)*** is fixed and the set of processors in it, $P(D_0(V))$, is exactly D_0.

Denote the index of p_0 in the set D_0 by r_0. By the rules of the algorithm it is clear that for every $p \in W$, the index of p in the set D_0 is at most $t + 1$. In particular,

Lemma 4.2: $r_0 \leq t + 1$. ∎

Let us now classify the names in UN S as follows. Let X_{D_1} (resp., X_F) denote the set of (final) new names suggested by processors in D_1 (resp., F), and let $G = UNS - (X_{D_1} \cup X_F)$. Intuitively, G is the set from which the processors of W continuously attempt to choose names. For every stable limit vector V let $X_W(V)$ denote the set of new names suggested by the processors of W in V. Note that for every limit vector V, G = ***free(V)*** $\cup X_W(V)$. Assume that G is ordered, and let $G = \{x_1, x_2, \ldots\}$. For every stable limit vector V and for every name $x \in$ ***free(V),*** denote by f(x) its index in ***free(V).*** Clearly $f(x_i) \leq i$.

There is a later point in time after which every processor in W have 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 in which the choice vector of every name in $X_W(V'_L)$ is a limit vector.

Lemma 4.3: ***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 suggested by some $q \in W$, $q \neq p_0$. Then q suggested x_{r_0} according to some stable limit vector V'. But then $f(x_{r_0}) \leq r_0$ in $free(V')$, so q could not have suggested it, as its index in D_0 is strictly larger than r_0. ▮

Therefore, upon seeing V'_L, p_0 either decides immediately on x_{r_0} as its name (in case 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 decide on a new name, contradicting the assumption that $p_0 \in D_0$. This proves

Lemma 4.4: ***In every run of the algorithm, all the correct processors eventually decide on new names.*** ▮

Theorem 4.5: ***There is an algorithm for the uniqueness problem which uses the names $\{1, \ldots, n+t\}$.*** ▮

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.

5. The Order Preserving Problem

5.1. The Algorithm

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 indicate how to shrink the name space further, so as to match the lower bound. The name space, $OPNS$, contains names which are of the following form. Each name x consists of a sequence of $t+1$ numbers, $< x_{n-t}, x_{n-t+1}, \ldots, x_n >$, where $0 \leq x_i \leq n$ 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 sometimes call such names ***legal sequences.***

The rightmost nonzero entry in a name x, x_K, plays an especially important role in the description of the algorithm. In every suggested name x, this K corresponds to the choice length of x, and x_K itself is the choice index of x.

We define a partial order on the name space ***OPNS*** as follows.

Definition 5.1: Let $x_1 =< x^1_{n-t}, \ldots, x^1_n >$ and $x_2 =< x^2_{n-t}, \ldots, x^2_n >$ be two names in ***OPNS,*** with choice lengths K_1 and K_2, respectively. Without loss of generality assume $K_1 \geq K_2$. We say that $x_1 > x_2$ iff $x^1_{K_2} > x^2_{K_2} > 0$.

The order relation is not defined for every pair of names in ***OPNS,*** but the algorithm guarantees that names that are actually chosen in any specific run are always ordered.

Again we use an instance of the schematic algorithm. We have to specify the process of suggesting a name. Initially for every p, x_p is the all-zero tuple. Suppose p obtains a stable vector V of length K in which it is in the r'th entry. The condition C_A is always "yes", that is, a processors never "passes" the opportunity to suggest a name. The name. suggestion procedure P_A is defined as follows. New suggestions always update previous ones, in the sense that some zero entries are changed into nonzero ones (but nonzero entries do not change again). Specifically,

(1) if $x^p_K = 0$ then $x^p_K \leftarrow r$.

(2) for every $x' \in X(V)$ whose choice length and choice list are K' and L' respectively, if $x^p_{K'} = 0$ then $x^p_{K'} \leftarrow \alpha$, where α is the unique integer satisfying $L'(\alpha - 1) < p \leq L'(\alpha)$.

Again, partial correctness follows from the proof of the schematic algorithm. It is also clear that whenever a process has to suggest a name, the procedure P_A does not "get stuck". It remains to prove termination.

Lemma *5.1: If V and V' are stable vectors of the same length then P(V) = $P(V')$.*

Proof: Immediate from Lemma 3.2 and Definition 3.2. ▮

Lemma *5.2: **Suppose a processor p obtains a stable vector V in which its suggested name x_p violates requirement CV w.r.t. a suggested name x_q, whose choice length is Ii'. Then x^p_K = 0.***

Proof: Assume that $x^p_K = \alpha \neq 0$. Then a describes the index of p in $P(V)$ for some stable vector V of length K. Let V' be the choice vector of x_q. Since K is the choice length of x_q, $|V'|=|V|$, so by Lemma 5.1 $P(V') = P(V)$. Let β be the index of q in $P(V')$ ($\beta = x^q_K$). By the choice of α, $p > q$ iff $\alpha > \beta$, or, iff $x^p_K > x^q_K$, which, by the definition of the ordering on ***OPNS,*** holds iff $x_p > x_q$, which means that x_p and x_q do not violate requirement CV, contradicting the assumption of the lemma. ▮

Lemma *5.3: **In every run of the algorithm, every correct processor decides on a new name after at most $t + 1$ suggestions.***

Proof: In every suggestion made by p_i, at least one entry in x_i changes from zero to a nonzero value, and there are only $t + 1$ entries. Therefore a processor cannot make more than $t + 1$ suggestions. Since processors that decide do not change their entries any more, and processors with $b = 0$ do not make any changes (new suggestions) before obtaining a new stable vector, it is clear that as long as there are processors with $b = 0$, some of them will eventually obtain a stable vector and either decide or make a new suggestion, until all processors decide. ▮

5.2. Reducing the name space

In this section we slightly modify the algorithm, and then show that the number of different possible sequences that processors can choose is $2^t(n-t+1)-1$.

Definition 5.2: A ***complete legal sequence,*** or CLS, is a legal sequence $<x_{n-t},\ldots x_K,0,\ldots 0>$ such that for all $n-t \le i \le K$, $x_i \neq 0$.

Lemma 5.4: ***Every legal sequence $<x_{n-t},\ldots x_K,0,\ldots 0>$ can be extended into a CLS $<x'_{n-t},\ldots x'_K,0,\ldots 0>$ such that if $x_j \neq 0$ then $x'_j = x_j$.*** ▮

We now modify step 3a of the algorithm to be: Choose a CLS y_p which is an arbitrary extension of x_p and decide on y_p.

Lemma 5.5: ***The modified algorithm is still correct.***

Proof: We have to show that for every two processors p and q, the names y_p and y_q preserve the same ordering as the original x_p and x_q. Let K_p (resp., K_q) be the choice length of x_p (resp., x_q). These lengths remain the same in y_p and y_q, as no entries to their right are changed. Assume w.l.o.g. that $K_p \ge K_q$. It follows that $x^p_{K_q}$ and $x^q_{K_q}$ are nonzero, so they are not changed in y_p and y_q. Thus, the same ordering is preserved. ▮

Let us now analyze the size of the resulting name space. Every CLS $x =< x_{n-t},\ldots,x_K,0,\ldots,0>$ can be encoded by a sequence $< r, \epsilon_1,\ldots \epsilon_k >$, where $\epsilon_i \in \{0,1\}$ and $k = K-(n-t)$. This encoding is obtained by taking $r = x_{n-t}$ and $\epsilon_i = x_{n-t+i} - x_{n-t+i-1}$. The total number of possible names now becomes

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

This value is already less than twice the value of the lower bound given in Thm. 2.3. We now proceed to reduce this value 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, e.g., those whose last bit is 0.

Define C as the class of all CLS's $x =< \alpha_{n-t},\ldots,\alpha_K,0,\ldots,0>$ such that $K > n-t$ and $\alpha_{K-1} < \alpha_K$.

For every $x =< \alpha_{n-t},\ldots \alpha_K,0,\ldots 0 > \in C$, let

$$f(x) = <\alpha_{n-t},\ldots \alpha_{m-1},\alpha_m+1,\alpha_{m+1}+1,\ldots,\alpha_{K-1}+1,\alpha_K,0,\ldots 0>,$$

where m is the maximal index s.t. $\alpha_{m-1} = \alpha_m$, if exists, and $m = n-t$ otherwise. Note that $f(x)$ is a CLS and $f(x) \notin C$.

Lemma *5.6: Let x $\in$ C and let y = f(x) as above. If some processor p decides on x then no other processor decides on y.*

Proof: The claim follows from the fact that the names x and y have the same choice length and the same choice index, so they cannot be chosen together in the same run. ▮

Lemma 5.7: *Let x $\in$ C and let y = f(x) as above. If some processor p decides on x then:*

a. For every CLS z, if some processor q decides on z then z is comparable with y and $z > y$ iff $z > x$.

b. For every CLS z $\in$ C, if some processor q decides on z and w = f(z), then w is comparable with y and w > y iff w > x.

Proof: For the first claim, assume that some processor ***q*** decides on z, and let K_z be the choice index of the name z. We distinguish between the following two cases.

z < x: If $K_z \leq K$ then $z_{K_z} < x_{K_z} \leq y_{K_z}$. Otherwise, $z_K \leq x_K = y_K$. In both cases it follows that z < y .

z > x: This case is divided further into three subcases.

(1) $K_z \geq K$: Then $z_K > x_K = y_K$ which implies z > y.

(2) $K_z <$ ***m:*** Then $z_{K_z} > x_{K_z} = y_{K_z}$ (because x and y are identical up to the ***m*** $- 1$'th entry), so z > y.

(3) $K > K_z \geq$ ***m:*** If $z_{K_z} \geq y_{K_z}$ then immediately z > y. Otherwise, we get a contradiction to the maximality of ***m*** as follows. In this case, $z_{K_z} = x_{K_z} = y_{K_z} - 1$. The choice list of ***q*** could not have contained ***p,*** because if it did then the K_z'th entry in the name suggested by ***p*** would have to be different than α_{K_z}. Therefore, the first stable vector obtained by ***p*** was of length m', $m' > K_z \geq$ ***m,*** which implies that $x_{m''} = x_{m''-1}$ for some m < $m'' \leq$ ***m***; contradicting the definition of ***m.***

The second claim follows from a slightly more involved case analysis, based on similar considerations. ▮

We now modify the algorithm once more, changing step 3a of the algorithm to be: Choose a CLS y_p which is an arbitrary extension of x_p. If $y_p \notin$ C then decide on it. Otherwise, decide on ***f*** (y_p).

Lemma *5.8: The modified algorithm is still correct.*

Proof: We have to show that for every two processors ***p*** and ***q,*** the final names chosen by ***p*** and ***q*** preserve the ordering of y_p and y_q, which by Lemma 5.5 is identical to the ordering of the original x_p and x_q. This follows directly from the last two lemmas. ▮

Finally we re-analyze the size of the resulting name space. Using the encoding described earlier, There are two cases:

1. If $k = 0$ then there are $n - t$ possible CLS's.
2. If $k \geq 1$ then $\epsilon_k = 0$ according to lemma 5.6, and there are $(n - t + 1)2^{k-1}$ possible CLS's (since if p does not appear in the list with length $n - t$ and his name is greater than all the name in that list then $r = n - t + 1$).

   The size of the new name space is thus

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

Theorem 5.9: ***There is an algorithm for the order preserving problem which uses the names*** $\{1, \ldots, 2^t(n - t + 1) - 1\}$. ▮

Acknowledgements

We wish to thank C. Attiya, M. Ben-Or, D. Dolev, D. Koller and N. Linial for helpful discussions.

References

[ABDK] H. Attiya, A. Bar-Noy, D. Dolev and D. Koller, Borderline Cases in Asynchronous Achievability, Tech. Report 86-11, August 1986, Dept. of Computer Science, the Hebrew Univ. of Jerusalem, Israel.

[DDS] D. Dolev, C. Dwork and L. Stockmeyer, On the Minimal Synchronism Needed for Distributed Consensus, *Proc.* 24th *Symp.* on Foundations of *Comp. **Science, 1983.***

[FLP] M.J. Fischer, N.A. Lynch, M.S. Paterson, Impossibility of Distributed Consensus with one Faulty Processor, *Proc. **2nd Symp.*** on ***principles of*** Database ***systems,*** 1983.

[R] M.O. Rabin, The Choice Coordination Problem, *Acta Informatica* **17**, (1982), pp. 121-134.