RAIRO. INFORMATIQUE THÉORIQUE ET APPLICATIONS

PAUL SPIRAKIS
BASIL TAMPAKAS

Efficient distributed algorithms by using the archimedean time assumption

RAIRO. Informatique théorique et applications, tome 23, n° 1 (1989), p. 113-128

<http://www.numdam.org/item?id=ITA_1989__23_1_113_0>

Informatique théorique et Applications/Theoretical Informatics and Applications
(vol. 23, n° 1, 1989, p. 113 à 128)

EFFICIENT DISTRIBUTED ALGORITHMS BY USING THE ARCHIMEDEAN TIME ASSUMPTION (*)

by Paul SPIRAKIS ([1, 2]) and Basil TAMPAKAS ([1])

Abstract. – *This work examines the effect of limited asynchrony on three fundamental problems of distributed computation: The problem of symmetry breaking in a logical ring, that of mutual exclusion and the problem of readers and writers. We assume our distributed system to be Archimedean in the sense that processors know upper and lower bounds on the message delays and processor speeds. We use the knowledge of those bounds to get algorithms for the above mentioned problems which well improve the efficiency of algorithms presented by previous research. For the symmetry breaking problem we get a protocol which admits arbitrary initiation, and uses only linear number of message* bits *and linear time on the average. For the mutual exclusion problem we break the lower bound on the number of messages which holds in case of unrestricted asynchrony. We also find an important difference between Archimedean and Synchronous networks. Our algorithms are practical in the sense that any existing distributed system up to now follows the Archimedean time assumption.*

Résumé. – *Ce travail étudie l'effet de l'asynchronisme limité sur trois problèmes fondamentaux du calcul distribué: le problème de la rupture de symétrie dans un anneau logique, celui de l'exclusion mutuelle et le problème des lecteurs et des écrivains. Nous supposons notre système distribué archimédien, en ce sens que les processeurs connaissent les bornes inférieures et supérieures sur les retards de messages et sur les vitesses des processeurs. Nous utilisons la connaissance de ces bornes pour obtenir des algorithmes pour les trois problèmes cités, qui améliorent considérablement l'efficacité des algorithmes présentés antérieurement. Concernant le problème de la rupture de symétrie, nous obtenons un protocole qui admet une initialisation arbitraire, et qui prend seulement un nombre linéaire de bits de message, et un temps moyen linéaire pour s'exécuter. Concernant l'exclusion mutuelle, nous passons en dessous de la borne inférieure du nombre de messages qui existe dans le cas de l'asynchronisme non restreint. Nous montrons aussi une importante différence entre les réseaux archimédiens et synchrones. Nos algorithmes ont une portée pratique, puisque tous les systèmes distribués existants suivent l'hypothèse du temps archimédien.*

(*) This research was funded in part by the NSF contract DCR 8503497 and by the Ministry of Industry, Energy and Technology of Greece.

([1]) Computer Technology Institute, PO BOX 1122, 26110, Patras, Greece.
([2]) Courant Institute of Mathematical Sciences, 251 Mercer St, NY NY 10012, U.S.A.

Informatique théorique et Applications/Theoretical Informatics and Applications
0296-1598 89/01 113 16/$ 3.60/

1. INTRODUCTION

This work examines the effect of limited asynchrony on three fundamental problems of distributed computation: The problem of mutual exclusion, that of readers and writers and the problem of symmetry breaking in a logical ring. The amount of asynchrony among local clocks of the various sites is limited as follows: We assume the distributed system to be *Archimedean* (*see* [17]). That is, the duration of a step of any process in any site is bounded above by r_{max} and below by r_{min} units of (absolute) time and the (absolute) time it takes for any message to be sent through a direct communication link is bounded above by d_{max} and below by d_{min}. Although processes are assumed to know the bounds r_{min}, r_{max}, d_{min} and d_{max}, they do not have access to any global clock showing the absolute time.

Any practical distributed system up to now follows the Archimedean time assumption. However, there is very little previous research investigating the gains in efficiency that distributed algorithms may have by exploiting such assumptions. The work of [17] presented an algorithm to elect a leader in an Archimedean ring of N processors with distinct names, by using only $O(N)$ messages, while it had been previously shown that $O(N \log N)$ messages are needed by may distributed algorithm in order to elect a leader in a ring, if unlimited asynchrony is assumed. [13, 14] used Archimedean asynchronism in their distributed algorithms for interprocess communication and resource allocation.

On the other hand, there is a considerable amount of research on distributed algorithms that work in a *synchronous* network. Most of those techniques depend crucially on the exact timing one can do in the absence of any asynchronism. It is therefore not all clear whether such techniques can be easily modified so they can apply to an environment of limited asynchrony. (*See*, for example, [4, 6, 7, 10, 16, 5] as a representative sample of work in synchronous distributed computation.)

We consider here three major problems of distributed computation: The first is the achievment of mutual exclusion in a complete network of sites. We show how the notion of Archimedean time can be exploited to get a message complexity which is below the lower bound for the same problem when unlimited asynchrony is assumed. We also provide an analysis of the time delay needed to achieve mutual exclusion. Our second problem is that of synchronizing readers and writers, both by preventive (i. e. mutual exclusion based) techniques and by optimistic techniques (each process is allowed to execute and, if the result is unsatisfactory, has to start again). We show that

the optimistic techniques are much improved (e. g. starvation is avoided) by a simple use of the knowledge of limited asynchrony. Finally, we consider the problem of electing a leader in an anonymous ring of processors (symmetry breaking). We show how to adopt the synchronous algorithm of [5] in order to get a solution for an Archimedean ring, which has the same message complexity and much better time complexity that the algorithm of [17]. It seems that modifying a synchronous distributed algorithm to work in an Archimedean environment is a general technique which optimizes both message and time complexity simultaneously.

In some of our algorithms there will be more than one processes at each site of the network. These synchronous processes, of a site may enter into competition for local critical resources, in which case known mechanisms (such as semaphores) are used to resolve any conflict between them.

2. MUTUAL EXCLUSION IN A COMPLETE NETWORK

2.1. The Ricart and Agrawala algorithm

The algorithm proposed by Ricart and Agrawala [15] was selected because it minimizes the number of messages necessary for mutual exclusion. The algorithm assumes the presence of a complete transport network free of errors. Each node has three processes local to it. They operate on a set of common variables, with a semaphore serializing access to them.

The three processes are (*a*) one which invokes mutual exclusion for the site (*b*) one which receives "request" type messages from other sites and (*c*) one which receives reply messages from other sites. When a site P_i wishes to enter its critical section, it generates a timestamp (as in [9]) and sends a message to all the other sites, of the "request" type, accompanied by this timestamp. When a site receives such a message, it may either reply favorably by sending back a "reply" type message straight away, or defer its response. A site that has received a reply message from every other site may enter its critical section. On leaving its critical section, a site sends any deferred reply messages to all sites awaiting such a reply. The decision to reply at once or to defer the reply is based on the following priority mechanism: if the site does not wish to enter the critical section it replies at once (favorably). Else, it compares its timestamp with that of the request and the older (smaller valued) timestamp wins (in case of a tie, the process with smaller identity wins).

The protocol described requires $2(n-1)$ messages per entry into a critical section: $n-1$ for requests and $n-1$ for favorable replies (*See* also [12] for a nice description of the algorithm of [15]).

2.2. The protocol for Archimedean networks

If we reverse the meaning of a reply, so that no reply during a maximum "waiting period" implies a favorable response, and if a message of the "delayed" type is explicitly sent to indicate an unfavorable reply, then the number of messages necessary for each entry into the critical section varies between $n-1$ and $3(n-1)$, because the sending of a delayed type message implies that a favorable reply will have to be sent later. (*See* also [15].)

Since the number of the "delayed" type messages depends on the number of processes really competing for the resource, we propose to reduce the amount of competition by introducing random waits every time a process enters a competition for the critical region. If the maximum size of the uniformly random wait is carefully selected, this technique will reduce competition in half without introducing deadlock or starvation.

2.2.1. *The modified protocol*

The protocol requires a certain number of declarations, local to each or the sites $P_1, \ldots, P_n$. For P_i these are:

```
var     osn : 0...+∞        /* (our sequence number)*/
        hsn : 0...+∞        /* (highest sequence number seen)*/
        nnr : 0...n-1       /* (number of negative replies)*/
      csreq : boolean       /* (true when site wants to
                               enter the critical section)*/
sleep,  priority: boolean
        repgiven: array [1...n] of boolean
        ppr: 0...n-1        /*(#of positive replies)*/
```

The variable repgiven [j] is true if P_i has sent a (temporary) negative reply to a request from P_j.

We now present the three subprocesses competing for these local variables. The access to the variables must be protected by an exclusion mechanism. We assume a fair scheduling of the three subprocesses.

For site P_i:

(a) *Process which invokes mutual exclusion for this node:*

csreq ← true;

osn ← hsn + 1;

nnr ← 0;

ppr ← 0;

sleep ← true

wait for x steps, selected uniformly randomly between [mindelay + predict*delaystep, mindelay + 2*predict*delaystep]

sleep ← false

for $j \neq i$, $j \in 1 \ldots$ n send (request, osn, i) to j; wait for window (i) steps /* count nnr*/ wait until ppr = nnr

```
⟨ CRIRICAL SECTION ⟩
      csreq ← false;
  for j = 1 to n do
    if repgiven[j] then
      begin
      repgiven[j] ← false;
      send (positive rep) to j
      end
    od
```

(b) *Process which receives (request, k, j) messages*

```
/* k is the sequence number of the requesting site, j is the site name making the request */
  on receipt of (request, k, j) do
    begin
        hsn ← max (hsn, k);
  priority ← csreq Λ (not (sleep)) Λ [(k > osn) V (k = osn Λ i < j)];
  if priority then
    begin
    send (negative reply) to j;
    repgiven [j] ← true
    end
 end
  od
```

(c) *Process which receives replies*

on receipt of (negative reply) do

nnr ← nnr + 1;

on receipt of (positive reply) do

ppr ← ppr + 1;

2.2.2. *The delay variables*

DEFINITION: Let $\gamma = r_{min}/r_{max}$ $(0 < \gamma \leqq 1)$.

DEFINITION: Let s be the number of actually competing sites (with csreq $\leftarrow$ true) at any particular time ($0 \leqq s \leqq n$).

(a) The value of predict (of an invocation of process a) is set equal to the value of nnr seen in the previous invocation of *process a.*

This estimate for "predict" is good only when the number s changes slowly with time (i. e. is of *bounded acceleration* see [14]). Otherwise we can use the estimate predict $= n$.

(b) window (i) is set equal to $(2\, d_{\max}/r_{\min} + c \,.\, r_{\max}/r_{\min})$ where c is approx equal to the number of steps of process (b). This quarantees that site i is going to collect all negative replies possible. This is so because $r_{\max}/r_{\min}$ steps of a process correspond to at least one step of any other process.

(c) mindelay is set to $c_1 \,.\, r_{\max}/r_{\min}$, *where* $c_1 = \#$of statements of process (a). That is, mindelay corresponds to the "best" case for i, when nnr $= 0$.

(d) delaystep is set to $c_1 \,.\, r_{\max}/r_{\min}$ (so that nnr $*$ delaystep corresponds "approximately" to the time delay that site i is going to suffer due to low priority).

2.2.3. *The performance of the Archimedean protocol*

LEMMA 1: *Due to the random shifts, the* mean value E (nnr) *of* nnr *is* $1/(1+\gamma) \,.\, s/2$. *Furthermore, for each* $\beta \in (0, 1)$, nnr $\leqq (1+\beta)$ E (nnr) *with probability* $\to 1$ *as* $n \to \infty$.

Proof: The random shift causes a process to sleep for a period of time which is at least γ and at most $2/\gamma$ of the active round of the process. Due to independent random shifts, the probability that a starting to compete process finds another particular process active, is then at most $1/(1+\gamma)$. Hence, on average, a competing process finds $1/(1+\gamma) \,.\, s$ of its competitors awake. Its timestamp is then "in the middle", again due to the random shifts. So, the mean value of nnr is $1/(1+\gamma) \,.\, (s/2)$. The rest of the lemma follows by applying a theorem on tails of Bernouilli trials. □

LEMMA 2: *The number of messages,* n_1, *required by the Archimedean protocol for mutual exclusion has mean value* $\bar{n}_1 = n - 1 + (1/(1+\gamma)) \,.\, (s/2)$. *The probability that* n_1 *exceeds* $(1+\beta)\bar{n}_1$ [*for any* $\beta \in (0, 1)$], *goes to zero as* $n \to \infty$.

Proof: By the protocols, the number of messages sent by any process is equal to $n - 1 +$ ppr $= n - 1 +$ nnr. The lemma then follows by Lemma 1. □

2.2.4. *The time spent before entering the critical region*

The length of time a process waits before entering its critical section depends on whether (or not) messages are received in the order in which they are transmitted.

DEFINITION: The *granting delay D* is the stretch of time beginning with the requesting node asking for the critical section and ending when that node enters its critical section.

As noted in [15] the worst case happens when the order of messages is not preserved, specifically when messages of the "reply" type overtake messages of the "request" type. In such a case D can be (in the original protocol) as bad as $(n(n+1)/2-1)\,r_{\max}$.

In our modified protocol, *a* node A of the lowest possible priority will change its priority as soon as a message of the request type arrives from another node. This will take time at most $d_{\max}$. Any other node may not enter its critical section more than twice in succession after that (once because its timestamp is older and once more because its site number is smaller). Therefore, the node A will wait at most $D \leqq 2\,d_{\max}+2(n-1)$. mindelay units of time before it is allowed to enter its critical region. Our remarks about the mean number of actually competing nodes leads to the conclusion that the mean value of D satisfies

$$\bar{D} \leqq 2\,d_{\max}+\left(\frac{1}{1+\gamma}\right)\left(\frac{s}{2}\right). \text{mindelay}.$$

2.2.5. *Conclusions and Remarks*

Our protocol exploits the Archimedean asynchrony in two ways: (*a*) by using a timeout to implicity detect a favorable reply and (*b*) by introducing random waits to reduce competition for the critical resource. Both its message and time complexity are below those of the Ricart and Agrawala protocol (which are shown to be optimal for unlimited asynchrony). Note that node failures do not affect our protocol (since no reply in a timeout is taken to be a favorable response any way). Of course failures happening after a negative reply and before the corresponding positive reply must be detected (by introducing another timeout, without the sending of any message).

The sequence numbers osn and hsn are theoretically unbounded but can be stored modulo M where $M \geqq 2n-1$, as noted in [15], since the maximum difference of any two sequence numbers at any time instance can be made to be, $\leqq n-1$. (In such a case, when making a comparison, the smaller number should be increased by M if the difference is n or more.)

3. THE PROBLEM OF READERS AND WRITERS

3.1. Introduction

Consider $n+1$ processes sharing a data set. Among these processes, n can read these data at the same time; these are the readers. The remaining process may change the value of the data; this is the writer. If the shared data set cannot be reduced to a single location and if the operations are only atomic with respect to one data item at a time, then a protocol is needed to guarantee the consistency of the whole data set. The readers and the writer must "mutually exclude each other" while all readers may read data simultaneously. The problem was first posed by [3].

Our protocol of Section 2 can be easily modified to solve the Readers-Writers problem: The modification is simply that "readers" never give a negative answer to a request of another reader. The writer follows the original protocol. Any other mutual exclusion algorithm can of course be used. This is the "traditional" approach to a solution of the problem.

3.2. The optimistic approach

The traditional solution to the Readers-Writers problem, based on mutual exclusion, has two unwanted characteristics (*a*) the fairness of the solution depends on the fairness of the mutual exclusion algorithm used and (*b*) processes are blocked for as long as the state of the system does not allow them to advances i. e. they are blocked due to "global" conditions.

Lamport in [8] proposed an optimistic technique in which each process is allowed to execute, and if the result is unsatisfactory, has to start again. The writers may always execute, giving new values to data. Readers, on the other hand, may check whether their read operation has overlapped with a single writing or with two or more write operations. In such a case, the reader has to start again. (Details in [8]).

Let W be the number of statements of the body of the infinite loop in the code of the writer process in the solution of [8]. Let R be the number of statements of the body of the infinite loop in the protocol of any reader process. Our proposal is again to exploit the Archimedean asynchrony by introducing random waits in both the readers and writers. The modification to the protocol is very easy: Each process has to wait for a random number of steps, selected uniformly (and independently for each process) from the interval $[(R+W)/\gamma, 2(R+W)/\gamma]$, where $\gamma = r_{\min}/r_{\max}$ before each access to the data. One can then prove that when a reader starts its code, the probability

p that the writer will be "in the middle" of its sleeping period will be at least $(R+W)/(2(R+W)/\gamma+W)$. This is so, because the random shifts make p to be equal to the ratio between the "sleeping period" of the writer and the sum of the "active" and "sleeping" periods. The duration of the active period is in $[W r_{min}, W r_{max}]$ units of absolute time and the duration of the sleeping period is in $[(R+W) r_{max}, 2(R+W) r_{max}^2/r_{min}]$.

The successive trials of a reader will independently succeed with probability p, hence the mean number of reader attempts before success is $1/p$ i. e. at most $(2(R+W)/\gamma+W)/(R+W) \leqq (2/\gamma)+1$.

Note that this simple idea eliminates the starvation problem that the solution of [8] has. Our protocol introduces waiting but it is a "local" pre-estimated wiating, very different from blocking due to global conditions. (*See* also [8] for the presentation of the optimistic protocol.)

4. SYMMETRY BREAKING IN AN ARCHIMEDEAN RING

4.1. Introduction

We consider here a unidirectional ring of n processors (i. e. messages are sent clockwise). The processors *are not assumed* to have unique identities and the ring follows the Archimedean restrictions on asynchrony. The problem is to device a randomised protocol so that a unique *leader* is elected in the ring. ([1] showed that no deterministic solution exists for the problem.) [7] presented an algorithm which uses $O(n)$ message bits and time on the average, under the assumptions of a *synchronous* network where all nodes start executing simultaneously. [5] presented a protocol which again use $O(n)$ bits and time, without any assumption on initiation time, again for synchronous networks. [17] presented an algorithm for solving a related problem (the minimum finding problem), in an *Archimedean* ring which uses *either* $O(n)$ messages *or* $O(n)$ time [but not both – in case of $O(n)$ messages the time becomes exponentially long]. In that solution the processor identities are assumed *distinct* from the beginning. (No symmetry breaking is needed.)

We propose here a protocol which solves the Symmetry Breaking problem in an *Archimedean* ring, by using *simultaneously* $O(n)$ messages and $O(n)$ time on the average. This is the first time the Symmetry Breaking problem is solved for Archimedean rings. Our technique was motivated by the synchronous protocol of [5].

4.2. A lower bound result

Since [1] proved that no deterministic solution exists for the symmetry breaking problem, most probabilistic algorithms that solve it work in the following way: They first have the processes to select names at random, independently of each other. Then, they run some "minimum finding" technique, with the additional burden that it must report whether multiple minima exist, in which case the protocol is run once more. If the ring is a synchronous one, then the reporting of multiple minimums can be done through a deterministic protocol. The following result says that *this is not the case* even with limited asynchrony, in the case of messages of length $0(\log n)$ bits.

LEMMA 3: *If the minimum and maximum message delays in an Archimedean ring are such that a message can make a full circle at the same time with making half a circle and messages have length* $0(\log n)$, then there can be no deterministic protocol *to discover whether there are multiple minimums among the (arbitrary) names of the processes.*

Proof: See figure 1. Consider two rings which are identical except for the fact that in ring A there is only one processor with name equal to the minimum value (1) and in ring B there are two such processors in symmetric positions (e. g. all other processors may have names equal to 2). If processor (1′) in B does exactly the same things as processor (1) and if the message delays are appropriately selected, then processor (1) can not distinguish whether the messages arriving to it come *from itself* (after making a full circle) or by the possible (1′). For the full argument, (1) and (1′) have to proceed in full synchrony, while the delays along the paths $1-1'$ and $1'-1$ have to be at least $nd_{\min}$. □

Therefore, any protocol that does symmetry breaking, even in an Archimedean network, *has to reside in probabilistic means to discover multiple minimums* (and therefore it may err with some small probability).

4.3. A protocol optimal in message and time complexity

The protocol is composed of a sequence of *phases.* In each phase, every node randomly selects an identity from $\{1, 2, \ldots, n\}$.

Initially all nodes are in a sleeping state. Any sleeping node can spontaneously become awake at any time, and start the first phase. As in [5] all nodes will be forced to "wake-up" even if they do not become awake on their own. Let $X=\{x_1, \ldots, x_m\}$ be the set of nodes which select 1. Now the processors run a *probabilistic* sub-protocol, to determine whether $|X|=\varnothing$, 1

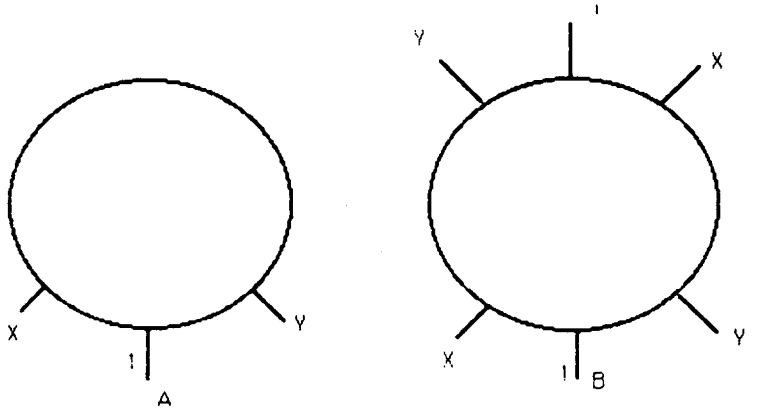

Figure 1.

or more. This sub-protocol goes through a sequence of k *stages* (k is a small predetermined constant e. g. $k=30$). At the end of the k stages one of the following two things may happen. (A) Each node has determined either that $|X|=0$ (with certainty) of $|X|>1$ (with certainty) in which case (we call it *situation A*) the processors have to select random names again and go through another phase. (B) Each node has determined that $|X|=1$, with probability $\geq 1-(3/4)^k$. In that case, one node becomes *elected* and the others become *defeated.* Note that the protocol here may elect more than one leaders with probability at most $(3/4)^k$, which can be made very small and is controllable by the implementer.

The way the sub-protocol runs is as is as follows: Each *candidate* node (one with identity equal to 1) uses a local timer to go through k *stages* of $n\lceil(d_{\max}+r_{\max})/r_{\min}\rceil$ steps each (so that each stage has actual duration $\geq n\,d_{\max}+r_{\max}$ i. e. at least the maximum time it takes a message to go a full circle). At the beginning of each stage, the candidate chooses to become "*holding*" or a "*forwarding*" candidate during that stage. In the beginning of the subprotocol a candidate sends a "claim" message. If a candidate receives just one claim during a "forwarding" period, it forwards the claim *at the end of the period only if the next period is again forwarding. If the next period is holding, the candidate does not forward (holds) the claim.*

If a candidate receives any claim during a holding period, then it decides that $|X|>1$ and causes the start of a new *phase.* If a candidate receives >1 claims during a forwarding period then it again decides that $|X|>1$ and causes the start of a new phase. If none of these happens during the k stages,

then the candidate becomes *elected* and causes for the other nodes to become *defeated.* The protocol is specified in detail in the *Appendix A.*

LEMMA 4: *Every node starts the execution of a phase within* $n(d_{\max}+r_{\max})$ *time units from the* beginning *of that phase.*

Proof: A phase begins by either a sleeping node being spontaneously *awake* (and all others sleeping) or by a node becoming a *next-timer.* In either case, in a full round, the corresponding messages will start everybody for the new phase.

LEMMA 5: *If* $|X|=1$ *in a phase, then the corresponding node is* elected *at the end of the phase and all others are defeated at end of phase.*

Proof: The node will become a *candidate.* The k stages will run without problems. □

LEMMA 6: *If a node becomes* elected *in a phase, then with probability at least* $1-(3/4)^k$ *the node is the only one elected and all others are defeated, and the algorithm ends, in that phase.*

Proof: Suppose that node A becomes a *candidate* at the beginning of a phase. Suppose node B is also a *candidate.* Consider first the case where the current period of A is holding and that its next period is going to be a holding one. Node B must necessarily flip its coin during the current period of A. If the flip of B comes to be "forwarding" then the claim of B will arrive at A either in the current period of A (in which case A will decide to go into a *next phase*) or *at most* during the next period of A (where, again A will decide to go into a next-phase). The above event has probability 1/4 and A detects the presence of B. (If the current period of A is forwarding, a similar argument can be done.) In fact, the only possibility for B to go undetected, is for A, B to be synchronized and go through the same sequence of selections. Hence, if $|X|>1$ then this is going to be detected, with probability at least $1-(3/4)^k$. (See *fig.* 2.) □

THEOREM 1: *Our protocol breaks symmetry in an Archimedean unidirectional ring using* $4ecnk$ bits and $4end_{\max}$ k *time units on the average, regardless of the initiation time, where* e *is the base of natural logarithms and* $c=O(1)$ *is the number of bits needed to distinguish between* 4 *distinct message types.*

If any phase, each node may send 2 *Wake-up* messages and one *next-time* message. If at least one node becomes candidate then each node will send or forward exactly one *claim* message for at most k *stages.* There is a constant number of message types, so each message uses a constant (c) number of

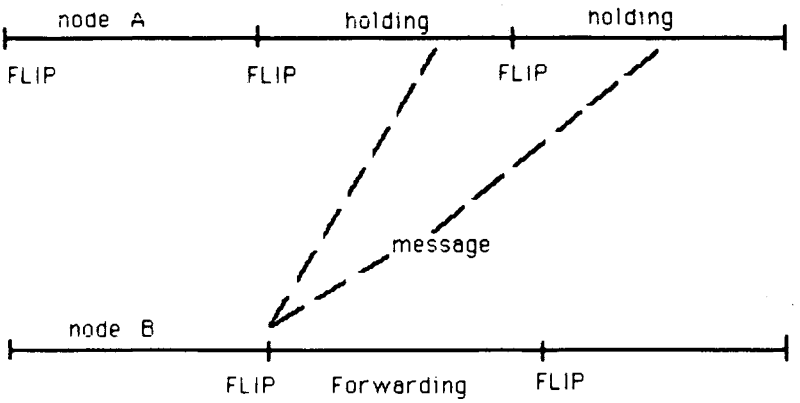

Figure 2. – Node A detects the presence of node B.

bits. So, each phase uses at most $4kcn$ bits. Within $4nd_{\max}k$ time units when the first node on a phase executed the wake-up routine, either a node is *elected* or a new phase is started. For any phase of random selections, the probability that exactly one node selects 1 is $(1-(1/n))^{n-1} > 1/e$. Thus, the expected # rounds until this occurs is less than $\sum_{j=0}^{\infty} (1/e)(1-1/e)^j(j+1) = e$. By Lemma 5 the protocol terminates when this happens. The protocol may terminate erroneously with probability at most $O((3/4)^k)$ which can be made arbitrarily small by selecting k to be e. g. 30 or 40. □

Note: The Archimedean assumption was *necessary* for timing the holding and forwarding periods. Also, from simulations that we did, it seems that our algorithm favors the *faster* candidate in being selected. Further work will quantify this.

5. CONCLUSION AND FURTHER WORK

We presented here strong evidence that Archimedean networks admit more efficient protocols than networks with no limits on the asynchrony. We did this for three fundamental problems of distributed computation, in each case providing a more efficient protocol than those in the literature. We also showed that Archimedean rings are less powerful than synchronous rings. Further work will examine the power of limited asynchrony through some impossibility theorems.

APPENDIX A

The symmetry breaking protocol

Rule 1 *A sleeping* node

1.1 It can become spondaneously *awake* and execute the Wake-up-1 routine.

1.2. If it receives a "wake-up" message it becomes *awake* and executes the Wake-up-1 routine.

Rule 2 An *awake* node

2.1. It ignores any received "wake-up" messages.

2.2. If it receives a "termination" it becomes *defeated* and passes the message on.

2.3. When its clock is equal to $n \lceil (d_{\max}+r_{\max})/r_{\min} \rceil$, if no claim is received and the number it selected is 1, it becomes a *candidate* and sends a "claim" message.

2.4. If the number selected is 1 and it receives a claim when its clock is $< n \lceil (d_{\max}+r_{\max})/r_{\min} \rceil$ then it becomes a *next-timer*, and sends a "next time" message.

2.5. If it receives a "next-time" message it becomes a *next-timer*, and sends a "next-time" message.

2.6. If the number selected is not 1 and no claim is received for $n(r_{\max}/r_{\min}+1) \lceil (d_{\max}+r_{\max})/r_{\min} \rceil$ steps, then the node becomes a *next-timer*, and sends a "next-time" message.

2.7. If always forwards the "claim" message it receives.

Rule 3 A *next-timer* node

3.1. If it receives a "wake-up" message it becomes *awake* and executes the wake-up-1 routine.

3.2. If it receives a "next-time" message for the first time, it passes it on. It ignores any subsequent "next-time" messages.

3.3. It generates a "wake-up" message.

Rule 4 A *candidate* node

4.1. Sets a *timer* $=0$ and a *counter* $j=0$ (The timer increases itself automatically by 1 at a rate equal to a step of the node).

4.2. Every time the timer $= n \lceil (d_{\max}+r_{\max})/r_{\min} \rceil$ and $j<k$ the timer is reset to zero, and $j \leftarrow j+1$.

4.3. Every time the timer is equal to zero, the *candidate* chooses (with probability 1/2) to become a "*holding*" or a "*forwarding*" candidate, except for the first time ($j=0$) is which the candidate chooses to be a forwarding one.

4.4. If the node receives a claim during a "holding" period, then it becomes a *next-timer* and generates a *next-time* message.

4.5. If the node receives >1 claims during a "forwarding" period, then it becomes a *next-timer* and generates a *next-time* message.

4.6. If the node receives only the claim during a forwarding period, and the buffer is empty, or if it receives no claims during the forwarding period and the buffer contains a claim, then (*a*) if its next period is again forwarding, it forwards the message on *exactly at the end of its current forwarding period*, or (*b*) if its next period is holding, then it puts the claim into a local *buffer* and does not forward it.

4.7. If the node receives a "next-time" message it becomes a *next-timer* and forwards the "next-time" message.

4.8. When $j=k$ (end of k stages) the node waits for an additional $n\,d_{\max}$ steps to get any messages in progress, "next-time" messages will cause it to proceed as in 4.7. Else, it become *elected* at the end of the period and sends a "termination" message.

*Wake-up-*1 *routine*

1. Choose a number at random in $\{1, \ldots, n\}$.
2. Set clock $=0$ and send a "wake-up" message.

REFERENCES

1. D. Angluin, *Local and Global Properties in Networks of processes*, Proc. 12th A.C.M. Symp. on Theory of Computing, April 1980, pp. 82-93.
2. C. Attiya, M. Snir and M. Warminth, *Computing on an Anonymous Ring*, Proc. 4th A.C.M. Symp. on Principles of Distributed Computing, Aug. 1985, pp. 196-204.
3. P. J. Courtois, F. Heymans and D. L. Parnas, *Concurrent Control with Readers and Writers*, C.A.C.M., Vol. 14, No. 10, pp. 667-668.
4. G. Frederickson and N. Lynch, *The Impact of Synchronous Communication on the Problem of Electing a Leader in a Ring*, Proc. 16th A.C.M. Symp. on Theory of Computing, April 1984, pp. 493-503.
5. G. Frederickson and N. Santoro, *Breaking Symmetry in Synchronous Networks*, V.L.S.I. Algorithms and Architectures, AWOC 1986, Lecture Notes in Computer Science, No. 227, Springer Verlag, pp. 26-33.

6. E. GAFNI, *Improvements in the Time Complexity of two Message-optimal Election Algorithms*, Proc. 4th A.C.M. Symp. on Principles of Distributed Computing, Aug. 1985, pp. 175-185.
7. A. ITAI and M. RODEH, *Symmetry Breaking in Distributive Networks*, Proc. 22nd I.E.E.E. Symp. on Foundations of Computer Science, Oct. 1981, pp. 150-158.
8. L. LAMPORT, *Concurrent Reading and Writing*, C.A.C.M., Vol. 20, No. 11, 1977, pp. 806-811.
9. L. LAMPORT, *Time Clocks and the Ordering of Events in a Distributed System*, C.A.C.M., Vol. 21, No. 7, 1978, pp. 558-565.
10. J. VAN LEEUWEN, N. SANTORO, J. URRUTIA and S. ZAKS, *Guessing Games and Distributed Computations in Synchronous Networks*, 14th I.C.A.L.P., L.N.C.S., No. 267, 1987, pp. 347-356, Springer-Verlag.
11. M. OVERMARS and N. SANTORO, *An Improved Election Algorithm for Synchronous Rings*, preliminary draft, Carleton University, March 1986.
12. M. RAYNAL, *Algorithms for Mutual Exclusion*, The M.I.T. Press, 1986.
13. J. REIF and P. SPIRAKIS, *Real Time Synchronization of Interprocess Communication*, A.C.M. Transactions of Programming Languages and Systems, April 1984.
14. J. REIF and P. SPIRAKIS, *Unbounded Speed Variability in Distributed Systems*, S.I.A.M. Journal of Computing, February 1985.
15. G. RICART and A. AGRAWALA, *An Optimal Algorithm for Mutual Exclusion in Computer Networks*, C.A.C.M., Vol. 24, No. 1, Jan., 1981.
16. N. SANTORO and D. ROTEM, *On the Complexity of Distributed Elections is synchronous graphs*, Proc. 11th Int. Workshop on Graphtheoretic Concepts in Computer Science, June 1985, pp. 337-346.
17. P. VITÁNYI, *Distributed Elections in an Archimedean Ring of Processors*, Proc. 16th A.C.M. Symp. on Theory of Computing, April 1984, pp. 542-547.