Using atomic broadcast to implement *a posteriori* agreement for clock synchronization*

L. Rodrigues ler@inesc.pt
P. Veríssimo paulov@inesc.pt
A. Casimiro casim@inesc.pt

Technical University of Lisboa - IST - INESC †

Abstract

In a recent paper we presented a new clock synchronization algorithm, dubbed *a posteriori agreement*, a variant of the convergence non-averaging technique. By exploiting the characteristics of broadcast networks, the effect of message delivery delay variance is largely reduced. In consequence, the precision achieved by the algorithm is drastically improved. Accuracy preservation is near to optimal.

In this paper we present a particular materialization of this algorithm, implemented as a time service of the *x*AMp group communications system. The algorithm was implemented using some of the primitives offered by *x*AMp, which simplified the work and stressed its advantages. The paper also presents performance results for this implementation obtained on two different infrastructures. Timings validate the design choices and clearly show that our algorithm is able to provide improved precision without compromising accuracy and reliability.

1 Introduction

A solution to the problem of maintaining a global timebase consists of creating a virtual clock in each node of the distributed system, which is synchronized with all other virtual clocks using a clock synchronization algorithm. Usually, each node must read remote virtual clocks regularly or send a message at a previously agreed instant. When using software based algorithms[9,14] message delivery times cannot be computed with exactitude due to the variability of network message delivery delays, thus affecting virtual clocks precision. When using hardware [4,7] or hybrid schemes[5,11] the variability of network access delays is minimal but these solutions are expensive and hard to implement. Probabilistic or statistical solutions to damp the effect of the variance have also been proposed [3,2]. In fact, a major limitation of all known software clock synchronization algorithms designed for arbitrary networks, is that precision is limited either by the variance of the message delivery delay [8], or by its upper bound [14].

In a recent paper [15], we presented an algorithm, named *a posteriori agreement*, which uses broadcast facilities inherent to local area networks to avoid the influence of the network access delay variability on the precision of virtual clocks. The general algorithm is communication and agreement protocol independent, i.e., the choice of different communication infrastructures and agreement protocols would lead to different implementations of the algorithm. We now present an implementation of the *a posteriori* agreement algorithm as a time service, provided together with a group communications package: the *x*AMp [12]. The paper demonstrates that it is possible to efficiently take advantage of the primitives provided by a group oriented communications service. In particular, using the *x*AMp membership services, the integration of new processes becomes very simple and transparent to the user: there is only one primitive both to initiate the synchronization algorithm and join its execution. Another advantage of using *x*AMp is the dynamic adaptation of the algorithm's fault tolerance degree accordingly to process membership changes.

The paper is organized as follows: For self containment, an introduction to clock synchronization is given in section 2 and an overview of the *a posteriori* agreement algorithm is presented in section 3. The

*A version of this report will be published in the Proceedings of the 12th Symposium On Reliable Distributed Systems, Oct. 6-8, 1993, Princeton, New Jersey,

†Instituto de Engenharia de Sistemas e Computadores, R. Alves Redol, 9 - 6º - 1000 Lisboa - Portugal, Tel.+351-1-3100000. This work was partially supported by the CEC, through Esprit Projects 1226 (DELTA-4) and BR 6360 (Broadcast), and by JNICT through project Codicom.

xAMp implementation of the algorithm is presented in section 4. Next, initialization and integration are presented in section 5, along with the dynamic protocol adaptation to membership changes. An optimized version of the synchronization algorithm using a specialized atomic multicast synchronization protocol is described in section 6. Performance results are presented in section 7 and final concluding remarks appear on section 8.

2 The clock synchronization problem

The goal of clock synchronization is to establish a global timebase in a distributed system composed of a set of processes, $\mathcal{P}$, which can interact exclusively by message passing. Processes can only observe time through a *clock*. One commonly used solution to achieve this goal is to provide each processor, k, in the distributed system ($k \in \mathcal{P}$) with an imperfect physical clock pc_k (notation closely follows that of [13]). The clock at a correct processor k can then be viewed as implementing, in hardware, an increasing, continuous[1] function pc_k that maps (non-observable) real time[2] t to a clock time $pc_k(t)$. Through a clock synchronization algorithm it is possible to derive, from the physical clock at each node k, a virtual clock vc_k satisfying *precision*, *rate*, *envelope rate*, and *accuracy*. These properties are formally presented in table 1.

Precision δ_v characterizes how closely virtual clocks are synchronized to each other, ρ_v is the drift rate of virtual clocks, ρ_α is the long term drift rate of virtual clocks. Accuracy, α_v characterizes how closely virtual clocks are synchronized to real time. Due to the nonzero drift rate of physical clocks, accuracy cannot be ensured without some external source of real time. In the context of internal synchronization, a good algorithm should maintain clocks as close as possible to the best real time source available, which may be one of the correct clocks in the system. In that sense, minimizing[3] ρ_v and ρ_α, should *preserve* accuracy, and that term will be used in this paper when informally discussing these attributes.

Since physical hardware clocks can be permanently drifting from each other, virtual clocks must be re-synchronized from time to time. A clock synchronization algorithm should then be able to: (i) generate a periodic re-synchronization event. The time interval between successive synchronizations is called the re-synchronization interval, denoted T. (ii) provide each correct processor with a value to adjust the virtual clocks in such a way that *precision* and *rate* hold. The clock adjustment can be applied instantaneously or spread over a time interval. In both techniques, for the sake of convenience, the adjustment is usually modeled by the start of a new virtual clock upon each re-synchronization event.

The computation of the adjustment can be modeled by the evaluation of a *convergence function* [13]. The *precision enhancement property* specifies the best precision guaranteed after any two clock value evaluations at different processors. The worst-case clock precision, δ_v, is obtained by adding the term due to the convergence function to the imprecision generated by the drift between clocks during the re-synchronization interval T. However, since the drift, ρ_p, is typically of the order of $10^{-6}s$, the precision enhancement property of the convergence function is the relevant factor. Next section describes how the *a posteriori* agreement algorithm uses the properties of broadcast networks to implement a highly precise convergence function.

PC - Physical clocks; VC - Virtual clocks
(for some positive constants μ_p and ρ_p, $\forall_{k,l} \in \mathcal{P}$)

PC1- Initial value, $pc_k(0)$:
$0 \leq pc_k(0) \leq \mu_p$

PC2- Rate, ρ_p:
$0 \leq 1 - \rho_p \leq \frac{pc_k(t_2) - pc_k(t_1)}{t_2 - t_1} \leq 1 + \rho_p \text{for} \quad 0 \leq t_1 < t_2$

VC1 - Precision, δ_v:
$|vc_k(t) - vc_l(t)| \leq \delta_v$, for $0 \leq t$

VC2 -Rate, ρ_v:
$1 - \rho_v \leq \frac{vc_k(t_2) - vc_k(t_1)}{t_2 - t_1} \leq 1 + \rho_v$ for $0 \leq t_1 < t_2$

VC3 - Envelope Rate, ρ_α:
$1 - \rho_\alpha \leq \frac{vc_k(t) - vc_k(0)}{t} \leq 1 + \rho_\alpha$ for $0 \leq t$

VC4 - Accuracy, α_v:
$|vc_k(t) - t| \leq \alpha_v$ for $0 \leq t$

Table 1: Summary of Clock Properties.

[1] It is known that digital clocks have a finite granularity and increase by ticks. This notion is of utmost relevance to derive ordering and synchronism properties of real-time systems [6], i.e. in using clocks. However, for sake of clarity, we chose to simplify our expressions in this matter.

[2] In an assumed Newtonian time frame.

[3] In any case, limited to ρ_p [14].

- *Broadcast:* Nodes receiving an uncorrupted message transmission, receive the same message.
- *Error detection:* Nodes detect any corruption done by the network in a locally received message and discard it.
- *Bounded Omissions:* In a network with N nodes, in a known interval, corresponding to a series of M unordered message transmissions, omission failures may occur in at most f_o transmissions.
- *Bounded Transmission Delay:* The time between any broadcast send request and the delivery at those nodes that receive the message, is bounded by two known constants $[\Gamma^{min}, \Gamma^{max}]$. The variance in the message delivery delay, $\Delta\Gamma$, is then: $\Delta\Gamma = \Gamma^{max} - \Gamma^{min}$.
- *Tightness:* Nodes receiving an uncorrupted message transmission, receive it at real time values that differ, at most, by a known small constant $\Delta\Gamma_{tight}$.

Table 2: Summary of Network Properties.

3 The *a posteriori* agreement algorithm

The *a posteriori* agreement algorithm is a new variant of the convergence non-averaging technique. The algorithm is highly precise and preserves close-to-optimal rate drift without the need for hardware support. The properties of broadcast networks are used to drastically attenuate the traditional limitation imposed by message delivery delay variance on the obtained precision. The algorithm was formally presented in [15]. This section intends to give an informal overview of its basic principles.

To start with, we present our assumptions about the system: (i) *clocks* may have arbitrary failures (eg. provide erroneous or conflicting values when read); (ii) clock server *processes* (the ones running the protocol over the network) exhibit failures from crash to uncontrolled omission or timing failures; (iii) the maximum number of clock-process pairs with failures during a protocol execution is f_p.

Fundamental to our technique, is the use of broadcast network properties to improve the precision of clock synchronization. In particular, we are targeting networks with the properties presented in table 2. A study on the influence of network timing properties on clock synchronization presented in [6] will help explaining our method. It decomposes a message delivery delay in the following terms: *send time*, Γ_{send}, to assemble the message and issue the send request; *access time*, Γ_{access}, for the sender to access the channel; *propagation time*, Γ_{prp}, for the channel to copy the message to all recipient links[4]; and *receive time*, Γ_{rec}, to process the message at the receiver. These contributions are illustrated in figure 1. The precision of an algorithm is influenced by the variances, which together make up the message delivery delay variance. We dubbed the influence on the clock precision of the send time variance, $\Delta\Gamma_{send}$, the *send error*. A similar terminology is used for the influence on the clock precision of the remaining terms.

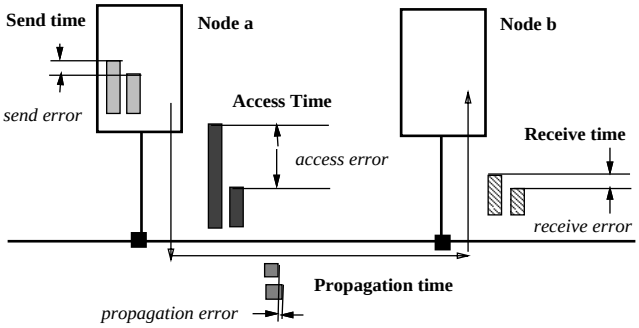

Figure 1: Network timing properties.

We now analyze the relative importance of each of these terms when local broadcast networks are used. The propagation error is very small: in an Ethernet, for example, the maximum difference between the times of physical reception of a message is less than 20 μs. The receive error, $\Delta\Gamma_{rec}$, cannot be disregarded but it is usually small and may be improved. On the contrary, the variance of the access time, $\Delta\Gamma_{access}$, is hardly controlled and strongly depends on variations of the network load and other operating factors (eg. collisions in Ethernet, token rotation time in a token-passing LAN). It is the dominant term in message delivery delay variance, $\Delta\Gamma$, and given that, $\Delta\Gamma_{tight} = \Delta\Gamma_{prp} + \Delta\Gamma_{rec}$, a relevant timing property of architectures based on local broadcast networks is formulated the following way:

$$\Delta\Gamma_{tight} \ll \Delta\Gamma$$

The reader will now note three attributes of a faultless broadcast which are crucial for understanding our algorithm: (i) a successfully transmitted message $\langle m \rangle$ arrives virtually at the same time on all nodes, the difference corresponding to the propagation error; (ii) processing times of $\langle m \rangle$ reception at any two nodes

[4] This is the physical propagation time, dependent on the variable distance between nodes.

For every processor m.

```
00 start^i = 0 ;
01 when vc_m^{i-1}(t) = iT do
02       bcast ⟨start, i, m⟩ ;
03 when message ⟨start, i, n⟩ received from processor n
04       t_m^{i,n} = reception (real) time of ⟨start, i, n⟩
05       cc_m^{i,n}(t_m^{i,n}) = iT; ; start^i =start^i + 1 ;
06       if (start^i > f_p) then bcast ⟨valid, i, n, vc_m^{i-1}(t_m^{i,n})⟩
07       else bcast ⟨notsure, i, n, vc_m^{i-1}(t_m^{i,n})⟩ ;
08 // acknowledges must be collected and an agreement must
09 // be reached to select an unique and associated adjustment
10 when cc^{i,n} and Δ_cc^{i,n} agreed :
11       vc_m^i = cc_m^{i,n} + Δ_cc^{i,n};
```

Figure 2: The *a posteriori* agreement (sketch)

vary at most by the receive error; (iii) thus, in response to a successful broadcast, all nodes are able to take an action within $\Delta\Gamma_{tight}$ of each other.

An aim of the *a posteriori agreement* technique is to improve precision by making the clock synchronization algorithm depend on $\Delta\Gamma_{tight}$ (instead of $\Delta\Gamma$ or Γ^{max}). To achieve this, the protocol exploits the listed attributes of a fault-less broadcast as follows (see figure 2): synchronization starts with each processor disseminating a $\langle start \rangle$ message at a pre-agreed instant on its clock (line 1); after a series of broadcast exchanges, each tentatively initiating a new virtual clock (line 5), an agreement is obtained both on a broadcast yielding high precision, and on the clock to synchronize from in order to yield the best accuracy possible (line 11). It was thus dubbed *a posteriori agreement.*

More specifically, if message $\langle start \rangle$, addressed to all including the sending node, meant: "Let us synchronize! I think the time is iT. What time is it on your clocks?", one can obtain the following:

- *precision enhancement:* in response to $\langle start, i, m \rangle$, a new virtual clock is tentatively initiated everywhere with iT, at the same physical time more or less an error equal to $\Delta\Gamma_{tight}$ (line 5);
- *accuracy preservation:* also at that time, the clock of each recipient can be read and disseminated, in an acknowledgment message (line 6); thus, it is possible to agree on the best clock in terms of accuracy (eg. the median) and to compute its difference to iT (Δ_{cc}), to adjust accuracy of the tentative clocks.

This happens every time a process broadcasts $\langle start \rangle$ with success. There will be a number of tentative virtual clocks launched, and an agreement protocol is run (a posteriori) to disseminate the chosen one, together with the adjustment, through the clock processes. To avoid starting a re-synchronization in response to a faulty processor, at least $f_p + 1$ $\langle start \rangle$ messages must be received by a majority of members to reach agreement. Thus, at least $2f_p + 1$ processors are required to tolerate f_p clock/processors failures.

The algorithm becomes more clear with an example (see figure 3): at the beginning of the synchronizations virtual clocks exhibit a *'0:04'* precision and *'0:03'* accuracy (units are abstract) as vc_2^{i-1} *('5:01')* = *'5:02'* and vc_3^{i-1} *('5:01')* = *'4:58'* (left hand values); processor 1 sends a $\langle start \rangle$ message when its clock is at *'5:00'*, which is received by all with a $\Delta\Gamma_{tight}$ = *'0:01'*; the candidate clocks are started with a dummy value of *'5:00'* (right hand values); the virtual clocks of all processors are read and, during the agreement the median value (*'5:10'*) is chosen to adjust candidate clocks: the computed adjustment is $\Delta_{cc} = +$*'0:10'*; when agreement is reached candidate clocks are adjusted; at the end of the agreement processors exhibit a precision of *'0:01'* (introduced by $\Delta\Gamma_{tight}$) and an accuracy of *'0:02'*; finally, continuous synchronization is used to spread the difference between vc^{i-1} and vc^i [5].

For clarity, in the example we have disregarded the clock drift during protocol execution. However, the precision obtained at the end of execution will also depend on the clock rate drift, as with other agreement-based synchronization protocols. The time required to run the agreement protocol influences precision by a factor of ρ_p but its effect can be neglected, given that ρ_p is very small. The receive error may be practically canceled with co-processors and interrupt treatment, provided that there is an upper bound for message reception interrupt service latency and that bound is small. Then it may be intuitively said that precision is optimal, in the sense that it cannot be better than the difference between physical reception times of a message at any two nodes (propagation error).

Accuracy *preservation* — which for internal synchronization means following a correct hardware clock, i.e. respecting an envelope rate — on the other hand, cannot be optimal in the terminology of Srikanth & Toueg [14], but is very close to it. In fact, the algorithm collects the values of all clocks at the time the candidate clock was started, and chooses a value assured to be correct by discarding all possibly erroneous clocks (for instance, by using the median) in order to adjust the selected candidate clock. The worst of the correct clocks forms a bound of the optimal rate

[5] Although the new instantaneous clock may lag the previous clock, continous synchronization guarantees the monotonicity of the virtual clocks.

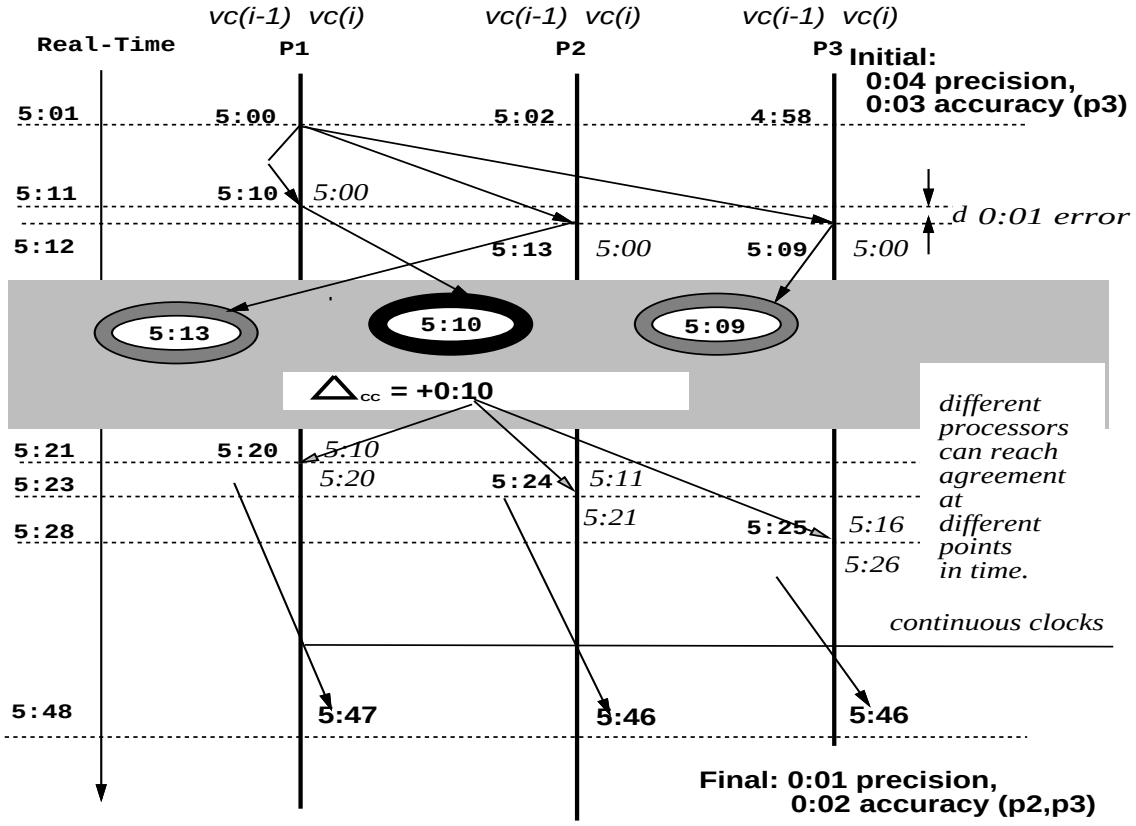

Figure 3: *a posteriori* agreement execution.

drift envelope. Our algorithm will, in worst-case, synchronize by that clock, deviating to the outside of the envelope at most by the measure of $\Delta\Gamma_{tight}$, which, as just discussed, can be made very small (see [15] for details).

Our algorithm, like consistency based algorithms [8], requires the execution of an agreement protocol. However, the algorithm does not require any particular agreement protocol (as long as it exhibits bounded termination). Since most fault-tolerant distributed systems are LAN-based and implement some form of agreement protocol, our algorithm can be easily integrated in such architectures. This paper presents a case-study of this approach, by describing the integration of *a posteriori* clock synchronization in xAMp communications service.

4 xAMp support

The xAMp [12] is a highly versatile group communications service aimed at supporting the development of distributed applications, with different dependability, functionality, and performance requirements. These range from unreliable and non-ordered to atomic multicast, and are enhanced by efficient group addressing and management support. The basic protocols are synchronous, *clock-less* and designed to be used over broadcast local-area networks, and portable to a number of them. The functionality provided yields a reasonably complete solution to the problem of reliable group communication. The use of xAMp to support the implementation of the *a posteriori* agreement algorithm has many advantages:

- Using xAMp, the implementation can benefit from the support to manage processor groups in order to maintain information about the processors that are participating in clock synchronization.
- Additionally, xAMp communication primitives can be used to generate simultaneous broadcasts and to implement the agreement algorithm necessary to select the most precise clock.

The *membership* service supports the dynamic modification of groups, providing the primitives that allow processors to join or leave a group. Furthermore,

when a group member fails, the event is detected and indicated to the remaining group members. Naturally, we have created a group dedicated to clock synchronization. To participate in the clock synchronization, one processor has simply to join the synchronization group. The *multicast* service allows a message to be delivered transparently to a group of processes. In particular, xAMp is able to mask the network omission degree, alleviating the user of explicit concern with omission faults (f_o). Several qualities of service (QOS) are available in xAMp making possible to choose the ones that are more suitable to the selected application.

To implement the *a posteriori* agreement algorithm, we have used the *atomic* QOS and the xAMp's reliable datagram service, dubbed *transmit-with-response* (or simply *tr-w-resp*). The *tr-w-resp* service can be seen as an interface to the underlying unreliable multicast (network-level) service but with automatic collection of responses, message retransmission when omissions occur, and membership maintenance. The atomic QOS is more complex and assures *unanimity* and *total order respecting causality*. An important property of all qualities of service provided by xAMp is *synchronism*, i.e. xAMp guarantees that the service is provided within known time bounds. This last property is essential to us, since the precision of virtual clocks also depends on the time necessary to execute the agreement protocol.

Clock synchronization can be implemented using xAMp as follows (see figure 4). At each re-synchronization instant, the members of the synchronization group try to start a new candidate clock by sending a $\langle start \rangle$ message using the *tr-w-resp* service (line 25). Since *tr-w-resp* implements automatic retransmission, the generation of a simultaneous broadcast is assured if the sender does not fail. Two types of acknowledgments are expected for the start message (line 34). At each member, the first f_p $\langle start \rangle$ messages received are considered not safe, as they may have been generated by faulty senders. Thus, a $\langle notsure \rangle$ acknowledgment is returned to the sender. The remaining messages are acknowledged with a response of type $\langle valid \rangle$, meaning that at least f_p other messages have already been received. In any case, a *candidate* clock is started for each $\langle start \rangle$ message received (line 33). Note that in this implementation, acknowledgments are sent only to the sender of the $\langle start \rangle$ message, who will verify whether it created a valid candidate clock. Note also that if a $\langle start \rangle$ message is retransmitted, the associated clock is just started again.

The sender waits for the termination of the *tr-w-resp* procedure. Under our failure assumptions, if the

sender part (for each member m)

```
20 // let P be the set of correct processors
21 ( P is automatically updated by xAMp)
22 // let R be a bag of acknowledgments
23
24 when vc_m^{i-1}(t) = iT do
25    tr-w-resp ( ⟨ start, i, m ⟩, R );
26    if (∃r ∈ R : r is of type ⟨valid⟩ ) then
27       select Δ_cc^{i,m} from R;
28       send ⟨ accept, i, m, Δ_cc^{i,m} ⟩ using atomic QOS ;
```

receiver part (for each member n)

```
30 start^i := 0; vc_n^i := none;
31 when ⟨ start, i, p ⟩ message received from
32 processor m at real time t_n^{i,m} do
33    start candidate clock cc_n^{i,m}; start^i :=start^i + 1;
34    if (start^i ≤ f_p) then send ⟨ notsure, vc_n^{i-1}(t_n^{i,m}) ⟩ to m
35    else send ⟨ valid, vc_n^{i-1}(t_n^{i,m}) ⟩ to m ;
36 when ⟨ accept, i, p, Δ_cc^{i,p} ⟩ message received from
37 processor p and vc_n^i = none do
38    vc_n^i = cc_n^{i,p} + Δ_cc^{i,p}; send ⟨ ok ⟩ to p ;
39 when ⟨ accept, i, q, Δ_cc^{i,q} ⟩ message received from
40 processor q and vc_n^i ≠ none do
41    send ⟨ ok ⟩ to q; //just acknowledge reception
```

Figure 4: *a posteriori* agreement using xAMp

sender does not fail before *tr-w-resp* completes it will receive an acknowledgment from every correct member of the group. The incorrect members are automatically excluded by xAMp. If at least a process responds with $\langle valid \rangle$, the sender can safely assume that it started a timely candidate clock (line 26). It then uses the information provided in the acknowledgments to compute an appropriate adjustment for its candidate clock such that accuracy is preserved (using the algorithm summarized in the previous section) and broadcasts an *accept* instruction using the *atomic* QOS (simultaneously it disseminates the appropriate adjustment for that candidate clock). Usually, several candidate clocks will be started and several *accept* messages will be received by every member. Since *accept* messages are sent through atomic QOS (line 28), they will be received by all members in the same order. The first *accept* message delivered selects the candidate clock to be used during the next synchronization interval (line 36); all other candidate clocks are simply discarded. Note that if a process crashes before sending an accept, its candidate clock will simply be discarded as soon as an accept arrives from a correct process.

5 Dynamic fault-tolerance adaptation, initialization and integration

An important advantage of using *x*AMp services is that the membership information provided can be used to verify if the system maintains the required fault-tolerance degree. When desired, the clock synchronization service can be requested to dynamically adapt to membership changes. Thus, the user may select one of two clock synchronization modes, namely the *hard-clock mode* and the *soft-clock mode*:

- In the *hard-clock mode*, the user specifies a desired fault-tolerance degree, i.e., the number of clock and/or processor faults that need to be tolerated. As stated in section 3, the minimum number of nodes to tolerate f_p faults is $2f_p + 1$. The service ensures that the virtual clocks are only running while the minimum number of nodes is available in the system.

- In the *soft-clock mode*, continuity of service is provided for any number of clocks in the system. When the membership changes, in particular if some nodes are lost, the protocol dynamically adapts and the synchronization service reports to the user that the service is running with a lower fault-tolerance degree.

Both modes are particular cases of a more generic parameterization supported by the *x*AMp clock service. When the clock service is started, the desired tolerated faults are specified as an interval $[f_{min}, f_{max}]$. In the hard-clock mode, we have $f_{min} > 0$. In the soft-clock case, we simply have $f_{min} = 0$. Upon system initialization, the clock synchronization service is only started when at least $2f_{min} + 1$ nodes are present (in the soft-clock mode the service can be started as soon as the first node joins the algorithm). If during system evolution the number of processors becomes lower than f_{min}, the service is halted and an exception is generated. The parameter f_{max} is used to save system resources. If more than $2f_{max} + 1$ nodes are present, the extra members of the clock-synchronization group do not need to generate redundant start messages (although they still have to acknowledge incoming start requests).

When the number of processors in the system, N, lies within fault-tolerance interval, i.e., $N \in [2f_{min} + 1, 2f_{max} + 1]$, the protocol runs in adaptative mode. When the number of processors is in the range, $[2f_{min}+1, 2(f_{min}+1)+1[$, the algorithm tolerates f_{min} faults, in the range $[2(f_{min} + 1) + 1, 2(f_{min} + 2) + 1[$, it tolerates $f_{min} + 1$ faults, and so on. Whenever the interval changes, the user is informed of the new fault-tolerance degree.

The membership information is also extremely useful for system initialization. Every time a process joins the clock synchronization the new membership is provided to all members, including to the new member. From the membership information, the newcomer may infer the state of the group. Let N be the number of group members after a join; If $N < 2f_{min} + 1$, there is still no quorum[6] to start the synchronization service and the newcomer simply awaits for other processors to join the service. If $N = 2f_{min} + 1$, the newcomer has just formed the required quorum, thus all members will start the clock service by broadcasting $\langle start \rangle$ messages for the first synchronization round.

Finally, if $N > 2f_{min}+1$, the clock synchronization service is already running and the newcomer just waits for the next re-synchronization round to catch up with the running members. However, there is still a problem to be solved in this case: the new process may join the synchronization group during a re-synchronization round and thus miss some of the *start* or *accept* messages. The membership change service has two important properties that we use in this case: (i) joins and leaves are totally ordered with respect to atomic messages; (ii) group dependent data can be transferred to the joining process during the join operation. Using these two mechanisms, integration of joining processes is performed in the following way: when a processor joins the synchronization group it obtains the current round, i. It then waits for an *accept* for round $i+1$. If there is no local matching candidate clock for the first round $i + 1$ *accept*, this means that the process has joined between the transmission of the *start* message and of the corresponding *accept*: it must then wait for a round $i + 2$ accept. Otherwise it can synchronize immediately on round $i + 1$.

6 A specialized protocol

In section 4, we have described an implementation of the *a posteriori* clock synchronization using the *x*AMp user interface. In this section we show how an even simpler and more efficient implementation of our algorithm can be obtained by developing a *specialized atomic multicast synchronization protocol*. This specialized protocol was designed using the experience obtained by developing *x*AMp. In particular, it closely follows the implementation of *x*AMp atomic QOS.

[6] In the hard-clock mode.

sender part

```
50 when user request to send ⟨m⟩ do
51    tr-w-resp (⟨m⟩, R);
52    if (∀r ∈ R, r is ⟨ok_m⟩) then tr-w-resp (⟨acc_m⟩, R);
53                           else tr-w-resp (⟨rej_m⟩, R);
```

receiver part

```
60 when message ⟨m⟩ received from processor p do
61    remove ⟨m⟩ from Q;
62    add ⟨m⟩ to Q; start wdTimer_⟨m⟩;
63    if ( I am accessible for ⟨m⟩ ) then send (⟨ok_m⟩ );
64                                  else send ( ⟨nok_m⟩ );
65 when message ⟨acc_m⟩ received from processor p do
66    stop wdTimer_⟨m⟩; send ( ⟨ok_acc⟩ ); accepted_⟨m⟩ := true;
67 when message ⟨rej_m⟩ received from processor p do
68    stop wdTimer_⟨m⟩; send ( ⟨ok_rej⟩ ); remove ⟨m⟩ from Q;
69 when wdTimer_⟨m⟩ expires do
70    call recovery function (accept or reject will be received)
71 when ⟨m⟩ is on top of Q and accepted_⟨m⟩ do
72    deliver ⟨m⟩;
```

Figure 5: two-phase accept (used in atomic QOS)

The implementation described in section 4 runs in two phases: in the first phase, messages are broadcast to generate a simultaneous event; in the second phase, an agreement is run by atomically broadcasting validation messages (the *accept* decision). The reason to use an ordered broadcast quality of service in the second phase is that the message ordering is exploited to implement the agreement. Since all *accept* messages are received in the same order by all members, the first message can be used to unanimously select the candidate clock. However, the careful reader will notice that the first protocol phase can be immediately used to order ⟨*start*⟩ messages, making possible to implement a cheaper agreement protocol. In fact, the underlying broadcast medium is a natural sequencer of messages: if message ⟨*a*⟩ crosses the network before message ⟨*b*⟩, ⟨*a*⟩ will be received before ⟨*b*⟩ everywhere. Not surprisingly, this technique is already used in the *x*AMp to implement the *atomic* QOS! Thus, before developing a new protocol, we will take a closer look at *x*AMp to see how the atomic QOS works.

The operation of the atomic protocol is depicted in figure 5. It consists of a two-phase accept protocol that resembles a commit protocol where the *sender* coordinates the protocol: it sends a message, implicitly *querying* about the possibility of its acceptance (line 51), to which recipients reply (dissemination phase). When messages are received, they are inserted in a receive queue by the order they have crossed the network (line 62). If a message is retransmitted, it is removed and inserted again, to respect its new order (line 61). In the second phase (decision phase), the sender checks whether *responses* are all affirmative, in which case it issues an *accept* (line 52) – or *reject* (line 53), otherwise. To ensure the reception of the decision, by all correct recipients, the *accept* and *reject* frames are also sent using the *tr-w-resp* procedure. If the sender fails before successfully disseminating the accept or reject message, this is detected by the recipients through the wait-decision timer (*wdTimer*, line 69). In this case, a dedicated recovery function is run to terminate the protocol (line 70). The function ensures that the same decision (an accept or a reject) is delivered to all recipients. The recovery algorithm was described in detail in [16] and formally validated [1].

The reader will notice the similarities between this protocol and that of section 4. The main difference is that order is established immediately in the first phase of the protocol, by inserting messages in a queue by the order they have crossed the network. However, note that a message can only be considered safely ordered when a decision arrives. Before that point, there is the possibility of message retransmission, and the consequent re-ordering of the message in the queue. A consequence of this strategy is that now every message must be explicitly accepted or *rejected* (that is discarded). Due to their similarities, it's almost trivial to modify the two-phase accept algorithm to implement a specialized atomic multicast synchronization protocol. The specialized protocol is presented in figure 6. It is almost a line-by-line copy of the two-phase accept used for atomic QOS, with minor changes such as the start of candidate clocks and the exchange of timestamps and clock adjustments (lines 86, 87, 83 and 99). The techniques for initialization, integration and dynamic adaptation to faults described in the previous section are still valid for this version of the algorithm.

7 Performance

This section presents performance results of the implementation of the *a posteriori* clock synchronization algorithm provided by the *x*AMp protocol suite. We have collected results from two different infrastructures: (i) using the specialized communications board developed in the DELTA-4 project [10] with the protocol running on a real-time executive (the SPART[7] kernel) and accessing a token-bus network; (ii) using SUN-4 workstations, with the protocol running in the kernel and accessing an Ethernet network.

The precision of the convergence function executed by the *a posteriori* agreement algorithm was

[7] SPART is a registered trademark of Bull S.A.

sender part

80 **when** $cc_m^{i-1} = kT$ **do**
81 *tr-w-resp* $(\langle \text{start}, i, m \rangle, \mathcal{R})$;
82 **if** ($\exists r \in \mathcal{R} : r$ *is* $\langle valid \rangle$) **then**
83 *choose* $\Delta_{cc}^{i,m}$; *tr-w-resp* $(\langle acc_{i,m}, \Delta_{cc}^{i,m} \rangle, \mathcal{R})$;
84 **else** *tr-w-resp* $(\langle rej_{i,m} \rangle, \mathcal{R})$;

receiver part

85 **when** *message* $\langle \text{start}, i, n \rangle$ *received from processor* n **do**
86 $t_m^{i,n}$ = *reception (real) time of* $\langle \text{start}, i, n \rangle$
87 *start* $cc_m^{i,n}(t_m^{i,n}) = iT$; $start^i := start^i + 1$;
88 *remove* $\langle i, n \rangle$ *from* $\mathcal{Q}$; *add* $\langle i, n \rangle$ *to* $\mathcal{Q}$; *start* $wdTimer_{\langle i,n \rangle}$;
89 **if** ($start^i \leq f_p$) **then** *send* $\langle$ notsure, $cc_m^{i-1}(t_m^{i,n}) \rangle$ *to* n
90 **else** *send* $\langle$ *valid*, $cc_m^{i-1}(t_m^{i,n}) \rangle$ *to* n
91 **when** *message* $\langle acc_{i,p}, \Delta_{cc}^{i,p} \rangle$ *received from processor* p **do**
92 *stop* $wdTimer_{\langle i,p \rangle}$; *send* $\langle ok_{acc} \rangle$ *to* p;
93 $accepted_{\langle i,p \rangle}$:=*true*; $cc_m^{i,p} = cc_m^{i,p} + \Delta_{cc}^{i,p}$
94 **when** *message* $\langle rej_{i,q} \rangle$ *received from processor* q **do**
95 *stop* $wdTimer_{\langle i,q \rangle}$; *send* $\langle ok_{rej} \rangle$ *to* q; *remove* $\langle i, q \rangle$ *from* $\mathcal{Q}$;
96 **when** $wdTimer_{\langle i,q \rangle}$ *expires* **do**
97 *call recovery function (accept or reject will be received)*
98 **when** $\langle i, r \rangle$ *is on top of* $\mathcal{Q}$ *and* $accepted_{\langle i,r \rangle}$ **do**
99 **if** ($cc_m^i \neq$ *none)* **then** $cc_m^i := cc_m^{i,r}$;

Figure 6: Specialized protocol.

proven [15] to be limited by:

$$\delta_v \geq (1 + \rho_p)\Delta\Gamma_{tight} + 2\rho_p\Gamma_{agreem}^{max}$$

Thus, we have concentrated on measuring $\Delta\Gamma_{tight}$, which is the limiting factor of the precision achieved by our protocol. The value of $\Delta\Gamma = \Gamma^{max} - \Gamma^{min}$, is not relevant for our algorithm and strongly depends on the network access method, network configuration and — particularly — network load. However, for comparison of *a posteriori* agreement with other algorithms we can use as reference values, $\Gamma^{min} < 1ms$, $\Gamma^{max} \in [10ms, 50ms]$ (with load), and a typical value for Γ within the range of $[1ms, 1.5ms]$.

Due to the characteristics of $\Delta\Gamma_{tight}$, the set-up for making the measurements is an interesting issue on its own. Since Γ_{tight} is the real time difference between two distributed events, we had to resort to some external measurement tool. Thus, we have instrumented the clock-synchronization code to, upon reception of synchronization messages, activate some externally accessible pins of the hardware infrastructure. On SUN workstations, we have used one of the RS232 serial port control signals. These signals were collected by a data-analyzer, allowing us to measure the real time interval between the associated events. It is interesting to notice that, since we were measuring very low values (usually, far less than $1ms$), we had to enforce the direct activation of external pins. For instance, in the SUN workstations we had to access directly the port controller (in our machines, the Zilog 8530 SCC) since the access through a kernel service introduced a variance as large as the intervals we were trying to capture.

Scenario	typical	worst-case
$\Delta\Gamma_{tight}$		
SUN (C)	$200\mu s$	$500\mu s$
SUN (B)	$100\mu s$	$200\mu s$
SUN (A)	-	-
in SPART	$40\mu s$	$100\mu s$
Γ_{agreem}		
SUN (x10^{-6})	$2ms (< 0.5\mu s)$	$< 100ms (< 1\mu s)$

Table 3: $\Delta\Gamma_{tight}$ and Γ_{agreem}

The measurements are summarized in table 3. In the SUN infrastructure, we have collected values from two different software layers. In fact, although in both cases the protocol executes in the kernel, the values clearly show that a better precision is obtained in the lower layers of the architecture (this is also demonstrated by the results of Kopetz [6]). The software executing in the SUN kernel can be viewed as having three layers: (A) the proprietary Ethernet device-driver; (B) the *abstract network* layer, that gives to the *x*AMp an abstraction of a generic local-area network, and; (C) the *x*AMp layer, where the protocol runs. The buffering time between the abstract network and the *x*AMp software (and its variance) is enough to raise $\Delta\Gamma_{tight}$ from $200\mu s$ to $500\mu s$. We have not changed the SUN's Ethernet driver but we can assume that if the sources were available, we could reach values similar to those obtained in the DELTA-4 boards, where $\Delta\Gamma_{tight}$ is less than $100\mu s$. We have made our measurements under different network loads, by artificially loading the network with background traffic. Unlike, Γ^{max}, the value of $\Delta\Gamma_{tight}$ suffered no influence from the network load. This is an important advantage of our algorithm in comparison with algorithms that depend on $\Delta\Gamma$.

The precision is also affected by the drift of the clock during the time required to reach agreement. However, since physical clocks have a drift rate in the order of the 10^{-6}, even a conservative value of $100ms$ for the agreement time (typical values are in the order of $2 - 6ms$, depending on the number of group members) adds just a $1\mu s$ to the algorithm precision.

8 Conclusions

Convergence-non-averaging algorithms are attractive because they use a convergence function both to generate the re-synchronization event and to adjust virtual clocks. However, the precision achieved by previous algorithms in this class was limited by the maximum message transit delay in the system. We showed that the properties of broadcast networks can be used to overcome this disadvantage. The *a posteriori* agreement approach removes the dependency of precision on message delay variance or maximum, and shifts it to the dependency of $\Delta\Gamma_{tight}$, a value drastically smaller in most existing LANs.

In this paper we have shown that a reliable broadcast protocol suite, with accompanying group membership services, can simplify the implementations of our protocol. The *x*AMp communications package was specially designed to make full profit of the properties of broadcast networks, thus it is a perfect match for the *a posteriori* agreement requirements. Fundamental to the *a posteriori* agreement technique is the generation of "simultaneous" events to start precise candidate clocks and the subsequent execution of an agreement protocol to select the appropriate candidate clock to be used during the next synchronization period. The *x*AMp provides the qualities of service required to achieve both goals. A specialized atomic multicast synchronization protocol, obtained as a variant of the atomic QOS, was also presented.

Finally, the paper presents performance results obtained on two different infrastructures. Timings validate the design choices and clearly show that our algorithm is able to provide improved precision without compromising accuracy and reliability.

Acknowledgments

The authors are grateful to W. Wogels who helped to set-up the performance measurement tools. We are also grateful to F. Cosquer, J. Rufino, and to the anonymous reviewers for their comments on earlier versions of this paper.

References

[1] M. Baptista, L. Rodrigues, P. Veríssimo, S. Graf, J.L. Richier, C. Rodriguez, and J. Voiron. Formal specification and verification of a network independent atomic multicast protocol. In J. Quemada, J. Mañas, and E. Vazques, editors, *FORMAL DESCRIPTION TECHNIQUES, III*, IFIP, pages 345–352. North-Holland, 1991.

[2] D. Couvet, G. Florin, and S. Natkin. A Statistical Clock Synchronization Algorithm for Anisotropic Networks. In *Proceedings of the Tenth Symposium on Reliable Distributed Systems*, pages 41–51. IEEE, 1991.

[3] Flaviu Cristian. Probabilistic Clock Synchronization. *Distributed Computing, Springer Verlag*, 1989(3), 1989.

[4] A.L. Hopkins, T.B. Smith, and J.H. Lala. FTMP - A Highly Reliable Fault-Tolerant Multiprocessor for Aircraft. *Proceedings IEEE*, 66(10):1221–1240, October 1978.

[5] Hermann Kopetz and Wilhelm Ochsenreiter. Clock Syncronization in Distributed Real-Time Systems. *IEEE Transactions on Computers*, C-36(8):933–940, August 1987.

[6] Hermann Kopetz and Wolfgang Schwabl. Global time in distributed real-time systems. Technical Report 15/89, Technische Universitat Wien, Wien Austria, October 1989.

[7] C.M Krishna, K.G. Shin, and R.W. Butler. Ensuring Fault Tolerance of Phase-Locked Clocks. *IEEE Transac. Computers*, C-43(8):752–756, August 1985.

[8] L. Lamport and P. Melliar-Smith. Synchronizing Clocks in the Presence of Faults. *Journal of the ACM*, 32(1):52–78, January 1985.

[9] Stephen R. Mahaney and Fred Schneider. Inexact agreement: Accuracy, precision, and graceful degradation. In *Proceedings of the Proceedings 4th Symposium Princ. Distributed Computing*, pages 237–249, Montreal-Canada, August 1985.

[10] D. Powell, editor. *Delta-4 - A Generic Architecture for Dependable Distributed Computing*. ESPRIT Research Reports. Springer Verlag, November 1991.

[11] Parameswaran Ramanathan, Kang G. Shin, and Ricky W. Butler. Fault-Tolerant Clock Synchronization in Distributed Systems. *IEEE, Computer*, pages 33–42, October 1990.

[12] L. Rodrigues and P. Veríssimo. *x*AMp: a Multi-primitive Group Communications Service. In *Proceedings of the 11th Symposium on Reliable Distributed Systems*, Houston, Texas, October 1992. INESC AR/66-92.

[13] Fred B. Schneider. Understanding Protocols for Byzantine Clock Synchronization. Technical report, Cornell University, Ithaca, New York, August 1987.

[14] T. K. Srikanth and Sam Toueg. Optimal Clock Synchronization. *Journal of the Association for Computing Machinery*, 34(3):627–645, July 1987.

[15] P. Veríssimo and L. Rodrigues. A posteriori Agreement for Fault-tolerant Clock Synchronization on Broadcast Networks. In *Digest of Papers, The 22th International Symposium on Fault-Tolerant Computing*, Boston - USA, July 1992. INESC AR/65-92.

[16] P. Veríssimo, L. Rodrigues, and M. Baptista. AMp: A highly parallel atomic multicast protocol. In *Proceedings of the SIGCOM'89 Symposium*, Austin-USA, September 1989. ACM.