, , 1–48 ()
© Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

CesiumSpray: a Precise and Accurate Global Clock Service for Large-scale Systems

PAULO VERíSSIMO, LUíS RODRIGUES, ANTÓNIO CASIMIRO paulov@inesc.pt
Technical Univ. of Lisboa - IST/INESC, Instituto de Engenharia de Sistemas e Computadores.
R. Alves Redol, 9 - 6° - 1000 Lisboa - Portugal, Tel.+351-1-3100000. *
Navigators Group, WWW- http://pandora.inesc.pt/

Abstract. In large-scale systems, such as Internet-based distributed systems, classical clock-synchronization solutions become impractical or poorly performing, due to the number of nodes and/or the distance. We present a global time service for world-wide systems, based on an innovative clock synchronization scheme, dubbed CesiumSpray. The service exhibits high precision and accuracy; it is virtually indefinitely scaleable; and it is fault-tolerant. It is deterministic for real-time machinery in the local area, which makes it particularly well-suited for, though not limited to, large-scale real-time systems.
The clock synchronization scheme is a pseudo-hierarchical mix of external and internal synchronization. The root of the hierarchy are the GPS satellites, which "spray" their reference time over a set of nodes provided with GPS receivers, one per local network, where the second level of the hierarchy performs internal synchronization, further "spraying" the external time inside the local network. The algorithm of the second level is inspired on the high precision a posteriori agreement synchronization algorithm, modified to follow an external clock, and able to use simple group communication and membership facilities.

1. Introduction

A global timebase is a requirement of growing importance in distributed systems, to allow decentralized agreement on the time to trigger actions, or on the time at which events occurred. It is also a very useful block for building fault-tolerant distributed algorithms. The common solution for the global timebase problem lies on using the processor hardware clock to create a virtual clock at each node, which is locally read. All virtual clocks are internally synchronized by a *clock synchronization algorithm*. Surveys of existing clock synchronization algorithms can be found in [26], [21]. In large-scale systems, e.g. Internet-based distributed systems, such a cooperative solution becomes impractical or poorly performing, due to the number of nodes, and the distance among them. Hierarchical or master-based algorithms[4], [17], [18] are preferred solutions, since they attenuate this problem.

Another facet of interactive applications running on large-scale systems is the need for coordination in terms of absolute time references, such as TAI or UTC[1]. This also requires external synchronization. The service of [17] allows nodes on the Internet to synchronize their clocks from *master* nodes possessing access to an absolute time reference. The achievable quality of the global time reference in these cases is limited by variations in communication delays between master and other nodes, or by inaccessibility of these masters, due to partitions. Using a probabilistic approach[4] one can trade determinism for precision.

A large-scale distributed real-time system can reasonably be modeled by a WAN-of-LANs. It is more difficult to enforce *hard* properties with regard to real-time in the WAN part[32], but the local network part can be made to have such a stricter behavior[31].

We take advantage of this observation to present a global time service for large-scale (world-wide) systems, based on an innovative clock synchronization scheme, that we called CesiumSpray . The service exhibits high precision and accuracy, is virtually indefinitely scalable, and is deterministic for real-time machinery in the local networks.

The underlying clock synchronization scheme is a mix of external and internal synchronization, which can be seen as pseudo-hierarchical, as depicted in figure 1. As the figure suggests, the root of the hierarchy is the source of absolute time, the set of cesium TAI clocks in the NavStar GPS[19] (Global Positioning System) satellites, which "spray" their unique reference time over a set of nodes provided with GPS receivers (GPS-nodes). This forms the first level of the hierarchy, where the nodes' clocks are set via *external* synchronization, and thus maintained highly accurate and precise.

The second level of the hierarchy works in *internal* synchronization. It is formed by every local network of the system, with the condition that each be provided with at least one GPS-node[2]. Thus, in this level, internal synchronization is performed in a way such that the external time resident in the GPS-node is further "sprayed" inside the local network, i.e. used as the reference for the internal synchronization rounds.

Not all synchronization algorithms are suitable for this latter objective, and for preserving the high precision and accuracy obtained in the first step. We use an internal synchronization algorithm based in the *a posteriori agreement* algorithm. This algorithm is optimized for local networks, and given real-time machinery (network and operating system), may exhibit a precision in the order of the submillisecond, better than that normally found in software-based algorithms. The algorithm uses properties of broadcast networks to drastically attenuate the traditional limitation imposed by message delivery delay variance on the obtained precision. The fundamental principles of the original *a posteriori agreement* algorithm, published in [30], are followed. However, the algorithm used in CesiumSpray is modified to follow an external clock when one exists in the set of synchronizing clocks. It also uses simple group communication and membership facilities, when such facilities exist.

The protocol works as follows: GPS-node clocks are permanently externally synchronized, in absence of failure, and supply the value of their GPS clock when read by the protocol; internal synchronization starts with each processor disseminating a start message at a pre-agreed instant on its clock; after a bounded series of such broadcasts, each tentatively initiating a new virtual clock, an error-free broadcast— one getting to all correct processors— is guaranteed to occur; the broadcast recep-

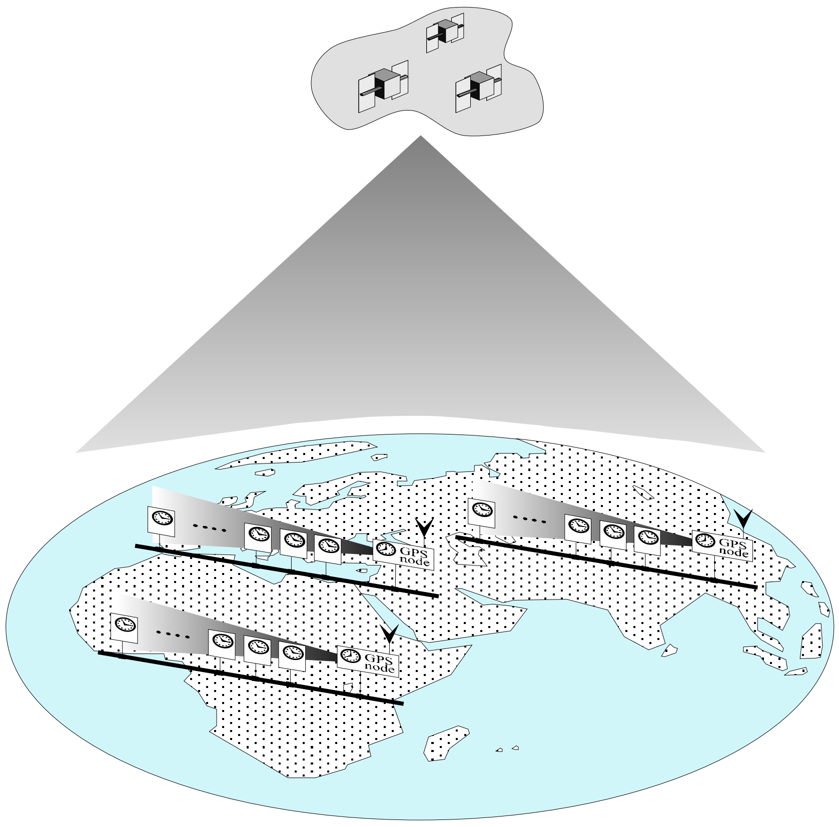

Figure 1. The pseudo-hierarchical nature of CesiumSpray

tion time is practically the same everywhere in the local network, and is marked by each processor's clock; each processor synchronizes by the value of the GPS clock (or one of them, if there exist several), by using the common reference in the time-line yielded by the broadcast reception instant, to compute the relative deviation to the GPS-node's clock at that time, and applying it to its clock.

Fault-tolerance of the external synchronization scheme is achieved by replicating GPS-nodes in the desired local networks. Fault-tolerance of the internal synchronization scheme is achieved by having enough nodes to mask the assumed failures, as explained in the text. If external synchronization fails, the global time is p-reserved by the internal synchronization algorithm, which exhibits quasi-optimal accuracy preservation[30].

The time service described in this paper can readily be materialized for large-scale systems over global WAN-of-LAN networks, such as the wide-area point-to-point Internet, with real-time LAN technologies in the edges. The merit of our scheme is also related to the fact that, for reasons that we will explain later in the text, it is not technically viable to provide every other node with a GPS receiver.

The paper is organized as follows. The main concepts about clock synchronization, needed along the text, are briefly introduced in section 2. The approach taken for the architecture of the system— failure assumptions, local network model, technology issues, and the principles of internal and external synchronization— is described in section 3. Section 4 presents the basic concepts of *a posteriori agreement* and describes the synchronization protocol. Section 5 presents the improved algorithm, illustrating how the use of group communication and membership improve the protocol execution and initialization. Section 6 discusses dynamic adaptation of fault-tolerance and performance parameters. Section 7 addresses the problem of accuracy preservation, when external time is lost and the system continues in internal synchronization. The paper concludes by the evaluation of the service in section 8, with regard to cost in bandwidth and nodes; quality, in precision and accuracy; and resilience to uncoverage of assumptions. Derivation of protocol parameters and bounds, and proofs, are presented in appendix.

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 processors whose processors can interact exclusively by message exchange. processors can only observe time through a *clock*. For convenience, a clock is usually represented by a function $c(t)$ that maps (non-observable) real time[3] to *clock time* (notation generally follows that of [26]).

One commonly used solution to achieve a global timebase is to provide each processor in the distributed system with an imperfect physical clock pc. A correct clock at processor k can then be viewed as implementing, in hardware, an increasing, discrete[4] function pc_k that maps real time t to a clock time $pc_k(t)$. The physical clock *ticks*, advancing a unit at each tick, t_{tk}, which corresponds to a discrete amount of time g, the *granularity* of the clock. For some positive constants g, μ_p and ρ_p, the function satisfies:

PC 1 (Physical Clock Initial value)

$$0 \leq pc_k(0) \leq \mu_p$$

PC 2 (Correct Physical Clock Rate)

$$0 \leq 1 - \rho_p \leq \frac{pc_k(t_{tk+1}) - pc_k(t_{tk})}{g} \leq 1 + \rho_p \quad for \quad 0 \leq t_{tk} < t_{tk+1}$$

Through a clock synchronization algorithm it is possible to derive, from the physical clock at each node k, a virtual clock vc_k satisfying the following conditions:

VC 1 (Precision)

$$|vc_k(t) - vc_l(t)| \leq \delta_v, \quad for \quad 0 \leq t$$

VC 2 (Rate)

$$1 - \rho_v \leq \frac{vc_k(t_{tk+1}) - vc_k(t_{tk})}{g} \leq 1 + \rho_v \quad for \quad 0 \leq t_{tk} < t_{tk+1}$$

VC 3 (Envelope Rate)

$$1 - \rho_\alpha \leq \frac{vc_k(t) - vc_k(0)}{t} \leq 1 + \rho_\alpha, \quad for \quad 0 \leq t$$

VC 4 (Accuracy)

$$|vc_k(t) - t| \leq \alpha_v, \quad for \quad 0 \leq t$$

Precision δ_v characterizes how closely virtual clocks are synchronized to each other, ρ_v is the rate drift of virtual clocks, ρ_α is the long term rate drift of virtual clocks. Applied to a set of clocks, the latter characterizes the envelope of their rates. ρ_v characterizes the instantaneous rate measurable between any two clock ticks. Its maintenance calls for continuously adjusted clocks, as discussed later in the paper.

Accuracy α_v characterizes how closely virtual clocks are synchronized to real time at any moment. An interesting consequence is that in a set of clocks with accuracy α_v, precision is at least as good as $\delta_v = 2.\alpha_v$. Due to the nonzero rate drift of physical clocks (normally quartz crystal), accuracy cannot be ensured unless some external source of real time is available, i.e. when there is the possibility of *external* synchronization.

In CesiumSpray , external time is injected by GPS-nodes, which have *GPS-clocks*, i.e. virtual clocks synchronized by the NavStar system, with the following property:

VC 5 (GPS-clock Accuracy)
$|vcg_k(t) - t| \leq \alpha_g, \quad for \;\; 0 \leq t$

An external source is not always necessary, or available. In the context of *internal* synchronization, a good algorithm should maintain virtual clocks as close as possible to real time, by minimizing[5] ρ_v and ρ_α. In that sense, it should *preserve* accuracy, by fulfilling conditions **VC 2** and **VC 3**.

As we have seen, physical hardware clocks are permanently drifting from each other. In consequence, from a precision viewpoint, virtual clocks must be re-synchronized from time to time, in such a way that condition **VC 1** holds. A clock synchronization algorithm should then be able to generate a periodic re-synchronization event. The time interval between successive synchronizations is called the re-synchronization interval, denoted T.

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* [26]. The *precision enhancement* property specifies the best precision guaranteed right 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, the drift, ρ_p in **PC 2**, typically of the order of $10^{-6}s$, will mainly dictate the interval T duration. The precision enhancement property of the convergence function is the relevant quality factor of a clock synchronization protocol.

3. Time service architecture

Real-time applications in general require accuracy towards some real time reference. This implies external synchronization. Therefore, it is necessary to disseminate external time to all the nodes. Having in mind that, due to failures, the external synchronization source may not always be available, algorithms that preserve accuracy, like the one in [28], become an important choice for internal synchronization. Additionally, systems oriented to distributed real-time applications require a precision better than that normally achieved with software-based algorithms. In conclusion, the internal synchronization algorithm should not deteriorate the excellent quality of external synchronization.

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 [14], or by its upper bound [28]. This problem may be attenuated in special architectures, either by implementing clock synchronization exclusively by hardware [8], [13] or by using hybrid schemes [21], [11] which attempt at reducing that variance. In large-scale systems, the distance, added to the very large number of nodes, worsens the variance problem. Hierarchical or master-based algorithms, using probabilistic or statistical techniques to damp the effect of variance have been proposed [4], [2], [17], [18].

An alternative path was followed here, based on two realistic assumptions:

- distributed real-time systems are based on broadcast local networks;
- large-scale systems can be modeled as a point-to-point WAN-of-LANs.

Local area networks are commonly in use today. However, we know of no previous solution for the clock synchronization problem that fully exploits the intrinsic attributes of these networks: error rate is low, transmission delay is bounded but with high variance, median transmission delay is close to the minimum, and message reception is *tight* in absence of errors, meaning that the low-level message reception signal occurs at approximately the same time in all nodes that receive it. This is a crucial feature for the mechanism underlying the synchronization algorithm, as will be shown ahead.

Protocols using the convergence-non-averaging technique [7], [28] are attractive. Since they are based on disseminating the event that "a node believes it is now a pre-agreed time" rather than on direct read-clock requests, they are inherently resilient to failures, requiring less messages and synchronization cycles than averaging algorithms [14], [16], [3].

The algorithm of Srikanth & Toueg among the former is appealing for its simplicity, ease of implementation, and optimal accuracy preservation. The reader is referred to [28] for a detailed description. However, in order to achieve sufficient evidence that a synchronization round is starting, processors relay messages, which allows the difference between two synchronization actions at different nodes to be as large as the maximum message delivery delay.

In this paper, we show how we developed a convergence-non-averaging technique with optimal precision on local networks. Furthermore, we present our method for scaling to the hierarchical two-tier network, using the GPS as the link between local networks.

3.1. Assumptions

Before proceeding, we present our assumptions about the system:

- **clocks** may have arbitrary failures (eg. provide erroneous or conflicting values when read);
- clock server **processors** (the ones running the synchronization protocol) may have failures from crash to uncontrolled omission or timing failures;
- the maximum number of clock-processor pairs with failures during a protocol execution is f_p.

- the maximum number of GPS-nodes with failures during a protocol execution is f_g.

In the system, each network *node* holds a clock *processor*, so we will use both words interchangeably. The reader will note that the combined assumption of arbitrary-failure clocks and 'omissive'-failure processors is realistic and not constraining, by allowing faulty processors to be **arbitrarily** delayed or even omit their participation in the algorithm in an **uncontrolled** manner, whereas their 'assertive' failed behavior is limited to sending wrong clock values, thus avoiding impersonation, collusion, etc. This removes the difficulty of handling genuinely arbitrary-failure (eg. Byzantine) processors. However, we believe that: (*i*) this processor failure model closely matches the behavior of real-life processors in a distributed system; (*ii*) it is weaker and easier to implement with high coverage, than a fail-silent one; (*iii*) the results presented here can be extended to arbitrary failures, by using signatures [28] and redundant broadcast channels.

The network is a single-channel broadcast local network, as detailed ahead, with the following failure semantics:

- the **network** components: (i) are weak-fail-silent[29], confined to crash if they exceed a given number of omission failures (otherwise behaving correctly); (ii) have a bound f_o on the number of omission failures they can produce during a protocol execution.

It is possible to put a bound on the time to send a message, to process a received message and read a clock value, etc.:

- the network and the clock server sub-systems are **synchronous**, in the sense of exhibiting known and bounded processing and communication delays.

3.2. Broadcast local network model

This section shows that broadcast local area networks have a number of properties on which clock synchronization may be built, namely the ability to deterministically generate a "simultaneous" event at all correct processors in the system.

A study on the influence of network timing properties on clock synchronization presented in [12] 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[6]; and *Receive time*, Γ_{rec}, to process the message at the receiver. These contributions are illustrated in figure 2. The precision of an algorithm is affected by the error components introduced by the variances of these terms, $\Delta\Gamma_{xxx}$, which together make up the message delivery delay variance. For example, we call the send time variance, $\Delta\Gamma_{send}$, the *send error*.

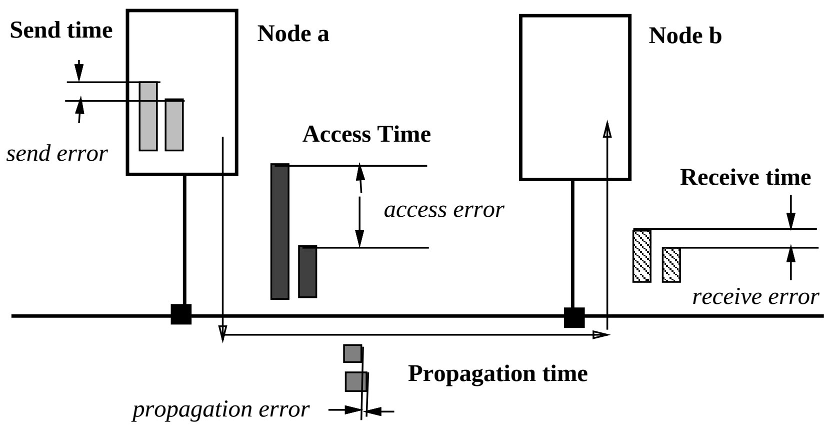

Figure 2. Network timing properties

In order not to depend on a particular network, the best approach is to define an *abstract broadcast network*, such that standard local area networks or their variants are represented [9], [10], [6], [15]. The network model of [29] is followed, though modified to be more generic. The abstract network components are: the *channel*, which comprises the passive medium and the interfacing electronics; and the *adapter*, comprising the low-level network protocols, implemented partly in VLSI partly in firmware. The abstract broadcast network appears to the user processors/protocols (namely the clock processors) as a low-level service with a set of properties and an interface [7].

Properties

BNP 1 (Broadcast) *Nodes receiving an uncorrupted message transmission, receive the same message* [8].

BNP 2 (Error detection) *Nodes detect any corruption done by the network in a locally received message and discard it.*

The network is thus prevented from altering messages, impersonating other senders or delivering conflicting information to different processors on the same broadcast. Existing broadcast networks usually implement cyclic redundancy checks for this purpose.

BNP 3 (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.*

This assumption yields a very simple solution to the membership problem as explained in the next section. It is equivalent to expecting that in f_o+1 transmissions, at least one is heard by all nodes. It has a very high coverage in local networks, provided that f_o is well-chosen. Note that under this assumption, an omission failure may be perceived inconsistently, i.e. a transmission that is not seen by only some (or one) of the recipients. On the other hand, omission count is per transmission, independent of the number of recipients that do receive omission failures for that transmission.

BNP 4 (Bounded Transmission Delay) *The time between any broadcast send request and the relevant 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:

$$\Delta\Gamma = \Gamma^{max} - \Gamma^{min}$$

Securing property **BNP 4** depends on network type and on additional assumptions about its operation, namely that rate and inter-arrival time of message transmission requests are bounded. The reader should note that this assumption must accommodate the load generated by the protocol itself at the synchronization points, which is known a priori. Existence of the bounds just mentioned allows estimating individual transmission delays in the presence of bounded background loads and queue lengths. In these conditions, Γ^{max} holds for every one of several concurrent transmission requests. For details about enforcing reliable real-time operation on a local network, the reader is referred to [31].

Also worthwhile noting — it will become clearer later on — that the time spent in queue by a clock-reading message does not influence precision, unlike any other software-based protocol we know of, apart from the second-order effect of the physical clock rate drift ρ_p.

BNP 5 (Tightness) *Nodes receiving an uncorrupted message transmission, receive it at real time values that differ, at most, by a known small constant* $\Delta\Gamma_{tight}$.

It is important to understand the timing properties of local broadcast networks. The *propagation error* is very small: in a Token-bus or 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: however, it remains more or less constant and may be improved as discussed ahead in the text. On the contrary, the *access error*, $\Delta\Gamma_{access}$, is hardly controlled and it can have a significant range, strongly depending 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 the total 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:

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

In the scope of the abovementioned properties, we make the following definitions:

Tight Broadcast (TB) - a single broadcast transmission, that is received by all correct nodes within $\Delta\Gamma_{tight}$.

Detected Tight Broadcast (dTB) - a tight broadcast known to be such by at least one node.

3.3. Technology issues

If every processor had access to a common reference of time, it could use it as the global timebase. One way to grant such an access is through a long-wave radio receiver, capturing an international time standard like *Universal Time Coordinated, UTC*, or a GPS satellite signal, carrying the *Temps Atomic International, TAI*, from its atomic cesium clock.

However, having a radio receiver at each processor is economically very expensive: one would like to have only one of these — or a few, for fault-tolerance — per system. Furthermore, the availability of radio time may be insufficient for some applications [12].

The NavStar Global Positioning System, GPS[19], is a network of 21 satellites covering the earth surface in a very complete way, so that normally at least 4 of them are above the horizon. Although used mainly for positioning and navigation, the feature of interest for this paper is that they provide an extremely good source of absolute time— a chronoscopic reference of the TAI kind— from their cesium atomic clocks, with a stability in the order of $\rho_g \simeq 10^{-14}$, that is, 1 s in 3 000 000 years. Satellite clocks are monitored and corrected periodically, in conditions which, given ρ_g and other errors deriving from propagation, ensure an accuracy on ground of $\alpha_g \leq 100ns$ [19], for the GPS-clocks installed in the GPS-nodes.

GPS receivers, on the other hand, have drastically been going down in price. Additionally, the availability of signal reception is higher than for the radio markers. The system only degrades its positioning ability in case of war, by control of U.S. authorities. However, the GPS receiver antenna must be under the light cone of the satellites it is receiving from during a 24h period[19], that is, external. As such, there are obvious restrictions to the number and location of GPS receivers in a distributed computer system and, neatly, one GPS receiver per node is impractical, even if prices continue to go down.

CesiumSpray picks the most appropriate reception technology for external time (GPS), and presents an effective solution to reduce the number of receivers to one per local network. With this: cost is kept acceptably low; the system management hindrance of having a GPS unit attached to every other node is avoided; the requirement for external mounting of an antenna out of a relatively short cable, is put on a per-local-network basis, which is acceptable.

With regard to network technologies, the applicability of the scheme is very wide, since it is appropriate for local networks with real-time behavior. This includes technologies such as token-bus, token-ring, FDDI or deterministic Ethernet [9], [10], [6], [15]. The system will also work on plain Ethernet. However, the quality of synchronization will be subject to the coverage of the assumption of time-bounded Ethernet operation, known to be probabilistic. The reader may refer to section 8, where a discussion about resilience to assumption violations is made.

Observe that the scope of 'local network' may also be extended, to include, for example, ATM-based networks with multicast ability, since they can readily be made to secure the **BNP** properties. The propagation error through an ATM fabric of 'local' dimension is rather small.

CesiumSpray can be materialized on large-scale systems structured as a WAN-of-LANs. This includes virtually any large-scale distributed computing infrastructure as we see them today— such as the wide-area point-to-point Internet. Given its pseudo-hierarchical nature, it has virtually unlimited scalability. Since that it supposes real-time behavior of the local networks, it is particularly well-suited for large-scale real-time systems.

Current Internet-based synchronization schemes, such as NTP[17], effective as they may be today, cannot reach the effectiveness of CesiumSpray , because they do not relate the synchronization architecture to the network architecture. The location of external time masters in NTP is not related to the existence of local networks with BNP properties, such as in CesiumSpray : reading a master clock may mean crossing several Internet gateways.

3.4. The approach taken

The architecture of CesiumSpray is shown in figure 3. As said in the introduction, the clock synchronization scheme is a pseudo-hierarchical mix of external and internal synchronization. The root of the hierarchy is the source of absolute time, the NavStar GPS, which "sprays" its time over the set of GPS-nodes. The second level of the hierarchy is formed by every local network of the system (anywhere in the world), with the condition that each be provided with at least one GPS-node. The external time resident in the GPS-node is further "sprayed" inside the local network through an internal synchronization algorithm.

From the accuracy viewpoint, all GPS-clocks are α_g accurate to real time. The internal synchronization algorithm is such that it imposes the GPS-node time on the other nodes, and guarantees a precision of the local set of clocks of δ_l. That is, accuracy of the latter to the GPS-node absolute reference is bounded by δ_l. In consequence, the global accuracy of CesiumSpray comes:

$$\alpha_{CS} = \alpha_g + \delta_l$$

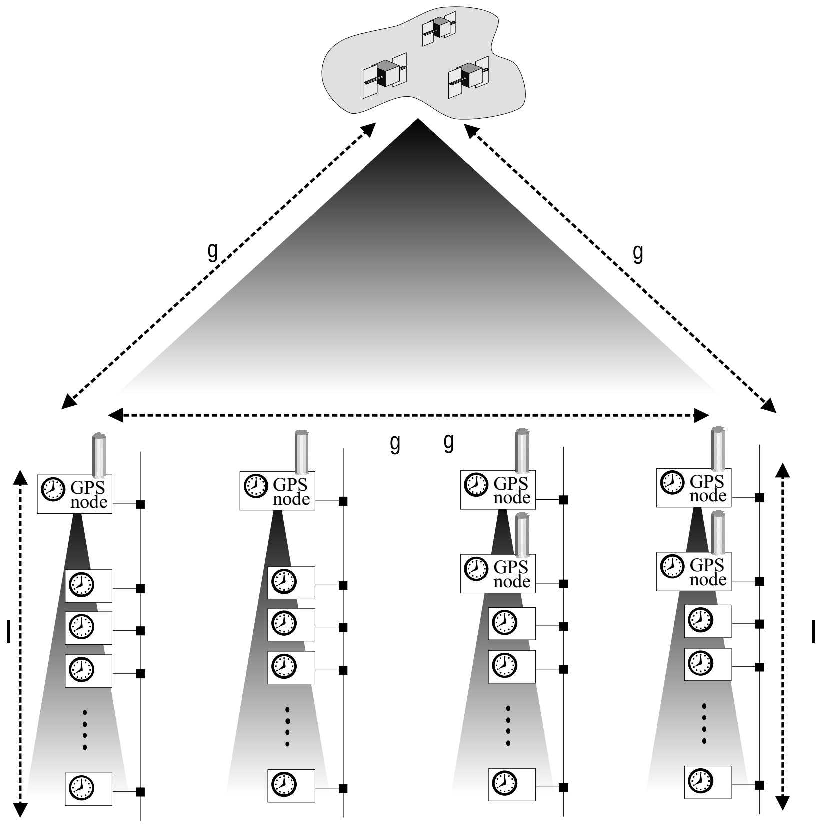

Figure 3. The architecture of CesiumSpray

On the other hand, it is easy to see that the worst-case precision of CesiumSpray world-wide, between any two nodes, is just influenced by three terms: precision of GPS-clocks amongst themselves, which is twice their accuracy (cf.§ 2); two times the internal precision in a local network. The contribution of these terms is illustrated in figure 3. Proofs are given in appendix. Precision of CesiumSpray can thus be easily derived:

$$\delta_{CS} = 2\,\alpha_g + 2\,\delta_l = 2(\alpha_g + \delta_l)$$

Not surprisingly, $\delta_{CS} = 2\,\alpha_{CS}$. We know that α_g is extremely small. Internal precision δ_l will be dictated by the *a posteriori agreement* technique. Its aim is to improve precision by making it depend on $\Delta\Gamma_{tight}$ (instead of $\Delta\Gamma$ or Γ^{max}). $\Delta\Gamma_{tight}$ is very small: as we show in section 8, we have measured values below $100\mu s$. This being the case, CesiumSpray will have excellent precision and accuracy, as promised in the introduction.

To understand how, let us consider the operation of broadcasting a read command S to all clocks in a local network and getting *all* replies in a bounded time, despite errors. In a real-time local network it is possible to define a bound on the number of tries and the amount of time needed to execute the operation above[9]. The reader will note four attributes of such a fault-less broadcast which are crucial for the understanding of the algorithm proposed:

- (i) *send* and *access* errors, meaningful in competing unicast transmissions, do not count in a single fault-less broadcast;
- (ii) the message S transmitted arrives virtually at the same time on all nodes, the difference corresponding to the *propagation* error;
- (iii) processing times of S reception at any two nodes vary at most by the *receive* error;
- (iv) all replies to S get back to the transmitter.

If message S, addressed to all including the sending node, meant: "Let us synchronize! I think the time is H. What time is it on your clocks?", one concludes the following:

- *precision enhancement:* in response to S, a new virtual clock is tentatively initiated everywhere with H, at the same physical time more or less an error equal to the *propagation* plus *receive* errors;
- *accuracy preservation:* also at that time, the clock of each recipient is read and delivered back to the sender; the sender selects the best clock in terms of rate or accuracy (eg. the median of the clocks, with purely internal synchronization, or one of those that have an external time reference, with external synchronization) and computes its difference to H, to adjust accuracy of the tentative clocks.

This happens every time a processor broadcasts S with success. There will be a number of tentative virtual clocks launched, and an election protocol is run (a posteriori) to agree on one of them, together with the adjustment.

At this point, it is important to signal that this protocol has new qualities vis-a-vis the averaging and non-averaging families of protocols. Averaging protocols, namely agreement-based ones, exchange the values of their clocks and calculate a convergence function, either in a centralized or decentralized manner. Precision directly depends on the variance of the delay in computing the adjustment or in disseminating it. Non-averaging protocols emit a time marker which, upon received, can trivially set the local clock to a preset value. Clock values are neither read nor transmitted. Precision directly depends on the reception instants of the marker.

A posteriori agreement allies the time marker principle of non-averaging protocols to the clock reading and agreement principle of averaging ones. Precision is achieved solely at the cost of "simultaneity" of reception of the time marker: it depends on the *receive* and *propagation* errors. Should all clocks be initialized with some value H at that moment, they would be fairly precise. Should H be chosen as the value of a selected accurate clock at the marker time, they would also be accurate. This is obtained a posteriori (thence the name): the values of all clocks at the marker time are read, the selected clock is agreed upon, clocks are corrected by adding an adjustment.

Unlike genuine agreement-based averaging protocols, precision of the a posteriori agreement protocol only has a *second order* dependence on the delays in agreement and computation of adjustment, caused by the rate drift, ρ_p, during that period. However, this effect is negligible, given that ρ_p is very small, as discussed in section 2.

4. Basic a posteriori agreement protocol

The principles of using the *Tightness* property (BNP 5) to allow very precise and accurate clock synchronization were already discussed. In the presence of failures though, incorrect processors/clocks may participate, and broadcasts may be only received by a subset (possibly empty) of the nodes in the system. The clock synchronization algorithm should then be able to:

- ensure that at least one tight broadcast is generated and detected;
- ensure that, when several tight broadcasts are generated, all correct processors choose the same broadcast and adjustment to synchronize their clocks;
- ensure that a tight broadcast is generated often enough to prevent virtual clocks to drift apart more that the desired *precision*;
- ensure that a new clock, when it starts to be used, has a value that preserves the desired *envelope rate*[10].

First, it is described how a tight broadcast can be generated and detected. Then, the achievement of *precision* and preservation and achievement of *accuracy* are discussed.

4.1. Generating, detecting and agreeing on a tight broadcast

The protocol is based on having every processor perform the same two basic actions: broadcast *once* a "start synchronization" message; and reply (in broadcast) to such messages coming from other processors. This way, modification of failure assumptions only influences the number of processors required to run synchronizations successfully.

With the present assumptions, the presence of $f_o + f_p + 1$ processors in the system is required to *generate* at least one tight broadcast, given that: each node tries only once; f_p processors may not transmit (eg. processor omissions or crashes); and f_o network omissions may occur (**BNP 3**).

Detecting the generation of a tight broadcast is more delicate: it requires feedback from the recipients of the broadcast. Let us assume that each correct recipient broadcasts an acknowledgment message $\langle \mathrm{ack}_b \rangle$ in response to a given broadcast $\langle b \rangle$. For the sake of simplicity, and without loss of generality, it is assumed that the time required to create the acknowledgment message is included in Γ.

Since we do not have a powerful tool such as a group membership management protocol, we use a very simple scheme based on three facts[11]:

- the set of processors $\mathcal{P}$ is known by all processors and is static;
- as per **BNP 4**, in absence of failures a correct processor, after the reception of a broadcast, should receive an acknowledgment message from every other correct processor by $\Gamma^{max} + \Delta\Gamma_{tight}$ (real) time (cf. §3.2);
- with the help of **BNP 3**, which accounts for actual network omissions, faulty processors can be detected, if they appear to do more than f_o omission failures[12].

The procedure for detection of a tight broadcast is depicted in figure 4 for a better understanding, although it is embedded in the algorithm of figure 5. Let $\mathcal{P}$ be the set of processors in the system, known a priori by all. Let $\mathcal{P}_q^b$ be the set of correct processors in the execution of broadcast b, from processor's q point of view (initially $\mathcal{P}_q^b = \mathcal{P}$) (line 10). For each processor q and for each broadcast message $\langle b \rangle$, let $\mathcal{A}_q^b$ include all processors from which an $\langle \mathrm{ack}_b \rangle$ message was received (l.30), and let $\mathcal{F}_q^b$ include those processors from which no acknowledgment has been received within the expected time interval (l.40). Let also $\mathcal{D}_q$ be the set of tight broadcasts *detected* by q (l.60).

A given processor p can be considered faulty by a processor q if p appears to q as having done more than f_o omissions, i.e. appearing in more than f_o $\mathcal{F}_q^{b_k}$ sets. In that case, q withdraws it from its view (line 50). When— because all expected replies did eventually arrive, or because some faulty processors were meanwhile withdrawn from $\mathcal{P}_q^b$— the sets $\mathcal{P}_q^b$ and $\mathcal{A}_q^b$ match (1.60), q detects broadcast $\langle b \rangle$ as a *tight broadcast*, and inserts it in $\mathcal{D}_q$.

For processor q

```
10    $\mathcal{D}_q = \emptyset$;
20    when message $\langle b \rangle$ is received do
          $\mathcal{A}_q^b = \mathcal{F}_q^b = \emptyset$;
          $\mathcal{P}_q^b = \mathcal{P}$ od
30    when $\langle \mathrm{ack}_b \rangle$ message is received from processor $p$ do
          insert $p$ in $\mathcal{A}_q^b$ od
40    when $q$'s clock reads $\Gamma^{max}$ after reception of $\langle b \rangle$ do
          $\mathcal{F}_q^b = \mathcal{P}_q^b - \mathcal{A}_q^b$ od
50    when $\exists p, n > f_o : \forall k, 1 \leq k \leq n, p \in \mathcal{F}_q^{b_k}$ do
          remove $p$ from $\mathcal{P}_q^b$ od
60    when $\mathcal{A}_q^b = \mathcal{P}_q^b$ do
          insert $b$ in $\mathcal{D}_q$ od
```

Figure 4. Detecting a tight broadcast

The mechanism just described does not prevent the generation of several simultaneous clock synchronization events. An election protocol must be run afterwards, to select only one broadcast. No particular protocol is required, as long as election is reached in a known bounded time. Fault-tolerant agreement protocols are well-known and can be easily found in the literature, although existing *reliable broadcast protocols* for broadcast networks are recommended [29], [5]. The reader should note that the simplest form of election that fulfills all the requirements of synchronization put forward in this paper is: 'select the first tight broadcast detected by all'.

4.2. Achieving precision

The first phase of the algorithm (figure 5) is very similar to the algorithm of [28]. Let us further define: T, resynchronization period; r_q, next synchronization round, from q's perspective— a round r_q starts when the local clock reads $r_q T$; $vc_q^{r_q}(t)$, the value of processor's q virtual clock after synchronization round r_q, at real time t; $cc_q^{i,p}(t)$, the value of candidate clock launched at real time t at processor q, in response to processor p broadcast starting synchronization round i; $\mathcal{V}_q^l$, the vector of clock readings obtained by q from the network, in response to the tight broadcast started by l: $\mathcal{V}_q^l = vc_1(t_1^l), \ldots, vc_n(t_n^l)$, for $n = \#\mathcal{P}_q^l$.

The former set $\mathcal{A}_q^b$ of figure 4— the acknowledging processors to a broadcast b by l— is now unfolded in: $\mathcal{A}_q^l$, the set of processors responding with $\langle notsure \rangle$, and $\mathcal{C}_q^l$, the set of processors responding with $\langle candidate \rangle$.

When $vc_q^{r_q-1}(t) = r_qT$, processor q decides to start the synchronization activity for round r_q, sending a $\langle start,\ r_q,\ q\rangle$ message (lines 10-11). Note that we postpone the discussion of round 0, until the section about initialization. Since faulty clocks/processors can send $\langle start\rangle$ messages out of time, the "achievement of sufficient evidence" [26] is desirable, before a message is eligible for a new virtual clock. The criterion of [28] is used: given that f_p clock/processor pairs may fail in an untimely manner, a $\langle start\rangle$ message can be considered correct if it has been received at least from $f_p + 1$ distinct processors, out of at least $2f_p + 1$.

Note that our protocol can withstand occasional early start messages, since the new clock value, unlike [28], does not depend on the time they are issued, but on the clock-readings vector. However, also note that the clock-readings vector, for a fault-tolerant *select* function, and thus clock synchronization, must have at least $2f_p + 1$ elements.

The number of processors required to achieve and detect sufficient evidence of a correct synchronization point, is $(f_p + 1)(f_o + 1) + f_p$ (cf. appendix). This bound is always dominant over the $2f_p + 1$ bound. It is thus the number required to correctly execute the complete synchronization protocol. An informal discussion of this bound is as follows: f_p correct processors may broadcast before it is detected as a correct synchronization point; f_p processors may fail to broadcast; the broadcasts of f_o processors may suffer transmit omissions done by the network, not getting to anyone; adding to these are the uncontrolled omissions in acknowledgments, that failed processors may do— in order to detect them, we need to have previously observed at least f_o such omissions per processor in worst case— which yields $f_p.f_o$. This is the worst-case failure scenario, so we finally need one more processor to successfully execute the protocol.

A tentative virtual clock is started upon the reception of every $\langle start,\ i,\ p\rangle$ message (lines 20-21). It is kept running in a *candidate* state. The local clock value at the same time is returned to vc_q, by the *readClock* function, whose detailed functionality we skip for the moment being. All $\langle start\rangle$ messages are acknowledged by correct processors. Before the achievement of sufficient evidence, start messages are acknowledged by a $\langle notsure\rangle$ (l.26). After, they are acknowledged with a $\langle candidate\rangle$ message (l.25). Sufficient evidence is achieved when the number of broadcasts for round i exceeds f_p. This is controlled by the sets $Start^i$ and $Start^0$, updated and tested upon each broadcast received (lines 23-24). Start^0 is formed by joining processors invoking round 0, and let us concede for now that they also count for the number of processors of any other round, we will get back to this issue in section 4.5.

Each processor q monitors all responses to each start message from a processor l that it receives. Depending on the type of response, $\langle notsure\rangle$ or $\langle candidate\rangle$, it updates $\mathcal{A}_q^l$ or $\mathcal{C}_q^l$. It also logs the reading of the clock of the responding processor when it received l's start message, $vc_p(t_p^l)$, in the clock-readings vector $\mathcal{V}_q^l$ (lines 30-42).

The procedure to update $\mathcal{F}_q^l$ and $\mathcal{P}_q^l$ is similar to that discussed in the preceding section (lines 50-61). Note that Γ^{max} can be measured locally, assuming a worst-case rate for the local physical clock, by waiting $(1+\rho_p)\Gamma^{max}$ on the local clock (l.50).

A candidate clock launched everywhere in response to a $\langle start \rangle$ message by a processor e, is eligible from the moment when one processor q recognizes it as such (l.70). For that purpose two things must occur with e's broadcast. Firstly, at least one processor (not necessarily q) must validate it as candidate (lines 30-31), so that q perceives it (line 70, $\mathcal{C}_q^e \neq \emptyset$). Secondly, q must detect e's broadcast as a tight broadcast, i.e. q must have seen all correct processors reply to e's *start* message (line 70, $\mathcal{C}_q^e + \mathcal{A}_q^e = \mathcal{P}_q^e$). The first processor q detecting the eligibility of e invokes the election protocol, proposing e's as the synchronizing broadcast (l.73)[13].

Recapitulating, each tight broadcast starts a candidate clock at every correct processor. Due to the Tightness property of the network, instantiations of the same candidate clock are no further than $\Delta\Gamma_{tight}$ apart from each other. At the end of the election procedure, whichever candidate clock is chosen, a new virtual clock satisfying *precision* can start being used. The re-synchronization interval should be chosen long enough to allow a tight broadcast to be generated, detected and agreed but short enough to ensure that virtual clocks do not drift apart more that the desired worst-case precision. The inequalities required to parameterize the protocol are in the appendix.

4.3. Preserving accuracy

The candidate clocks are initialized with iT (figure 5, l.21), the value of the sender's virtual clock at the *sending* time of $\langle$start$\rangle$. Clearly, while this satisfies precision, it does not preserve accuracy.

One could set the candidate clocks to $cc_q^{i,p}(t_q^p) = iT + \Gamma^{i,p}$, however, Γ can only be estimated. Instead, note that by reading the clocks in response to a tight broadcast— which is also the time at which the candidate clocks are set to iT (l.21)— we do so at approximately the same time, since $\forall j,k,\ |t_j^p - t_k^p| \leq \Delta\Gamma_{tight}$. Then, from the clock-readings vector, the differences between each clock and iT at the reading time can be obtained. So the candidate clocks can be started with a dummy initial value— we chose it to be iT— which will be corrected by an appropriate *adjustment*, $J^{i,l}$, applied at a (short) later time.

Srikanth and Toueg have shown [28] than no clock synchronization algorithm can achieve a rate drift better than that of the underlying physical clocks. Thus, optimal rate is approached by adjusting $J^{i,l}$ by the physical clock of one of the correct processors. For internal synchronization, to ensure that a value in the correct envelope of time is chosen, the median clock value should be selected. To understand why this is so, remember that there must be at least $2f_p+1$ processors, for f_p faulty clock-processor pairs. In consequence, the *select* function in line 71, giving in result the *median* of the clock values in $(\mathcal{V}_q^e)$, returns a correct value.

The adjustment depends on the chosen candidate l, and is applied after the election procedure. We have seen that the first processor q detecting the eligibility of a candidate clock e, invokes the election protocol. It extracts, from the clock-readings

For every processor q

```
         /* $r_q \geq 0$ is next local synchronization round at $q$ */
05    $r_q = 0$; $\forall i$, $\mathrm{Start}^i = \emptyset$;
06    bcast ($\langle start,\ 0,\ q\rangle$) od
......
10    when ($vc_q^{r_q-1}(t) = r_q T \wedge r_q \neq 0$) do
11       bcast ($\langle start,\ r_q,\ q\rangle$) od
20    when message $\langle start,\ i,\ p\rangle$ received from processor $p$ at real time $t_q^p$ do
21       $cc_q^{i,p}(t_q^p) = iT$; $vc_q = readClock(t_q^p)$;
22       $\mathcal{C}_q^p = \mathcal{A}_q^p = \mathcal{F}_q^p = \mathcal{V}_q^p = \emptyset$; $\mathcal{P}_q^p = \mathcal{P}$; $r_p = i$;
23       insert $p$ in $\mathrm{Start}^i$;
24       if # $\mathrm{Start}^i$+ # $\mathrm{Start}^0 > f_p$
               /* $\mathrm{Start}^0$ is formed by joining processors */
25          then  bcast ($\langle candidate,\ i,\ p, vc_q\rangle$)
26          else bcast ($\langle notsure,\ i,\ p, vc_q\rangle$) fi
27    od
30    when message $\langle candidate,\ i,\ l, vc_p\rangle$ received from processor $p$ do
31       insert $p$ in $\mathcal{C}_q^l$;
32       insert $vc_p$ in $\mathcal{V}_q^l$ od
40    when message $\langle notsure,\ i,\ l, vc_p\rangle$ received from processor $p$ do
41       insert $p$ in $\mathcal{A}_q^l$;
42       insert $vc_p$ in $\mathcal{V}_q^l$ od
50    when $vc_q^{r_q-1}(t) = vc_q^{r_q-1}(t_q^l) + (1+\rho_p)\Gamma^{max}$ do
51       $\mathcal{F}_q^l = \mathcal{P}_q^l - \mathcal{C}_q^l - \mathcal{A}_q^l$ od
60    when $\exists p, n > f_o$ : $\forall k, 1 \leq k \leq n, p \in \mathcal{F}_q^{b_k}$ do
61       $\forall u$, remove $p$ from $\mathcal{P}_q^u$ od
70    when $\exists e$ : $\mathcal{C}_q^e \neq \emptyset \wedge \mathcal{C}_q^e + \mathcal{A}_q^e = \mathcal{P}_q^e$ do
71       $vc$ = select ($\mathcal{V}_q^e$);
72       $J^{r_e,e} = vc - r_e T$;
73       bcast ($\langle election,\ r_e, e, J^{r_e,e}\rangle$) od
               /* start election protocol */
80    when election result is $(i, l, J^{i,l})$ do
81       if $\exists cc_q^{i,l}$ then
82          $vc_q^i = cc_q^{i,l} + J^{i,l}$ fi
               /* it is assumed it had launched $cc_q^{i,l}$ */
83       $\forall j, k$, terminate candidate clocks $cc_q^{j,k}$;
84       $\forall j$ $\mathrm{Start}^j = \emptyset$;
85       $r_q - 1$ =int($vc_q^i/T$);
86       if $vc_q.type$ =$init$ then  $vc_q.type$ =$int$
87    od
```

Figure 5. Basic clock synchronization algorithm

vector corresponding to e's broadcast for round j, the clock value chosen by the *select* function, and computes the adjustment $J^{r_e,e}$.

Let us observe function *select* for a moment (figure 7), for the steady-state internal synchronization case— that is, the system is already initialized— and with all clocks internal (type=*int*). The function selects the median vc of the clock-readings vector $\mathcal{V}$— let us suppose, the reading of the clock of processor x. That value is used to compute the adjustment, which is sent along with e's proposal (lines 71-73). Applying the adjustment obviously consists of subtracting the value with which the candidate clock was initialized in consequence of a tight broadcast, r_eT, and adding the chosen clock value vc— that is, $vc_x(t_x^e)$, read upon the same broadcast by x.

Whichever candidate is chosen, when the election is over there will be agreement on the candidate clock l for round i, and on the adjustment $J^{i,l}$ (line 80). The adjustment is applied to the instantiations of candidate clock l at each processor (l.82) and the resulting clock, vc^i, takes effect as the new virtual clock at each processor. The local round variable is updated (l.85) from the value of vc_q^i. The condition that there is a candidate clock launched (l.81) is a safeguard for initialization, that we will discuss later in the text, in steady-state operation it is always true. It should be noted that the same adjustment must be applied to all clocks, otherwise *precision* will be affected. At this time, the synchronization is over, all candidate clocks are terminated and Start^i is cleared (l.83-84).

During the re-synchronization interval, all correct virtual clocks drift from real time at the rate of their underlying physical clocks (thus, following the optimal rate). At each re-synchronization, virtual clocks are adjusted by one of the correct virtual clocks. However, there is a window of uncertainty equal to the Tightness interval, which is at most $\Delta\Gamma_{tight}$. In consequence, virtual clocks can deviate from the optimal real time envelope at every re-synchronization interval, by $\Delta\Gamma_{tight}$. Considering that $\Delta\Gamma_{tight}$ is much smaller than the re-synchronization interval, this does not represent in relative terms a significant deviation from the "optimal accuracy" featured by Srikanth's algorithm. A precise formulation of how much clocks deviate from the optimal real time envelope (essentially $\Delta\Gamma_{tight}$) is presented in appendix. We also show how to transform a succession of virtual clocks in a continuously adjusted clock, and achieve a bounded instantaneous rate, thus preserving monotonicity.

4.4. Achieving accuracy

We have just discussed internal synchronization with the a posteriori agreement protocol. In the context of CesiumSpray , this is the fallback situation when no external clock is available due to failures, and shows that the algorithm preserves accuracy during that period.

However, CesiumSpray is specified in a way that there should always be a correct external clock available per local network, to achieve accuracy towards an absolute reference. Referring to figure 3, that role is played by the GPS-nodes, which must

Function *readClock* at processor q

/* Consider that the clock-reading variable, vc_x, as per the algorithm in figure 5,
is indeed a pair $\langle value, type \rangle$, where *value* is the clock value,
type is one of *ext* or *int* or *init*, external or internal or initial clock.
vc_x with the extension omitted means its value. */

```
function readClock($t_q^p$) begin
  if $vc_q.type = ext$
    then $vc_q.value = vcg_q(t_q^p)$
    else $vc_q.value = vc_q^{r_q - 1}(t_q^p)$
  fi
  return $vc_q.value$;
end
```

Figure 6. Code of the readClock function

be provided with fault-tolerance. We have considered two situations: arbitrary failure external clock; weak-fail-silent external clock.

Arbitrary-failure external clock

This situation maintains exactly the same failure modes hypothesized in section 3 for the architecture: that is, clocks may do arbitrary failures. As such, given f_g external clock failures, there must be at least $2f_g + 1$ external clocks— that is, as many nodes with GPS receivers— per local network, so that a correct reading is always obtained.

Weak-fail-silent external clock

The arbitrary clock assumption is nevertheless demanding on the number of GPS-nodes. The intrinsic quality of GPS receiver clocks is normally high. Other works based on external clocks have considered the master nodes, i.e. the clock-processor pairs with external clocks, as being fail-silent or crash-on-failure [4], [17].

In consequence, we also study the situation where GPS-nodes (as clock-processor pairs) exhibit weak-fail-silent behavior. Each can do at most f_o omission failures, before being considered faulty, but it always supplies a correct and timely value. As such, given f_g external clock failures, there must be at least $f_g + 1$ external clocks. Remember that the algorithm (cf.§ 4.2) withstands f_o omissions from each of the failed processors, before eliminating it. So a GPS-node doing at most f_o omissions will still be read and included in the clock-readings vector.

Spraying external time

It is now the time to explain an important technicality. We assume that the GPS receiver clock is yet another virtual clock, vcg_q, besides the GPS-node's physical (pc_q) and virtual software (vc_q) clocks. The virtual GPS-clock (that we call just GPS-clock from now on) is "adjusted" by the NavStar system, and only read by the synchronization protocol. The virtual clock proper, is adjusted by the latter protocol, and is the user clock.

Normal node's clocks, represented by variable vc_q, are of type=*int*. GPS-nodes, on the other hand are, in principle, permanently externally synchronized: their vc_q clocks are of type=*ext*. However, when a GPS-node is not receiving from the NavStar satellites, due to failure or any other reason, we say that the local clock becomes of type=*int*. The toggle between types is represented in lines 90-93 of figure 8.

Now we can explain how the *readClock* function works, as coded in figure 6. Its purpose is to log the time at which a *start* message arrived, t_q^p (see lines 20-21 of figure 5), using one of two clocks: if clock type=*ext*, then the value of vcg_q is put in the local clock-reading variable, vc_q (lines 01-02); otherwise, the value of the running virtual clock, $vc_q.value = vc_q^{r_q-1}(t_q^p)$, is put instead (l.03).

The *select* function presented in figure 7 is now studied in detail. It will operate on the clock-readings vector supplied by the *readClock* functions in all processors. First, it tries to locate external clocks (l.01). If affirmative, it selects their values (i.e. of type=*ext*) in the clock-readings vector and picks the median (l.02) which, given the fault assumptions made above, is certainly a correct time "sprayed" from the GPS satellites. External time is then further sprayed into the other nodes, by forcing them to follow the selected GPS-clock in the adjustment they must apply, as discussed in section 4.3. If no external clocks exist, the select function picks the median of the internal clocks, with the protocol proceeding in internal synchronization.

The reason why we separate the GPS-clock from the physical or virtual clock of a GPS-node may be obscure at first sight. However, note that if we did not, the re-entering GPS-clock would make the local physical— and thus the virtual— clock jump, introducing continuity and/or monotonicity violations in the time service of the local machine. The virtual clock at a GPS-node is adjusted just as any other clock. By coincidence, the adjustment may come from its own GPS-clock, but it may also come from another one of the fault-tolerant set of GPS-nodes.

An interesting side effect of this mechanism as embedded in the a posteriori agreement protocol, is that it provides a good mechanism for a system administrator to force a manual setting of the global clock in a local network, when working exclusively in internal synchronization. Using administrator privileges, he/she "promotes" the clock of the machine he/she is working on to *type=ext*, and forces a value onto the clock. That value will be used to synchronize the global clock in the next synchronization round, since the rest of the clocks are internal.

Function *select* at processor q

/* Remember that the clock-reading variable, vc_x, is a pair $\langle value, type\rangle$.
Consider $\mathcal{V}$ as an array of such pairs. */

```
00        function select($\mathcal{V}$) begin
01          if $\exists k \in \mathcal{V} : k.type = ext$
02            then  $vc$ = median ($\{k.value | (k \in \mathcal{V} \wedge k.type = ext)\}$)
03            else if $\#\{k | (k \in \mathcal{V} \wedge k.type \neq init)\} \geq 2f_p + 1$
04              then  $vc$ = median ($\{k.value | (k \in \mathcal{V} \wedge k.type \neq init)\}$)
05              else  $vc$ = median ($\{k.value | k \in \mathcal{V}\}$)
06            fi
07          fi
08          return $vc$;
09        end
```

Figure 7. Code of the select function

4.5. Initialization and integration

For clarity, we have postponed the problems associated with the border conditions until this section: initializing the system, integrating new or recovered processors. *Initialization* is performed without changing the algorithm as presented in figure 5. A few lines of code, presented in figure 8, are appended. We have chosen the line numbers so that the code in both figures can be merged into the final algorithm. So, in mentioning the line numbers we refer to either figure.

A processor initializing itself starts an initial virtual clock, $vc_q^{-1}(t) = 0$ (l.00), and sets the current round to $r_q = 0$ (line 05). Depending on whether it has GPS connectivity or not, it sets the type of the local clock-reading variable (lines 01-03). Internal clock nodes will always have $GpsRx = nok$. However, note that if they are initial clocks, they are denoted of *type=init*, instead of *type=int*. Initialization concludes by sending a $\langle$start, 0, $q\rangle$ message (l.06).

If it is a system start-up, there will be several $\langle$start, 0, $q\rangle$ messages being sent. The synchronization procedure, though for round 0, is the same as laid down in figure 5. When there are enough processors to initialize the system, the first virtual clock is launched. Needless to say, if there are no external clocks in the clock-readings vector, the computed adjustment $J^{i,q}$ will follow the median of the initialized clocks, which started from 0.

Integration concerns the case where a new or recovered processor is joining the system when the latter is in operation. However, the joining processor does not know it, so it executes its initializing steps as just described in the last paragraphs. Notice that the algorithm is resilient to the situation where a running processor is requested to launch a candidate clock for a round different than its own. The round per se does not influence the final value of the clock, and on the other hand, this may

For every processor q

```
/* Let $GpsRx$ be the state of GPS reception, either $ok$ or $nok$ */
$vc_q^{-1}.value = 0$;
if $GpsRx = ok$
   then $vc_q.type = ext$
   else $vc_q.type = init$
fi
......
......
when $GpsRx \rightarrow nok$ do
   $vc_q.type = int$ od
when $GpsRx \rightarrow ok$ do
   $vc_q.type = ext$ od
```

Figure 8. Clock initialization and control

happen in a legitimate situation of system operation: (i) the system has stopped in round i, since there are not enough processors; (ii) a number of processors re-enter simultaneously, enough to bring their total number over $2f_p + 1$ again, and they start from round 0.

However, most of the times the joining processor will be isolated and will not succeed at generating a synchronization event. The joining processor sends response messages as specified by the algorithm, when the next round starts. If prior to integration the virtual clock of the joining processor has diverged from the others, its value is naturally discarded during the computation of the adjustment. Nevertheless, it will adjust its clock correctly in response to the election protocol.

Since a processor may join the system during a re-synchronization round i, it may not be able to collect enough information to do a correct election, or an election at all. Whatever the election protocol used, the only condition is that: *the newcomer should only install a clock of which it followed the complete election procedure.* In this case, it only installs the virtual clock resulting from the election if it had started the corresponding candidate clock (cf.fig.5, line 81). Otherwise, it remains in round 0 until the next round.

Absolute time at initialization

If a valid GPS-node clock is present during initialization, it will reply to start messages with the value of its GPS-clock. Following the selection function depicted in figure 7, it will force the adjustment to take the value of the GPS-clock. The same will happen if, for some reason, a local network had been left without external time for while: when its GPS-node is re-integrated, it will again force the clock set to follow external time.

A final aspect of absolute time at initialization is that, once in steady-state, the clock system should not be disturbed by the entry of new clocks (exception made to external clocks). A massive entry of new processors, in the absence of external clocks, might provoke the selection of initial clocks, instead of running clocks[14]. This is the reason why initial clocks are denoted of *type=init*: the selection function ignores them if there are enough clocks in the system (cf.fig.7, lines 03-04).

5. Improved a posteriori agreement protocol

In what follows, we present a particular materialization of our algorithm, with the support of a generic set of group communication tools. Group protocols and subsystems are gaining progressive acceptance for distributed systems builders and users. The tools used are specified in a way independent of any particular group-oriented system. They are amongst the simplest, and can be found in practically any 'groups' package.

To show the feasibility of this approach, we have implemented the a posteriori agreement algorithm as support of the time service of the xAMp group communications system[22]. The algorithm was implemented using some of the primitives offered by xAMp, which simplified the work and improved its efficiency. For the interested reader, the work is described with detail in [23]. The use of group tools to support the implementation of the *a posteriori* agreement algorithm has several advantages:

- processors that are participating in clock synchronization are transparently managed by group membership, addressing and communication; thus, to partake in the algorithm they just join the clock synchronization group; from then on, they are addressed by the communication protocol; failure detection and recovery are performed by the membership protocol;
- communication primitives can be used to generate and detect tight broadcasts; to mask omission failures; and to implement the election of the candidate clock.

We begin by presenting a general model of the group services needed, so that the reader be able to figure how to implement the protocol over any group support environment.

5.1. Group communication service model

There are a number of preferred qualities of the group communication system to be selected or implemented: synchronous; timer-driven; designed to be used over broadcast local networks; and portable.

The first two deserve some discussion. Synchronism is necessary to guarantee time bounded communication delay. Most synchronous group protocols are clock-driven [5]: they rely on the existence of a *clock*, in the sense of a global time-base.

To avoid having to deal with the recursivity or mutual dependence of the communication and clock synchronization protocols at initialization time, timer-driven— or (global-) clock-less— protocols are preferred. These are normally acknowledgement-based protocols, that rely on local timers [1], [22]. Their synchronism is achievable by relying upon real-time behavior (**BNP**4) and restrictive failure assumptions (**BNP**3) of the underlying network. The interested reader can find a more detailed discussion in [31].

The basic services needed are: *membership management* and *multicast communication*. The *membership* service supports the dynamic creation and 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. The *multicast* service allows a message to be delivered transparently to a group of processors, following pre-specified attributes (e.g. order, fault-tolerance).

To implement the *a posteriori* agreement algorithm, we need an *atomic* multicast service, and a reliable *transmit-with-response* service. We also need a basic group membership service to maintain the group of processors involved in synchronization. We detail their properties below.

Properties

GCP 1 (Group Membership) *Service providing: interfaces for processors to join or leave the clock synchronization group, G_{clk}; an address for transparently communicating with all currently correct G_{clk} members; a failure detection mechanism which automatically updates the membership of G_{clk}, based on the violation of the weak-fail-silent f_o-omissions assumption.*

In all primitives and protocol steps discussed hereon, messages are assumed to be addressed to G_{clk}, unless noted otherwise.

GCP 2 (Group View Change Indication) *Service ensuring that: the group membership is registered in a View, the current set of non-faulty elements; all View changes are indicated to all correct member processors in total order with sets of messages delivered by the communication service.*

Regardless of the type of communication ordering being used, change indications are signaled in the same order everywhere, relative to the stream of delivered messages. That is, before delivering a new View, it is ensured that all processors received the same messages since the last View. This way, processors have a consistent view of the group. In consequence, if total order is being used (see atomic multicast below), Views will be totally ordered with every other message.

For simplicity, in the protocol description of figure 9, the group is assumed to be already formed. Likewise, failed processors are assumed to be removed from the View as detected. This has implications in communication, as detailed below.

GCP 3 (Reliable Transmit-with-Response) *Service that ensures that a message is delivered to all surviving group member processors and responses from them are received in a single round: if the sender does not fail; despite the occurrence of f_o omission failures; despite the occurrence of recipient failures.*

The *tr-w-resp* service can be seen as an interface to the underlying unreliable multicast (network-level) service, with automatic collection of responses, message retransmission when omissions occur, and membership maintenance. In particular, by retransmitting more than f_o times, it is able of masking the network omission degree— alleviating the user from explicit concern with omission failures— and of providing failure detection information to the membership service, since by the weak-fail-silence assumption, a network adapter which does not reply more than f_o times is failed.

The service has a send interface (denoted **tr-w-resp**), and a reply interface (denoted **response**). For simplicity, the number of tries is not specified, but assumed to be greater than f_o.

The particular aspect of *tr-w-resp* as we define it, is that it seeks a successful round of: one transmission and all responses to it. That is, the message should be delivered by a *tight broadcast*. When failures occur, it discards previous responses, and retries until it gets a round with all responses from the surviving processors. In systems where a *tr-w-resp* is not implemented, it can easily be built on top of the local network datagram service. For the interested reader, we give a sketch of the *tr-w-resp* protocol in appendix.

GCP 4 (Atomic Multicast) *Service that ensures a message is delivered to all or none of the surviving group member processors: in total order; even if the sender fails; despite the occurrence of f_o omission failures; despite the occurrence of recipient failures.*

The *atomic multicast* is used to execute the election protocol.

Clock synchronization can be implemented using the group package as follows (see figure 9). To avoid unnecessary repetition, we will focus essentially on the way groups help in the implementation of the algorithm.

5.2. Generating, detecting and agreeing on a tight broadcast

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 (lines 20-21). By definition of *tr-w-resp*— namely the way it recovers from failures— the generation of a tight broadcast is assured if the sender does not fail. Furthermore, its detection is straightforward, since after execution of *tr-w-resp* the bag R_q has the responses of all correct processors— failed ones are excluded. This is the innovative part: with group support, the sender is able to detect its own broadcast with the group communication primitives, and have the failed processors automatically excluded by the group membership. It represents a simplification in

```
                    sender part (for each member $q$)

        /*Let $\mathcal{P}$ be the set of correct processors, all in membership of group $G_{clk}$.
          Let $G_{clk}$ be the group used for clock synchronization.
          $\mathcal{P}$ is automatically updated by the group support, in case of join, leave, failure, recovery.
          Let $\mathcal{R}_q$ be a bag of responses collected by $q$, of the form $\langle$ nature,value $\rangle$ (e.g. see line 44).
          $r_q \geq 0$ is next local synchronization round at $q$ */

20      when $vc_q^{r_q-1}(t) = r_q T$ do
21        tr-w-resp ($\langle$ start, $r_q, q\rangle, \mathcal{R}_q$);
22        if ($\exists resp \in \mathcal{R}_q$ : resp.nature $= candidate$) $\wedge$ ($vc_q^{r_q}$ = none) then
23          $vc$ = select ($R_q.value$);
24          $J^{r_q,q} = vc$;
25          atomic ($\langle$ install, $r_q, q, J^{r_q,q}\rangle$) fi
26      od

                    receiver part (for each member $p$)

30      Start$^i = \emptyset$; $vc_p^i$ = none;
40      when $\langle$ start, $i, q\rangle$ message received from processor $q$ at real time $t_p^q$ do
41        $cc_p^{i,q}(t_p^q) = 0$; $vc_p = readClock(t_p^q)$;
42        insert $q$ in Start$^i$;   /* insertion is idempotent in case of repetitions */
43        if # Start$^i > f_p$
44          then  response ($\langle$ candidate, $vc_p$ $\rangle$) to $q$
45          else  response ($\langle$ notsure, $vc_p$ $\rangle$) to $q$ fi
46      od
50      when $\langle$ install, $i, q, J^{i,q}\rangle$ message received from processor $q$ $\wedge$ $vc_p^i$ = none do
51        if $\exists cc_p^{i,q}$ then
52          $vc_p^i = cc_p^{i,q} + J^{i,q}$ fi  /* only installs when it had launched $cc_p^{i,q}$ */
53        $\forall j, k$, terminate candidate clocks $cc_p^{j,k}$;
54        $\forall j$, Start$^j = \emptyset$
55        $r_p - 1$ =int($vc_p^i/T$);
56        if $vc_p.type = init$ then  $vc_p.type = int$ fi
57      od
60      when $\langle$ install, $i, q, J^{i,q}\rangle$ message received from processor $q$ and $vc_p^i \neq$ none do
61        nop od   /* reception is just acknowledged via group protocol */
```

Figure 9. Improved clock synchronization algorithm

the protocol, which also contributes for the reduction in the number of processors needed to do fault-tolerant synchronization.

5.3. Achieving precision

Remember that at least f_p+1 $\langle start \rangle$ messages must be issued, to validate a correct synchronization point. Each recipient, when it receives a $\langle start \rangle$ message, includes the sender in $Start^i$ and tests its cardinal (lines 42-43). The $\langle start \rangle$ messages received from the first f_p processors are considered not safe, thus, a $\langle notsure \rangle$ response is returned (l.45). The remaining messages are acknowledged with $\langle candidate \rangle$, meaning that start messages from at least f_p other processors have been previously received (l.43-44).

In any case, a *candidate* clock is started for each $\langle start \rangle$ message received (l.41). Note that with the group-oriented implementation, responses are sent only to the sender of the $\langle start \rangle$ message, since detection of a tight broadcast is performed by the sender of *tr-w-resp*. Note also that if a $\langle start \rangle$ message is retransmitted, the associated clock is just started again (l.41), but the start-message count (l.42) is not incremented, since the sender is the same.

The sender waits for the termination of the *tr-w-resp* procedure. If at least one processor responds with $\langle candidate \rangle$, the sender can safely assume that it started a timely candidate clock (line 22). Upon computing an appropriate adjustment, it multicasts an *install* instruction using the *atomic* communication protocol (l.25). Usually, several candidate clocks will be started and, in consequence, several *install* messages may be received by every member. Since *install* messages are sent in total order, the first *install* message delivered selects the candidate clock to be used by all, during the next synchronization interval (lines 50,52).

This is a simple way of performing the election of the candidate clock, yielded by the 'groups' package. All other candidate clocks are simply discarded, and Start^i initialized (l.53-54). Note that if a processor crashes before sending an *install*, its candidate clock will be discarded as soon as an *install* arrives from a correct processor.

5.4. Preserving and achieving accuracy

Preserving and achieving accuracy is done much in the same way as for the basic a posteriori protocol depicted in figure 5. Note that this time we chose candidate clocks to be initialized with zero (line 41), in order to exemplify, as we had previously explained, that the value of the candidate clock does not matter until it is installed (it only changes the computation of the adjustment). Once the sender detects its own candidate to be eligible (l.22), it picks the clock to adjust from, with the *select* function (fig. 7), from the clock-values part of the bag of responses, ($R_q.value$), which is nothing else than the clock-readings vector of the basic algorithm. The sender computes the adjustment, and tries to install its candidate clock, using that adjustment (lines 24-25). After successful delivery of the first atomic multicast, each processor (including the sender) will have installed the associated candidate clock— with the relevant adjustment— as the new virtual clock (line 52).

It is possible that other $\langle install \rangle$ messages arrive afterwards (l.60): they are simply ignored, although acknowledged by the lower-level group support. Likewise, if a sender has already installed a clock for the next round (i.e. $vc_q^{r_q} \neq$ none), it does not try to install its own candidate (line 22), refraining from doing a useless atomic broadcast.

5.5. Initialization and integration

The membership information provided by the 'groups' package is extremely useful for initialization and integration. Namely, the property that joins and leaves are totally ordered with respect to atomic messages, allows processors to have a mutually consistent view of the group. This drastically improves on the basic algorithm, in two ways:

- the set of processors $\mathcal{P}$ no longer needs to be known a priori and static— processors enter and leave the group as they come up or fail;
- processors initializing or reintegrating can do it in a more disciplined way than in the basic algorithm, saving network resources— they rely on membership information for measuring the fault-tolerance threshold, instead of going blindly into the network as they initialize.

Every time a processor joins clock synchronization a new membership *view* is provided to all members, including to the new member. From the membership information, the newcomer may infer the state of the group. Let $N = \#\mathcal{P}$ be the number of group members after a join: if $N < 2f_p + 1$, there is still no quorum to start the synchronization service and all members simply keep awaiting for other processors to join. If $N = 2f_p + 1$, the newcomer has just formed the required quorum, thus all members start the clock service by broadcasting $\langle start \rangle$ messages for the first synchronization round.

Finally, if $N > 2f_p + 1$, 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, the newcomer may enter in the middle of a round. In that case, the condition laid down in the previous section must hold: *the newcomer should only install a clock of which it followed the complete election procedure.* Given the election method followed in the protocol of figure 9, the easiest safe way is the following:

- always participate in the algorithm, replying to *start* messages;
- wait to see an *install* for one round, and then only install the next round's clock.

The clock initialization and control code in figure 8 also applies to the improved algorithm code in figure 9.

Some membership services have a property that can be used to speed-up integration of processors: group-dependent state information can be transferred to the joining processor during the join operation [1], [22]. This feature can, if desired, be used to inform the newcomer of the state of synchronization activity, and reduce the worst-case delay in processor integration— which is approximately $2T$— with small modifications of the protocol.

6. Dynamic fault-tolerance and performance adaptation

In the practical use of the a posteriori agreement protocol in CesiumSpray , two problems arise in the following situations: (i) when the number of processors is or goes below the threshold number needed for fault-tolerant operation of the protocol, to a given set of failure assumptions; (ii) when too many nodes exist in the local network, risking loading it unnecessarily during synchronization.

An important advantage of using group support 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.

What we have discussed so far is a *static mode* of operation. The user specifies at design time a desired fault-tolerance degree, i.e., the number of clock and/or processor faults that need to be tolerated. As stated in section 4, the minimum number of nodes to tolerate f_p faults is $2f_p + 1$. The service ensures that the virtual clocks are only synchronized while the minimum number of nodes are available in the system. Secondly, regardless of their number, all processors participate in synchronization, generating start messages at the appropriate moments.

Alternatively, one could think of a *dynamic mode*, where continuity of service would be provided for any number of clocks in the system. Upon membership changes, in particular if some nodes are lost, the protocol would dynamically adapt and the synchronization service would report to the user the new (lower) fault-tolerance degree at which the service would be running. On the other hand, for any number of processors, the algorithm would only run with the minimum number necessary for fault-tolerant operation: above that threshold, additional processors would not issue start messages.

These cases can be represented by a generic parameterization, with the help of the group support package. When the clock service is started, the desired tolerated faults are specified as an interval $[f_{min}, f_{max}]$. The equivalent in the static mode, is $f_{min} = f_{max} > 0$. Upon system initialization, the clock synchronization service is only started when at least $2f_{min} + 1$ nodes are present (in the dynamic mode, if $f_{min} = 0$ 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). Identifying these extra processors is extremely easy with property **GCP**1 (group membership), and **GCP**2 (group change indiction).

When the number of processors in the system, N, lies within the fault-tolerance interval, i.e., $N \in [2f_{min}+1, 2f_{max}+1]$, the protocol runs in adaptive 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.

7. Accuracy preservation in internal synchronization

The weak-fail-silent clock model discussed in section4.4 allows the correct behavior of CesiumSpray with only one GPS-node per local network, in absence of failures. This hypothesis— illustrated in figure 3— is of economical interest, for applications with moderate fault-tolerance requirements. Moreover, CesiumSpray local networks can work with different fault-tolerance levels, with no impact on the functionality of each other.

However, the fault coverage is minimal: upon the GPS-node failure— or its clock— the system loses external time. In consequence, the objective of this section is to show how the system behaves in that abnormal situation. As a matter of fact, it can also occur in the operation of the fully-fledged CesiumSpray , if for some reason it loses all the GPS-nodes of a local network (e.g. reception problems).

The protocol is designed to be resilient to that situation. In each round, the *select* procedure tries to find external clocks (l.01-02). If no external clock exists, internal synchronization is performed in that round (l.03-05), along the lines discussed in section 4.3. This approach is not as fragile as it may seem. Given a desired accuracy of the time service, the system can withstand a specified duration of external clock unavailability. This bound is derived in section A.2.5 of the appendix, and the effectiveness of accuracy preservation of internal synchronization is discussed in section 8. The system will also gracefully re-integrate the GPS-node upon recovery, and re-inject external time, as seen in section 4.5. However, there is no guarantee that the normal bound on accuracy is secured, which may imply larger than expected adjustments when external time is recovered, with impact on the other bounds (precision and rate).

In consequence, this study should be understood as a discussion of resilience to assumption uncoverage, since the normal operation of CesiumSpray , and all the derivations made in appendix, assume that the correct clocks always remain within the specified precision and accuracy bounds. Extending the work to other situations is a matter of further study.

8. Evaluation

The merits and demerits of CesiumSpray are analized in this section, under several viewpoints: cost— in processors; quality of the synchronization parameters— precision, rate and accuracy; and resilience to uncoverage of assumptions.

Scenario	Basic Algorithm	Improved Algor.
CORRECT SYNCH. POINT AND CLOCK READING	$2f_p + 1$	$2f_p + 1$
ACHIEVE/DETECT SUFF. EVIDENCE (UNCONTROLLED-OMISSION PROCESSORS)	$(f_o + 1)(f_p + 1) + f_p$	$2f_p + 1$
ACHIEVE/DETECT SUFF. EVIDENCE (FAIL-SILENT PROCESSORS)	$f_o + f_p + 1 + \max(f_o, f_p)$	$2f_p + 1$

Figure 10. Bounds on number of processors— basic and improved algorithm

8.1. Cost

The cost of our algorithm can be assessed in terms of bandwidth and number of processors. The bandwidth used depends on the period of re-synchronization and in the number of processors. The broadcast nature of the network, the use of group (multicast) communication, and the possible use of the adaptive mode, reducing the number of processors involved in starting each round, render the cost in bandwidth low.

More important is the number of processors required. We recall that the minimum number of processors needed to run every round successfully in the basic algorithm is $(f_o+1)(f_p+1)+f_p$. The bounds can be reduced for stronger failure assumptions. These bounds are listed in figure 10. A proof is given in appendix. It is easy to see that the generic bounds are dominant over the $2f_p + 1$ bound— needed for fault-tolerant operation of the *select* function on the clock-readings vector. That is, if the former hold, the latter always holds, for any value of f_p and f_o.

With the improved algorithm, new bounds occur. They are also listed in figure 10: the $2f_p+1$ bound still holds to guarantee a correct synchronization point and clock reading; given that the group protocols recover from omissions, $2f_p + 1$ is also the bound for achieving and detecting sufficient evidence.

8.2. Quality

The consolidated figures of merit for the algorithm are given in the table of figure 11. They are more detailed in the appendix, namely in what concerns the definition of variables. Rate drift, ρ_{ap}, is given by ρ_v in lemma 10. Local precision of the system, δ_l, is given by δ_v in lemma 6. Values for accuracy preservation in internal synchronization are given in the table of figure 12, for several scenarios,

Prot.	**Parameter**	**Expression**	**Value** [μs]
AP	Convergence	$\delta_{ap} \geq (1+\rho_p)\Delta\Gamma_{tight} + 2\rho_p\Gamma^{max}_{agreem} + g$	101
AP	Rate drift	$\rho_{ap} \geq \rho_p + \frac{\Delta_{vc}}{\Delta_{spread}}$	$4.4/s$
AP	Local Precision	$\delta_l \geq \delta_{ap} + [(1+\rho_p)\Delta\Gamma_{tight}]+$ $+2\rho_p[(1-\rho_p)^{-1}(T + J^{max}) + \Gamma^{max}_{start} + \Gamma^{max}_{agreem}]$ with period $T = 150s$	500
CS	Global Precision	$\delta_{CS} \geq 2.\alpha_{CS}$	1000
CS	Global Accuracy	$\alpha_{CS} \geq \alpha_g + \delta_l$	500

Figure 11. Typical figures of merit of the CesiumSpray protocol: AP- a posteriori internal synchronization; CS- CesiumSpray global external synchronization

whose expression is derived from lemma 13. We are considering, for system figures, continuously adjusted clocks. We recall that $\alpha_g = 100ns$. The granularity of the clocks is $g = 1\mu s$.

We now analyze the expressions, and discuss how the results can be improved. We also give experimental figures: the values in the rightmost column are derived from performance measurements and from estimates, made to provide some insight on the possibilities of CesiumSpray in real systems.

From the table, it is obvious that the quality of the a posteriori agreement protocol is practically dictated by $\Delta\Gamma_{tight}$, whose components (cf.§3.2) are:

$$\Delta\Gamma_{tight} = \Delta\Gamma_{prp} + \Delta\Gamma_{rec}$$

The *receive* error, in a real-time kernel, may be practically canceled with local network co-processors and interrupt treatment, provided that there is an upper bound for message reception interrupt service latency and that the bound is small[15]. In the limit, 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).

Compared to the *receive* error, the *propagation* error is normally negligible. However, in architectures where programming inside the real-time kernel allows bringing the receive error down to the order of magnitude of the propagation error, the latter can be further reduced by using network mapping techniques [24], which allow to compute individual inter-node propagation times and use them to correct the message delay expression.

Accuracy preservation, which means the capability of maintaining accuracy to real time, by following a correct hardware clock in internal synchronization— i.e.

verifying an envelope rate (cf. §2)— cannot be optimal in the terminology of S-rikanth & Toueg[28], but is very close to it. The worst possible correct clock will form a bound of the rate drift envelope. The algorithm will, in worst-case, synchronize by that clock and deviate to the outside of the envelope at most by the measure of $\Delta\Gamma_{tight}$ which, as just discussed, can be made very small. Due to the good accuracy preservation of the a posteriori algorithm, CesiumSpray is capable of operating during periods of shortage of external time. This happens when the last GPS-node of a local network— or just the GPS reception— fails for some reason.

We base the values of figure 11 on measured performance, in a real scenario, of the implementation of the *a posteriori agreement* internal clock synchronization algorithm provided by the group support protocol suite xAMp [23]. The results were collected on a real-time distributed system formed of communication boards developed for the DELTA-4 project [20]. The protocol ran on a real-time executive (the SPART[16] kernel), over Motorola 68020 CPUs and accessing a token-bus network. We concentrated on measuring $\Delta\Gamma_{tight}$, which is the limiting factor of the precision achieved by our protocol. We have obtained values of $\Delta\Gamma_{tight} = [40 \ldots 100\mu s]$. The time to run the agreement was $\Gamma_{agreem} = [2 \ldots 100ms]$. We consider a conservative value of $\Gamma^{max}_{start} = 20ms$, and we estimated $J^{max} = 400\mu s$.

We have loaded our 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. With the drift rate considered for the physical clocks, $\rho_p = 10^{-6}$, even a conservative value of $100ms$ for the agreement time adds just $1\mu s$ to the algorithm precision.

In consequence, from the table of figure 11 we can deduce excellent figures of merit for CesiumSpray . We targeted a local precision of $\delta_l = 500\mu s$, which yielded a synchronization period of $T = 150s$. For a synchronization period of $T = 30min.$, we would have had $\delta_l = 4ms$. Better can be done, with faster kernels/CPUs, which spend less time within system calls— and the respective uninterruptible critical sections. The SPART platform's average primitive execution time was of the order of $100\mu s$. Current platforms can easily be faster by one order of magnitude, with the expected impact in the *receive* error, and thence in $\Delta\Gamma_{tight}$.

As for accuracy preservation, figure 12 shows the allowed period of external time outage, T_{out}, before a specified accuracy is lost, for several scenarios. We considered two starting situations: (a) the worst case, when the clock has already deviated from real time as much as α_{CS}, and will continue to be fast[17]; (b) the average case, when the clock is synchronized with real time. For each situation, we evaluated the outage duration for two target accuracies: the standard, α_{CS}; and a degraded-mode one, ten times the standard. The standard accuracy cannot be guaranteed for situation (a). The derivation of the expression of T_{out} is in appendix. For situation (b), we substituted 0 for α_{CS} in the expression. The results in the table confirm the good behavior of CesiumSpray in the presence of temporary external time failure.

Initial Accuracy $[vc_k(t_0)]$	**Target Accuracy** $[\mu s]$	**Allowed Outage Duration** $[s]$
$t_0 + \alpha_{CS}$	α_{CS}	—
	$10.\alpha_{CS}$	1034
t_0	α_{CS}	115
	$10.\alpha_{CS}$	1150

Figure 12. Accuracy preservation of CesiumSpray in internal synchronization

This is especially useful for the weak-fail-silent external clock model, with only one GPS-node per local network (cf. § 4.4).

8.3. Resilience to assumption uncoverage

Assumptions about the operational environment of a system allow designers to build a protocol and reason about its correctness, fault-tolerance degree, and so forth. In consequence, practically all reasoning is directed to demonstrate that: (i) the assumptions are realistic; (ii) the protocol operates correctly to those assumptions. However, it is reassuring to analyze what happens should things go very wrong, if one is intending to build a real system.

That analysis has been informally made for the *a posteriori agreement* protocol. We have not gone through all possible violations, but chose instead the few cases that are more likely to occur. The analysis focused on the situations where a single class of faults occurs, and the assumptions about that class are violated. It showed the protocol to be reasonably resilient to those cases:

1. maximum communication delays, Γ^{max}_{start} or Γ^{max}_{agreem}, exceeded;
2. $\Delta\Gamma_{tight}$ exceeded;
3. maximum number of failed processors, f_p, exceeded;
4. maximum number of network omission failures, f_o, exceeded.

In situation 1, the algorithm executes correctly, but precision and rate may be violated. The deviation is itself very small, maximum in the order of $[\delta(\Gamma^{max}_{xxx}) \times \rho_p]$. Fault-tolerant execution of the *select* function— that is, the guarantee that a correct clock is selected— may be compromised: in result of the violation of Γ_{start}^{max}, an election may be started (and won) before all readings come, so that the clock-readings vector does not have enough processors $(2f_p+1)$. The protocol is resilient to this violation, as long as the number of unaffected processors is greater than $2f_p$.

In situation 2, precision may be violated in the direct proportion of $[\delta(\Delta\Gamma_{tight})]$. The necessary and sufficient condition is that it occurs in the broadcast winning the election. The effect of this violation is important, so care should be taken in measuring/assessing $\Delta\Gamma_{tight}$.

In situation 3, given the single fault class assumption, the protocol only blocks if the number of processors failed in excess leaves the system with less than f_p+1 processors. The fault-tolerant execution of the *select* function is also secured, while the number of surviving processors is above $2f_p+1$. For $f_o \neq 0$, this is trivial for the basic algorithm. If the improved algorithm is working with only $2f_p+1$ processors, *select* may not be resilient to this violation. Its effect is also important, one safeguard is to have more processors than the minimum bounds.

In situation 4, such a violation can lead to unjustful failure detection, if the violations affect a processor in manner that it looks like having done more than f_o omissions. The consequences are similar as underlined for situation 3, although this is a difficult pattern to occur— only probable if the network is in bad conditions.

8.4. Related Algorithms

The a posteriori agreement is effective even under non-negligible loads, provided that the conditions for **BNP 4** still hold. However, it only presents significant advantages when $\Delta\Gamma_{prp}$, $\Delta\Gamma_{rec}$ are small. $\Delta\Gamma_{rec}$ depends on the underlying operating system and hardware machinery: some systems may not allow a small and predictable preemption time for reception interrupts. The use of LAN bridges may originate high $\Delta\Gamma_{prp}$ values. In the context of internal synchronization only, *a posteriori agreement* can be used to improve synchronization of nodes in the same segment while global synchronization could be ensured, for instance, by a hierarchical algorithm, such as studied in [27]. However, combination with external synchronization in CesiumSpray obviously yields improved quality.

Our algorithm requires the execution of an election protocol. However, the time required to reach election has only a second order effect on the achieved precision. We consider the cost in traffic negligible, in face of the usually available local network bandwidths and given the benefit in precision and determinism, with regard to other approaches [4], [7]. Furthermore, our algorithm does not require any particular agreement protocol (as long as it exhibits bounded termination). Since most fault-tolerant distributed systems are local-network-based and implement some form of agreement protocol, that can be used to perform the election, our algorithm can be easily integrated in such architectures.

The hardware-assisted algorithm of Kopetz removes practically all components of the message delay variance, except the propagation error[11]. Our algorithm, without hardware support other than interrupts, only leaves an attenuated receive error besides the propagation error.

The work in [25] is similar to ours, emphasizing the same pseudo-hierarchical structure, and is also based on GPS time for external synchronization. However, it is more related to smaller-scale factory-level real-time networks. Unlike our approach, they exploit hierarchical synchronization, with arbitrary network topologies. They perform a form of internal clock synchronization called clock validation, whereby there may be sets of clocks with high and low accuracy in the network. Unlike ours, when external time is lost, the system is re-initialized.

9. Conclusions

A precise and accurate global time service for large-scale distributed systems was presented in this paper. The system is a mix of external and internal synchronization, laid down in a pseudo-hierarchical way. It uses the highly precise and accurate GPS time reference to inject external time in each local network. It then progresses further down, inside the local networks, with an algorithm with excellent precision enhancement and accuracy preservation qualities.

This architecture has several advantages. Using GPS, it is superior in price and reachability to radio-receiver-based schemes. Using a symmetrical protocol, it has better fault-tolerance characteristics, and a better determinism$\times$quality product than master-based protocols. Using the a posteriori agreement approach in internal synchronization, it captures the best of non-averaging and averaging families of protocols, by allying the time marker principle of the former to the clock-reading and agreement principle of the latter. In absence of external time, the a posteriori agreement protocol performs better in accuracy preservation and precision enhancement than known external-time-based protocols.

A posteriori agreement behaves best in networks with physical multicast capability, such as LANs and prevailing ATM technologies. Given that the architecture of large-scale computer networks seems to be very well represented by a WAN-of-LANs model, CesiumSpray achieves a proper balance between price, quality, and fault-tolerance, by requiring at least one GPS receiver per local network. Allied to this, the fact that the first level of the synchronization hierarchy short-circuits the computer communication infrastructure where it most compromises scalability— the WAN — is responsible for the good precision and accuracy in large-scale exhibited by CesiumSpray . Notice that these parameters are not influenced by the size of the WAN (neither geographical nor in number of nodes), and only very remotely by the size of LANs— by the almost negligible increase of $\Delta\Gamma_{tight}$ due to the increase in the *propagation* error with distance, and the agreement time with increased load— which gives CesiumSpray virtually unlimited scalability.

As future work, we plan to extend this system to work with arbitrary duration of external time outages, using adaptive bi-modal accuracy, whereby the system switches automatically from normal accuracy to a degraded accuracy when external time is lost, and back, with a mechanism ensuring smooth recovery.

Appendix

A.1. Transmit-with-response protocol

We present a sketch of a *tr-w-resp* protocol able of tolerating $f_o = nrTries - 1$ omission failures, and seeking for delivery in a tight broadcast. The protocol is given in [31]. It returns a bag *Resp* of expected *nrResponses* responses, either when it is full or when it performed *nrTries* tries. In the latter case, given the bounded omissions and weak-fail-silent assumptions, it will have all the responses from non-failed members. Inserted in the clock synchronization protocol, *nrTries* will be given by the f_o assumption, and *nrResponses* will be given by the G_{clk} View. Notice that line 52 empties the bag at each try, which ensures that the bag only contains responses of the same try, satisfying the tight broadcast condition when all are received.

transmit-with-response (⟨ data, nrResponses, nrTries, Resp ⟩)

```
50 tries := 0;    Resp := empty;
51 do tries < nrTries  $\wedge$ Resp $\neq$ full $\rightarrow$
52     Resp := empty;
53     Tx(data,$id_{tries}$);
54     waitRepliesPutInBag(TwaitReply, Resp);
55     tries := tries +1
56 od
57 return Resp;
```

A.2. Protocol Parameters

In this appendix we show the results relevant to parametrize the protocol. The proofs are inspired in those in [28]. For brevity, we only present the crucial proofs. Since clocks have a granularity g, all clock-value-related figures are multiples of g.

A.2.1. Assumptions

Assumption 1 *There is a known upper bound, Γ^{max}_{start}, on the time required for a ⟨start⟩ message to be prepared by a correct processor and sent to all correct processors and processed by the recipients of the message.*

Assumption 2 *There is a known upper bound, Γ^{max}_{agreem}, on the time required to elect a candidate clock, after the start of the first candidate clock. For the sake of convenience, the time required to detect the tight broadcast is also included in Γ^{max}_{agreem}.*

Assumption 3 *The difference, in real time, between the starting time of the same candidate clock, $cc^{i,n}$, in two different processors, is bounded by $\Delta\Gamma_{tight}$.*

A.2.2. Internal a posteriori agreement algorithm

A.2.2.1. Instantaneous clocks: precision

LEMMA 1 *At the end of the i^{th} resynchronization period, virtual clocks differ by at most $(1+\rho_p)\Delta\Gamma_{tight} + 2\rho_p\Gamma^{max}_{agreem} + g$. That is, for $i \geq 1$ and for all correct processors n and m, $\exists \delta_{ap} : |vc^i_n(end^i) - vc^i_m(end^i)| \leq (1+\rho_p)\Delta\Gamma_{tight} + 2\rho_p\Gamma^{max}_{agreem} + g \leq \delta_{ap}$.*

Proof: δ_{ap} is the precision enhancement measure of the algorithm. By definition 1 and assumption 3 the value of a candidate clock, at the moment it is started at any processor, differs from the value of the same clock at any other correct processor by at most $(1+\rho_p)\Delta\Gamma_{tight}$. By Assumption 2, $end^i - elected^i \leq \Gamma^{max}_{agreem}$. Given clock granularity, there may be a quantification error of g. Thus, by definition PC 2, $|vc^i_n(end^i) - vc^i_m(end^i)| \leq (1+\rho_p)\Delta\Gamma_{tight} + 2\rho_p\Gamma^{max}_{agreem} + g \leq \delta_{ap}$. □

LEMMA 2 *Given δ_{iv}, all correct processors start the elected candidate clock soon after a correct processor is ready to do so. Specifically, $elected^i - ready^i \leq (1-\rho_p)^{-1}\delta_{iv} + \Gamma^{max}_{start}$.*

LEMMA 3 *The correction Δ^i_{ck} to virtual clocks is bounded by a know constant, Δ^{max}_{ck}, that is $\Delta^{max}_{ck} \geq |\Delta^i_{ck}|\ \forall_i$. Specifically, $\Delta^{max}_{ck} \geq (1+\rho_p)[\Gamma^{max}_{start} + (1-\rho_p)^{-1}\delta_{iv}] \geq |\Delta^i_{ck}|\ \forall_i$.*

Assume the following two relations:

Assumption 4

$$(T - \Delta^{max}_{ck}) > (1+\rho_p)\Gamma^{max}_{agreem}$$

Assumption 5

$$\delta_{iv} > \delta_{ap} + \underbrace{2\rho_p[(1-\rho_p)^{-1}(T + \Delta^{max}_{ck}) + \Gamma^{max}_{start} + \Gamma^{max}_{agreem}]}_{\delta_{drift}}$$

Assumptions 4 and 5 specify the minimum and maximum values for the re-synchronization period, T, for a given desired precision, δ_{iv}. The term δ_{ap} corresponds to the result of the convergence function, that is, the real time difference between virtual clocks at the end of the synchronization round. The term δ_{drift} corresponds to the worst-case drift during the longest possible re-synchronization interval.

LEMMA 4 *The instantaneous clocks obtained through the algorithm in figure 5 verify the precision property, that is, $\exists\delta_{iv}$ such that:*

$$|vc_k(t) - vc_l(t)| \leq \delta_{iv}, \quad for\ \ 0 \leq t$$

A.2.2.2. Instantaneous clocks: envelope rate

In this section we give a measure of how much the clocks in internal synchronization deviate from the real time envelope.

Lemma 5 *For any execution of the algorithm, there exists a constant $\rho_{i\alpha}$, such that the instantaneous clocks of the algorithm in figure 5 verify the envelope rate property, that is $\exists \rho_{i\alpha}$ such that:*

$$\begin{array}{lcl} 1-\rho_{i\alpha} \leq & 1-\frac{(1-\rho_p)\Delta\Gamma_{tight}-\rho_p T}{T+(1-\rho_p)\Delta\Gamma_{tight}} & \leq \\ \leq & \frac{vc_k^i(t)-vc_k^0(0)}{t} & \leq \\ \leq & 1+\frac{\rho_p T+(1+\rho_p)\Delta\Gamma_{tight}}{T-(1+\rho_p)\Delta\Gamma_{tight}} & \leq 1+\rho_{i\alpha} \end{array}$$

Proof: The complete proof is omitted for brevity. We prove the upper bound. Let $E(t_0)$ be the set of executions of the algorithm in which $ready^1 = t_0$. Consider an execution $e \in E(t_0)$ in which $\forall_{k\geq 1}$, $elected^i = ready^i$, and let the correction for the elected clock be $\Delta_{ck}^{fastest}$. In the execution e the physical clock of processor n runs at the maximum possible rate, that is, $1+\rho_p$ with respect to real time. It is clear that execution e is possible. It can be shown that, for processor n, the interval of real time between consecutive re-synchronizations is $(T-\Delta_{ck}^{fastest})(1+\rho_p)^{-1}$. In this period its virtual time increases by T. It can also be shown that, for the same processor, the maximum correction, $\Delta_{ck}^{fastest}$, will be given by $\Delta_{ck}^{fastest} = (1+\rho_p)\Delta\Gamma_{tight}$. The lower bound proof follows similar arguments. $\rho_{i\alpha}$ should be chosen in order to satisfy both bounds. It is easy to see that this is achieved by having $\rho_{i\alpha}$ follow the upper bound expression. $\square$

A.2.3. Maintaining continuous clocks

Let $t_n^{a,i}$ be the real time when processor n accepts the i^{th} virtual clock. Instantaneous virtual clocks exhibit discontinuities at each re-synchronization since there is usually a nonnull difference between two consecutive clocks, given by: $vc_n^i(t_n^{a,i}) - vc_n^{i-1}(t_n^{a,i})$. Continuous clocks can be obtained if this difference is spread over a real time interval Δ_{spread}. We leave to the reader the proof that with this transformation, clocks continue to verify the precision and envelope rate properties. We prove the rate property.

Definition. A continuous virtual clock can be obtained from the instantaneous virtual clocks, using the following function:

$$vc_m(t) = vc_m^0 \quad \text{for} \quad t < t_k^{a,1}. \qquad (i)$$

$$vc_m(t) = vc_m^{i-1} + \frac{(vc_m^1(t_m^{a,i}) - vc_m^{i-1}(t_m^{a,i}))(t - t_m^{a,i})}{\Delta_{spread}} \quad \text{for} \quad t_m^{a,i} < t < t_m^{a,i} + \Delta_{spread}. \qquad (ii)$$

$$vc_m(t) = vc_m^i \quad \text{for} \quad t_m^{a,i} + \Delta_{spread} < t < t_m^{a,(i+1)}. \qquad (iii)$$

A.2.3.1. Continuous clocks: Precision

LEMMA 6 *The continuous clocks obtained by definition A.2.3 based on instantaneous clocks resulting from algorithm in figure 5 verify the precision property. More precisely:* $|vc_n(t) - vc_m(t)| \leq \delta_{iv} + (1+\rho_p)\Delta\Gamma_{tight} \leq \delta_v$.

A.2.3.2. Continuous clocks: Envelope rate

LEMMA 7 *Assume* $\rho_\alpha \geq \rho_{i\alpha}$. *The continuous clocks obtained by definition A.2.3 based on instantaneous clocks resulting from algorithm in figure 5 verify the envelope rate property, that is* $\exists \rho_\alpha$ *such that:*

$$1 - \rho_\alpha \leq \frac{vc_m(t) - vc_m(0)}{t} \leq 1 + \rho_\alpha, \text{ for } 0 \leq t$$

A.2.3.3. Continuous clocks: Rate

LEMMA 8 *The difference,* Δ_{vc}, *between two consecutive instantaneous virtual clocks,* $\forall_{i,n}$, *is majored by:*

$$\delta_{iv} + 2\rho_p[(1-\rho_p)^{-1}\delta_{iv} + \Gamma^{max}_{start} + \Gamma^{max}_{agree}] + (1+\rho_p)\Delta\Gamma_{tight}$$

Proof: By assumption, virtual clocks are no more than δ_{iv} apart at *ready*^{i}. By Lemma 2 the elected candidate clock will be started no later than $(1-\rho_p)^{-1}\delta_v + \Gamma^{max}_{start}$ after *ready*^{i}. Virtual clocks will drift at most $2\rho_p[(1-\rho_p)^{-1}\delta_v + \Gamma^{max}_{start}]$ during this interval. By assumption 3 elected clocks are started within $(1+\rho_p)\Delta_{tight}$ of each other. By assumption 2 it will take at most Γ^{max}_{agree} to agree on the virtual clock. The maximum difference between virtual clocks will then be the sum of these factors. □

LEMMA 9 *The maximum interval over which the difference between consecutive instantaneous clocks can be spread,* Δ_{spread}, *is given by:* $\Delta_{spread} \leq (T - \Delta^{max}_{ck})(1+\rho)^{-1}$.

Proof: The spreading interval must not be larger than the minimum real time interval between any two consecutive re-synchronizations. In the proof of Lemma 5 we have shown that this interval is given by $(T - \Delta^{max}_{ck})(1+\rho)^{-1}$, thus proving the Lemma. This imposes a bound for the rate of continuous virtual clocks. □

LEMMA 10 *The continuous clocks obtained by definition A.2.3 based on instantaneous clocks resulting from algorithm in figure 5 verify the rate property, that is* $\exists \rho_v$ *such that:*

$$1 - \rho_v \leq 1 - \rho_p - \frac{\Delta_{vc}}{\Delta_{spread}} \leq \frac{vc_m(t_{tk+1}) - vc_m(t_{tk})}{g} \leq 1 + \rho_p + \frac{\Delta_{vc}}{\Delta_{spread}} \leq 1 + \rho_v \text{ for } 0 \leq t_{tk} < t_{tk+1}$$

A.2.4. Global CesiumSpray variables

Let α_g be the accuracy of the set of correct GPS-clocks, as postulated by the NavStar system specifications. Consider fully correct operation of the system, that is, local networks always have at least one correct GPS-node. Let δ_l be the precision of the set of node clocks of a local network, achieved by the a posteriori agreement algorithm, equivalent to δ_v as per the derivation of lemma 6.

A.2.4.1. Accuracy

Lemma 11 *The clock of any correct* CesiumSpray *node verifies the accuracy property, that is,* $\exists \alpha_{CS}$ *such that:*

$$|vc_k(t) - t| \leq \alpha_g + \delta_l \leq \alpha_{CS}, \quad \text{for} \quad 0 \leq t$$

Proof: There is at least one external GPS-node clock in each local network which follows the NavStar GPS time. By lemma 6, every clock is within $\delta v \equiv \delta l$ of that GPS-clock. Correct GPS-node clocks are at most α_g away from real time. □

A.2.4.2. Precision

Lemma 12 *The clock of any correct* CesiumSpray *node verifies the precision property, that is,* $\exists \delta_{CS}$ *such that:*

$$|vc_k(t) - vc_l(t)| \leq 2.\alpha_{CS} \leq \delta_{CS}, \quad \text{for} \quad 0 \leq t$$

Proof: It is a known result that the precision of a set of clocks with accuracy α is at least as good as $2.\alpha$. □

A.2.5. Accuracy preservation

All proofs until now have been based on a correctly operating CesiumSpray system, i.e., one with enough processors (cf. § A.3), and with an external time source per local network. The main consequence is that normal node clocks are closely following GPS-clocks.

However, it is important to consider the consequences of a temporary outage of external time— that is, a local network which is left without any correct GPS-node— even if the reentry conditions after standard accuracy α_{CS} is lost are not formally studied in this paper. The accuracy preservation achieved by the a posteriori agreement algorithm during a period T_{out} is given below.

Lemma 13 *Given the temporary absence of external time at* $t = t_0$*, the maximum interval that the clock of any correct* CesiumSpray *clock is guaranteed to withstand without losing accuracy* α_{out}*, is given by:*

$$T_{out} \leq \frac{\alpha_{out} - \alpha_{CS}}{\rho_v}$$

Proof: We are going to prove for a fast clock, i.e. one where $vc(t_0) = t_0 + \alpha_{CS}$. The bound for slow clocks is symmetrical. From lemma 10, we obtain α_v. So, for $T_{out} = t_1 - t_0$:

$$vc(t_1) - vc(t_o) \leq (1 + \rho_v)\ T_{out}$$

On the other hand, given final accuracy α_{out}, and worst-case starting accuracy α_{CS}:

$$vc(t_1) - vc(t_o) = T_{out} + \alpha_{out} - \alpha_{CS}$$

Finally,

$$T_{out} + \alpha_{out} - \alpha_{CS} \leq (1 + \rho_v)\ T_{out}$$

$$T_{out} \leq \frac{\alpha_{out} - \alpha_{CS}}{\rho_v}$$

□

A.3. Proof of bounds on number of processors

THEOREM 1 *In order to generate and detect a tight broadcast, $(f_p + 1)(f_o + 1)$ processors are required.*

Proof of necessity: Assume that only $f_o + f_o f_p + f_p$ processors are used. Let $\mathcal{P}^f$ be the set of failed processors and $\mathcal{P}^c$ be the set of correct processors. Consider an execution where f_p processors are failed and do not generate any broadcast. In this execution, only $f_o + f_o f_p$ messages are generated. Consider also that these messages can be grouped in disjoint sets $\mathcal{M}_0$, $\mathcal{M}_1$, ..., $\mathcal{M}_{f_p}$ such that:

$$\forall_{b \in \mathcal{M}_0} \exists_{p_o \in \mathcal{P}^c} \ \ s.t\ \ p_o \notin \mathcal{A}^b_{p_c} \forall_{p_c \in \mathcal{P}^c} \tag{A.1}$$

$$\forall_{b \in \mathcal{M}_i} \exists_{p_i \in \mathcal{P}^f} \ \ s.t\ \ p_i \notin \mathcal{A}^b_{p_c} \forall_{p_c \in \mathcal{P}^c} \tag{A.2}$$

$$\forall_{i,j}, 1 \leq i \leq f_p, 1 \leq j \leq f_p\ p_i \neq p_j \tag{A.3}$$

$$\forall_i, o \leq i \leq f_p, \#\mathcal{M}_i = f_o \tag{A.4}$$

In this execution, messages in $\mathcal{M}_0$ cannot be detected as tight broadcasts due to omissions during the acknowledgment phases. Messages in each set $\mathcal{M}_i$ cannot be detected as tight broadcasts due to lack of acknowledgments from one of the failed processors. Each failed processor, p_i affects a different set of messages, $\mathcal{M}_i$. This is execution is clearly possible. Since failed processors affect disjoint subsets of messages of size f_o they cannot be detected as failed, thus, they are not removed from $\mathcal{P}^c_m$ by any correct processor. It is easy to see that, in this execution, we never have $\mathcal{A}^b_m = \mathcal{P}^c_m\ \forall_{m \in \mathcal{P}^c}, \forall_{b \in \mathcal{M}}$. □

Proof of sufficiency: Let $\mathcal{M}_o$ be the set of messages affected by omissions. By definition $\#\mathcal{M}_o \leq f_o$. Let $\mathcal{M}_i$ be the set of messages affected by missing acknowledgments from failed processor p_i. If $\#\mathcal{M}_i > f_o$, processor p_i will be removed from $\mathcal{P}^c$. Thus, in order to contribute to prevent a message from being detected as tight broadcast, $\#\mathcal{M}_i \leq f_o$. There are at most f_p failed processors, thus:

$$\#\{\mathcal{M}_o, \mathcal{M}_1, \ldots, \mathcal{M}_{f_p}\} \leq \#\mathcal{M}_o + \#\mathcal{M}_i + \ldots + \#\mathcal{M}_{f_p} \leq f_o + f_p f_o$$

That is, at most $f_o + f_p f_o$ messages can be disturbed by omissions or failed processors. Thus, its sufficient to generate $f_o + f_p f_o + 1$ messages to detect a tight broadcast. Since failed processors may never generate any broadcast, at most $f_o + f_p f_o + 1 + f_p = (f_o + 1)(f_p + 1)$ processors are required in the system to generate *and* detect a tight broadcast. □

THEOREM 2 *In order to achieve and detect sufficient evidence, $(f_p + 1)(f_o + 1) + f_p$ processors are required.*

Proof: The previous result establishes the number of processors required to generate and detect a single tight broadcast (even if processors crash or are too late). To prevent a resynchronization from being triggered by a faulty process that initiates a broadcast too early, our protocol requires the generation of at least $f_p + 1$ tight broadcasts (this is known as waiting for sufficient evidence [28]). Thus, the new bound is the former, plus f_p: $(f_p + 1)(f_o + 1) + f_p$. □

THEOREM 3 *If processor behavior is strengthened to crash failure, in order to achieve and detect sufficient evidence, only $f_o + f_p + 1 + \max\,(f_p, f_o)$ processors are required.*

Proof: If the assumption about processor behavior is strengthened to crash failure, maintaining the others (arbitrary clocks, controlled network omissions), the number of processors needed is much smaller. The main reason for the improvement is that when processors start doing omissions, they do not recover. This result can be informally justified this way: f_o messages may be lost due to omissions; f_p process may crash and never send messages; another $f_o + 1$ messages are needed to detect them as crashed; if $f_o > f_p$, the messages required for crash detection are enough to achieve sufficient evidence, otherwise f_p messages need to be sent. □

THEOREM 4 *The bounds of theorems 2 and 3 are always greater than or equal to $2f_p + 1$.*

The bounds on number of processors presented by these theorems always verify the bound on the number of processors needed for fault-tolerant execution of the *select* function $(2f_p + 1)$, so it is enough to rely on the former alone. We leave the proof to the reader.

Notes

1. Temps Atomique International, Universal Time Coordinated.
2. Fault-tolerance is achieved by replicating the GPS-nodes.
3. In an assumed Newtonian time frame.
4. It is known that digital clocks have a finite granularity and increase by ticks. This notion is of utmost relevance to derive ordering relations in distributed real-time systems [12].
5. In any case, limited to ρ_p [28].
6. This is the physical propagation time, dependent on the variable distance between nodes.
7. This model fits practically any LAN attachment one may think of, from workstation type on-board VLSI controller to separate controller on multiboard computer. The little added functionality may be achieved by modifying the LAN driver or writing a shell on top of it.
8. A message is a generic name for a piece of encapsulated information that circulates on the network. It may contain a user-level message.
9. See [31] for details.

10. Continuous adjustment will then ensure it also preserves instantaneous rate (cf. appendix).
11. In section 5, we show the potential of such kind of tools.
12. Hence the utility of BNP 3 as a "failure detection" property, without which detection of faulty processors would be impossible in a non-space-redundant network.
13. $\mathcal{D}^i_q$, used in the explanation of figure 4, is no longer necessary, since any— e.g. the first— detected broadcast will do.
14. Suppose there are $2f_p + 1$ running clocks, and $2f_p + 2$ more enter.
15. A note about engineering: the longest non-preemptable CPU execution is a crucial figure to compose this bound.
16. SPART is a registered trademark of Bull S.A.
17. The case for a slow clock is symmetrical.

References

1. Kenneth Birman, Andre Schiper, and Pat Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems*, 9(3), August 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. F. Cristian, Aghili. H., and R. Strong. Clock Synchronization in the Presence of Omission and Performance Faults. In *Digest of Papers, The 16th International Symposium on Fault-Tolerant Computing*, pages 218–223, Viena - Austria, July 1986. IEEE.
4. Flaviu Cristian. Probabilistic Clock Synchronization. *Distributed Computing, Springer Verlag*, 1989(3), 1989.
5. Flaviu Cristian. Synchronous atomic broadcast for redundant broadcast channels. *The Journal of Real-Time Systems*, 2(1):195–212, 1990.
6. FDDI. *FDDI Token-Ring Media Access Control (MAC)*. ANSI X3.139, 1987.
7. Joseph Y. Halpern, Barbara Simons, Ray Strong, and Danny Dolev. Fault-Tolerant Clock Synchronization. In *Proceedings of the 3Rd ACM Symp. on Principles of Distributed Computing*, pages 89–102, Vancouver Canada, August 1984.
8. 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.
9. *ISO DIS 8802/4-85, Token Passing Bus Access Method*, 1985.
10. *ISO DP 8802/5-85, Token Ring Access Method*, 1985.
11. Hermann Kopetz and Wilhelm Ochsenreiter. Clock Syncronization in Distributed Real-Time Systems. *IEEE Transactions on Computers*, C-36(8):933–940, August 1987.
12. Hermann Kopetz and Wolfgang Schwabl. Global time in distributed real-time systems. Technical Report 15/89, Technische Universitat Wien, Wien Austria, October 1989.
13. 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.
14. L. Lamport and P. Melliar-Smith. Synchronizing Clocks in the Presence of Faults. *Journal of the ACM*, 32(1):52–78, January 1985.
15. Gerard Le Lann and Nicholas Riviere. Real-time communications over broadcast networks: the CSMA-DCR and the DOD-CSMA-CD protocols. Technical Report 1863, INRIA, March 1993.
16. Jennifer Lundelius and Nancy Lynch. A New Fault-Tolerant Algorithm for Clock Syncronization. In *Proceedings of the 3rd ACM SIGACT-SIGOPS Symp. on Principles of Distrib. Computing*, pages 75–88, Vancouver-Canada, August 1984.
17. David Mills. Network time protocol (version 2): Specification and implementation. Technical Report RFC 1119, DARPA Network Working Group, September 1989.
18. Alan Olson and Kang G. Shin. Probabilistic clock synchronization in large distributed systems. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 290–297, Arlington, Texas, USA, May 1991. IEEE.

19. B. Parkinson and S. Gilbert. Navstar: Global positioning system— ten years later. *Proceedings of the IEEE*, 71(10):1177–1186, October 1983.
20. D. Powell, editor. *Delta-4 - A Generic Architecture for Dependable Distributed Computing*. ESPRIT Research Reports. Springer Verlag, November 1991.
21. Parameswaran Ramanathan, Kang G. Shin, and Ricky W. Butler. Fault-Tolerant Clock Synchronization in Distributed Systems. *IEEE, Computer*, pages 33–42, October 1990.
22. 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. IEEE.
23. L. Rodrigues, P. Veríssimo, and A. Casimiro. Using atomic broadcast to implement *a posteriori* agreement for clock synchronization. In *Proceedings of the 12th Symposium on Reliable Distributed Systems*, pages 115–124, Princeton, New Jersey, October 1993. IEEE. Also as INESC AR/29-93.
24. R. Rom. Ordering subscribers on cable networks. *ACM Transactions on Computer Systems*, 2(4), November 1984.
25. U. Schmid. Synchronized universal time coordinated for distributed real-time systems. *Control Engineering Practice*, to appear 1995.
26. Fred B. Schneider. Understanding Protocols for Byzantine Clock Synchronization. Technical report, Cornell University, Ithaca, New York, August 1987.
27. Kang G. Shin and P. Ramanathan. Synchronization of a Large Clock Network in the Presence of Malicious Faults. *IEEE*, pages 13–24, 1985.
28. T. K. Srikanth and Sam Toueg. Optimal Clock Synchronization. *Journal of the Association for Computing Machinery*, 34(3):627–645, July 1987.
29. P. Veríssimo and José A. Marques. Reliable broadcast for fault-tolerance on local computer networks. In *Proceedings of the Ninth Symposium on Reliable Distributed Systems*, Huntsville, Alabama-USA, October 1990. IEEE. Also as INESC AR/24-90.
30. P. Veríssimo and L. Rodrigues. A posteriori Agreement for Fault-tolerant Clock Synchronization on Broadcast Networks. In *Digest of Papers, The 22nd International Symposium on Fault-Tolerant Computing*, Boston - USA, July 1992. IEEE. INESC AR/65-92.
31. Paulo Veríssimo. Real-time Communication. In S.J. Mullender, editor, *Distributed Systems, 2nd Edition*, ACM-Press, pages 447–490. Addison-Wesley, 1993.
32. Q. Zheng and K. G. Shin. Fault-tolerant real-time communication in distributed computing systems. In *Digest of Papers, The 22nd International Symposium on Fault-Tolerant Computing Systems*, pages 86–93. IEEE, 1992.