Uniform timing of a multi-cast service

Augusto Ciuffoletti
Università di Pisa
Dipartimento di Informatica
Corso Italia 40 – 56100 PISA (Italy)
e.mail: augusto@di.unipi.it
home-page: http://www.di.unipi.it/∼augusto/

Abstract

We present a new architecture for a clock synchronization protocol based on multi-cast communication. The protocol implements the gradual tuning of the clocks, in order to automatically compensate the systematic *drift and increase the time between re-synchronizations. Hosts participating to the same service are grouped into a cohort: the protocol self-stabilizes into an optimal routine where all members of a cohort are periodically synchronized with the same messages. New members dynamically joining the cohort perturb this routine, that is spontaneously and eventually restored.*

Keywords*: distributed algorithms, distributed timing, virtual clock, distributed multimedia applications, broadcast communication, self-stabilization.*

1 Introduction

The availability of a common time reference is one of the emerging needs of complex distributed services. Especially when the service involves soft real time tasks, the availability of a common clock improves both performance and robustness.

The literature offers several examples of implementations of timing protocols that are structured according to a common scheme (for instance, [8, 1, 6], and [12, 11] for the part of the protocol related to external clock synchronization) that here we call **system clock based**. In this view the clock synchronization is a separate service, provided by a specialized server, and conforms to some internationally agreed standard: for instance, the client might down-load the Universal Time Coordinate (a standard measure of time) from a server using the Network Time Protocol (NTP [8], a standard Internet protocol), and update the local *system clock*. The system clock is then used by the distributed applications that need a timely coordination: each client assumes that the other clients, as well as the server, have the *system clock* properly synchronized to the same standard. Whenever the accuracy of the *system clock* is considered insufficient, the client asks the NTP server for another accurate clock value, and adjusts the *system clock* using the reply.

The limits of this approach are in its monolithic structure: the synchronization of the system clock has a non negligible impact on the whole host, and the distributed service makes a critical assumption, that cannot be verified, about all participants. There are cases when these drawbacks have to be accepted. For instance, when the processing of the host is mainly time-dependent, as in the case of a database manager: time-stamps associated to the items stored in the database should conform to a universal standard.

In many other cases the use of the common time reference is limited to a single service [7]. For instance in a distributed multimedia application, where frames coming from different sources, and with different delays, should be presented preserving the synchronism, or at least verifying that the required quality of service in maintained.

In those cases, the synchronization of the *system clock* is regarded as unnecessary, provided that a restricted set of time-related operations are available; namely

- the production of **time-stamps** and
- the setup of **timeouts**

The implementation of these functionalities does not need the synchronization of the *system clock*: when the granularity of the time reference is compatible with the speed of the processor, a software (following [9]) *virtual clock*, can be effectively used instead. The advantages of a **virtual clock based** approach, as opposed to the *system clock based* one, can be summarized as follows:

- the scope of the timing received from the server is limited to the service itself. The synchronization of the *system clock* is avoided, improving the robustness of the service and of the overall system: in fact, the synchronization of the *system clock* is a sensible operation.
- each service may be based on its own time reference, admitting that the timing of distinct services may differ. However, the synchronization of distinct services remains possible using the same *virtual clock*.
- the control of the clock synchronization operation can be shifted from the client to the server. Therefore the server can schedule the clock re-synchronization rounds so that multiple clients are resynchronized in the same round, thus optimizing the cost of the clock synchronization algorithm.
- for the same reason the server can anticipate the cost of the service in terms of workload, and therefore guarantee a given quality of service to the clients. Mutual identification for security purposes may be based on a public key approach.
- the clock synchronization service would not affect a unique resource (namely, the *system clock*): therefore the client may maintain multiple, redundant timing devices, the *virtual clock*, and apply a selection rule.

We introduce a *virtual clock* based solution to the problem of providing a client-server distributed application with a common time reference: a *synchronization protocol* is in charge of maintaining a consistent *virtual clock* at each client site participating to the application. The *synchronization protocol* runs under the control of the server, which periodically provides each client with timing information, in order to keep the *virtual clock*s sufficiently close. It is up to the server to optimize the *synchronization protocol* operation, for instance using the same multi-cast communication to reach several clients. When the client *subscribes* for the service the *synchronization protocol* is associated to, it automatically enters the *synchronization protocol*. When the service admits the presence of several servers for fault tolerance reasons, multiple *virtual clock*s, each controlled by a distinct server, may be used at the client side.

We first discuss the data that must be exchanged between the server and a client, in order to efficiently implement the *virtual clock* on the client side. The next step is the description of a multi-cast based protocol, controlled by the server, that implements the data exchange. A similar approach can be found in [5]: our solution differs since we do not make any assumption on the communication media.

Finally we introduce an algorithm that converges to a routine where all clients are synchronized using a unique multi-cast message, and preserves this property despite perturbations.

In this sense we say that this part of our solution is self stabilizing [2]: it tends to the optimal situation (all clients synchronized with one multi-cast) whatever is the initial state. As a consequence, perturbations are eventually compensated without undertaking exceptional actions.

There is little or no relation with other clock synchronization algorithms that feature self-stabilization, as [4]: they are meant to provide an internal synchronization, while we suggest an external clock synchronization, driven by a time server. In this sense our solution is *less distributed*.

2 Implementation of a *virtual clock*

The overall synchronization algorithm consists in the generation of an (infinite) sequence of *virtual clock incarnation*s on each client. Each *incarnation* is responsible for the timing of the service S during a certain time interval. The basic requirement of a *virtual clock* is that the timing it provides is sufficiently accurate, with respect to the timing of the server. In the following, we will use the adjective “real” to refer to the properties of the timing of the server, although this timing may or not reflect any internationally agreed standard for real time.

The implementation of the *virtual clock* is based upon the existence of an internal timing (for instance, that derived from the *system clock*), whose relation with the real timing is known with a coarse approximation. The following definition embeds this basic knowledge about the relation between the value of the *internal clock* with respect to the *real clock*:

Definition 1 (Internal clock) *Let $t(T)$ be the value of the* internal clock *at real time T:*

$$\frac{dt}{dT} = 1 + \rho(T)$$

It simply says that the two clocks diverge at a variable speed: the $\rho()$ is usually called the *drift*. The *internal clock* is equivalent to the *an-isochronous clock* as defined in [10].

If we assume that the *drift* is known with some approximation,

$$\forall T, \mathit{drift} - \mathit{drift}_{err} \leq \rho(T) \leq \mathit{drift} + \mathit{drift}_{err}$$

using the definition of *internal clock*, and some analytical results [1], we obtain the inequality:

$$\begin{aligned} T &\geq \frac{t(T) - t(T_0)}{(1 + \mathit{drift} + \mathit{drift}_{err})} + T_0 \\ T &\leq \frac{t(T) - t(T_0)}{(1 + \mathit{drift} - \mathit{drift}_{err})} + T_0 \end{aligned} \tag{1}$$

We want to reduce the above inequality so that the bounds for T can be computed by the client: our first step is to assign a (initial) value to *drift* and drift_{err}.

The hardware constructor usually provides a frequency tolerance ρ_0 and a frequency stability ρ_R of the timing device. The first characteristic depends on the quality of the construction process, while the second depends on the technology of the timing device[2]. The former corresponds to a systematic drift, while the latter describes a term of the drift that depends on time:

Definition 2 (Bound drift)

$$\begin{aligned} \forall T, \rho(T) &= \rho_s + \rho_r(T) \\ |\rho_s| &\leq \rho_0 \\ \forall T, |\rho_r(T)| &\leq \rho_R \end{aligned}$$

For a quartz clock device the ρ_0 can be of the order of 10^{-3}, while ρ_R, for standard temperature excursions, is of the order of 10^{-6}.

From these data we derive an initial setup

$$\begin{aligned} \mathit{drift} &= 0 \\ \mathit{drift}_{err} &= \rho_0 + \rho_R \end{aligned}$$

However, this setup is extremely coarse, and one of the tasks of the *synchronization protocol* is that of refining it. Here we want to implement this at software level: namely, every new *incarnation* of the *virtual clock* is built taking into account the drift experienced by past *incarnation*s. As the series of the *incarnation*s increases, the estimate of the two components of the drift becomes more and more precise. The NTP protocol ([8]) implements a similar

[1] namely, the Lagrange theorem

[2] This parameter reflects the dependency of the timing device from environmental features: temperature, pressure, etc. This term does not take into account the “aging” of the device, that is considered irrelevant during the operation of the distributed activity

t_s^i	the value of *internal clock* when reading the remote clock
rcr^i	the remote clock reading result
rcr_{err}^i	the error in the clock reading operation
$drift^i$	the estimated drift
$drift_{err}^i$	the upper bound of the error of the drift estimate
ρ_R^i	the upper bound of the residual drift

Table 1. The internal state of the i-th incarnation of a *virtual clock*

algorithm when it calibrates the PLL device provided by some hardware platforms. We are going to consider this topic before the end of this section.

The value $t(T_0)$ cannot be computed locally, since it implies the availability of the real time to toggle a reading of the *internal clock*; however, the host can obtain an approximation of this value by interacting with the server. We will detail this operation later: here we define its effect, which is that of returning a 3-ple $[t_s, rcr, rcr_{err}]$, with the following meaning:

Definition 3 (Remote clock reading) *A remote clock reading operation returns a triple $[t_s, rcr, rcr_{err}]$, such that:*

$$\begin{aligned} t_s &= t(T_s) \\ T_s &\in [rcr - rcr_{err}, rcr + rcr_{err}] \end{aligned}$$

Now we can substitute $t(T_0)$ with t_s, and T_0 with the two bounds for T_s in the previous definition, in the inequality (1), and obtain:

$$\begin{aligned} T &\geq \frac{t(T) - t_s}{(1 + drift + drift_{err})} + rcr - rcr_{err} \\ T &\leq \frac{t(T) - t_s}{(1 + drift - drift_{err})} + rcr + rcr_{err} \end{aligned} \tag{2}$$

The values that are used to estimate the interval containing the real time T are presently known to the client, and the two bounds above can be encapsulated in a new *incarnation* of the *virtual clock*. When a remote clock reading operation completes successfully, a new *incarnation* of the *virtual clock* is created. Its internal state (summarized in Table 1) is set with the results of the remote clock reading operation, and will never be altered.

As pointed out in page 2, one of the fundamental functions of a *virtual clock* is the production of time-stamps. The inequality (2) states that the *virtual clock* can produce, using its internal state and the *internal clock*, a time-stamp representative of the real time, and bound its accuracy: we denote with $vc(T)$ the time-stamp produced at (real) time T, and the upper bound for its accuracy with $vc_{err}(T)$. The following lemma (the proof of its validity is in appendix A.1) gives the explicit expression to compute them:

Lemma 1 (Virtual Clock Time-stamps) *At time T, a* virtual clock *can produce a time-stamp $vc(T)$ and a bound $vc_{err}(T)$ of its accuracy by applying the following rules:*

$$vc(T) = rcr + (t(T) - t_s)(1 - drift) \tag{3}$$

$$vc_{err}(T) = rcr_{err} + (t(T) - t_s)drift_{err} \tag{4}$$

The other basic feature indicated in page 2 is the ability to setup timeouts. The *virtual clock* will exploit the knowledge of the correspondence between the *internal clock* and the real time in order to compute an internal time corresponding to the deadline. To implement the interrupt generation the client application uses the facilities offered by the native operating system, driven by the *internal clock*. The following lemma (the proof of its validity is in appendix A.1) gives the expressions for an estimate of the *internal clock* when the deadline expires, and its approximation:

Lemma 2 (Virtual Clock Timeout) *A* virtual clock *can produce an estimated value $ic(T)$ of the* internal clock *at real time T, and a bound $ic_{err}(T)$ of its precision by applying the following rules:*

$$\begin{aligned} ic(T) &= t_s + (1 + drift)(T - rcr) \\ ic_{err}(T) &= rcr_{err} + drift_{err}(T - rcr) \end{aligned}$$

The *incarnation* expires as soon as the accuracy of the real time it can compute exceeds a given threshold δ, which corresponds to the timing accuracy required by the distributed application: in order to fulfill its task, the *virtual clock* must never deliver a time-stamp whose vc_{err} exceeds this threshold. From the equation (4) we can derive the time-to-live τ of a given *incarnation* (the proof of its validity is in appendix A.3):

Lemma 3 (Time to live)

$$\tau = \frac{\delta - rcr_{err}}{drift_{err}} \tag{5}$$

The τ reflects the efficiency, and ultimately the reliability, of the *synchronization protocol*: a large τ means a lower network overhead due to remote clock reading, and a lower probability to hit a network problem. Excluding δ, which depends on the application, and rcr_{err}, which depends on the network characteristics, the most promising optimization parameter is $drift_{err}$.

Using the result of the current and of a past *remote clock reading* (see definition 3, the value of the drift can be estimated: under certain hypotheses, the precision of this estimate can be gradually refined.

The following rule can be used by the client in order to compute an approximation of the drift, and the precision of such approximation (the proof of its validity is in appendix A.2):

Definition 4 (Virtual clock drift estimate) *Let ε be an the upper bound of the accuracy rcr_{err} when a remote clock reading operation is considered successful. When the n-th remote clock reading returns successfully, if the $(n - z)$-th* virtual clock *is available*

$$\begin{aligned} drift^n &\approx \frac{t_s^n - t_s^{n-z}}{rcr^n - rcr^{n-z}} - 1 \\ drift_{err}^n &\approx \frac{2\varepsilon}{rcr^n - rcr^{n-z}} + \rho_R \end{aligned}$$

are valid values for the state of the n-th virtual clock.

When distant *incarnation*s are used to interpolate the systematic drift, we obtain that the drift estimate error tends to decrease. As a consequence, the τ tends to increase.

The following theorem proves that this process is limited, and the τ tends to a limit value (the proof of its validity is in appendix A.3):

Theorem 1 (Limit of the time to live) *Let ε be an the upper bound of the accuracy rcr_{err} when a remote clock reading operation is considered successful. The following holds, after n successful clock reading:*

$$\begin{aligned} \lim_{n\to\infty} \tau = \overline{\tau} &= \frac{\delta - \varepsilon - \frac{2\varepsilon}{z}}{\rho_R} \qquad (6) \\ &< \frac{\delta - \varepsilon}{\rho_R} \qquad (7) \end{aligned}$$

This proves that the time to live of *incarnation*s cannot increase indefinitely, and that the asymptotic value increases as farther values are used to interpolate the drift. However, there is an upper limit, given by inequality (7), that corresponds to a theoretical limit bound to the basic parameters of the clock synchronization scheme.

Since the limit value of τ cannot be negative, we can derive a further bound for z, the backlog of *rcr* that is needed to interpolate a new *drift*:

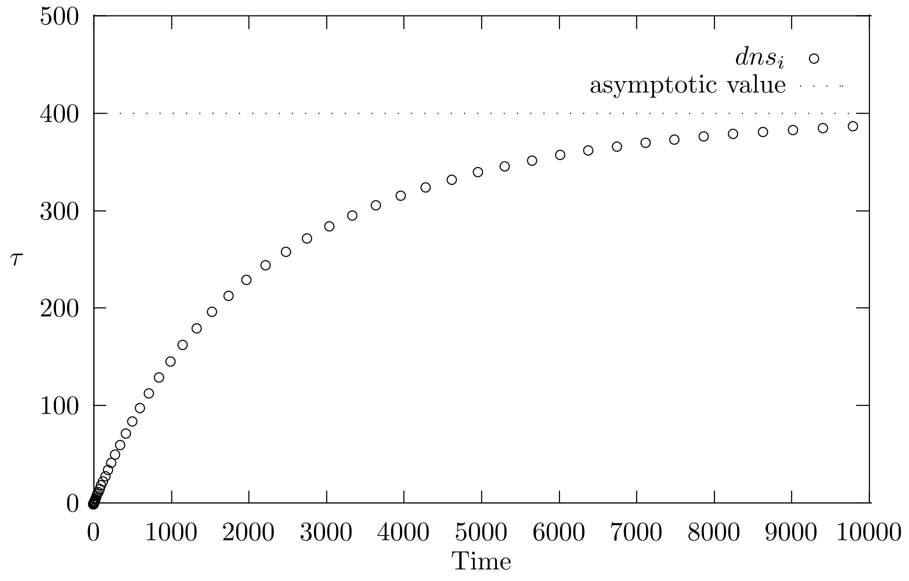


Figure 1. Evolution of the τ in time: each point is a setup operation

$$\delta > \varepsilon \left(\frac{2}{z} + 1 \right) \tag{8}$$

It is straightforward to note that a backlog of just one *rcr* (i.e. $z = 1$) is a quite restrictive starting point: we cannot ensure an accuracy better than three times the accuracy reached at the end of a successful remote clock reading operation. For instance, if the remote clock reading has a precision of 1 msecs, the *virtual clock* accuracy cannot be better than 3 msecs, if we want that the drift estimate converges. As a general rule the value of z is chosen well above that threshold: from equation (6) we see that the asymptotic value of the series gets closer to the upper bound when the value of z is large. Summarizing, extending the length of the record the history has a very limited cost, and significantly improves the performance of the protocol.

If we observe the τ of successive *incarnation*s (see Figure 1), we see that it rapidly approaches the asymptotic bound, and then slowly reaches the bound in an infinite time. The Figure 1 represents the evolution with the values indicated in Table 2.

ρ_0	10^{-3}	clock frequency tolerance
ρ_R	10^{-6}	clock frequency stability
δ	1.5 msecs	*virtual clock* accuracy limit
ε	1 msecs	threshold for clock reading
z	20	distance

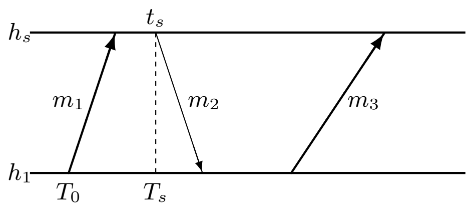
Table 2. Characteristics of a service, and sample values

The asymptotic value of the τ, indicated by the horizontal line, is 400 seconds. The τ reaches half that value (200 secs) after approximately 1500 secs, during the 40th incarnation of the *virtual clock*. The initial τ was 0.12 seconds. The distance z has been chosen well above 4, the value returned by inequality (8) using the values indicated in Table 2.

3 The remote clock reading operation

We detail the operation of the remote clock reading[3], whose functional description is given in definition 3.

[3]The remote clock reading is deliberately simplified, since the description of a smarter implementation would need the introduction of a number of details which are inappropriate for a paper addressing the architecture of the tool.



```
struct m1 {timeval T0;}
struct m2 {timeval T0; hostname h1;}
struct m3 {timeval T0;}
```

Figure 2. A remote clock reading operation and the content of the messages. Thick lines indicate broadcast messages.

Our discussion is limited to a set of clients that share the same timing, represented by the values assigned to the variables listed in table 2. Such set of clients is called a *cohort*. The extension to the synchronization of several cohorts is dealt with in the next section.

The remote clock reading operation involves one server, which provides the service and controls the *synchronization protocol*, and a set of clients, which act as subscribers. The overall operation is divided into two steps, which are coordinated by the server. In Figure 2 there is the message exchange pattern from the server to one of the clients, and the content of the messages.

During the first step, the server broadcasts a request of attention `m1` among the clients, and waits for a timely reply `m2` from them.

Message `m1` contains the value of the real time when the message was sent, T_0 in Figure 2. The reply `m2` contains the same value T_0, which is used as an identifier for the clock reading operation, and the network identifier of the client.

Upon sending `m2`, the client creates a new *incarnation* of the *virtual clock*, whose content is initialized as follows:

$$\begin{aligned} rcr &= T_0 + \varepsilon \\ rcr_{err} &= \varepsilon \end{aligned}$$

The value of t_s is set to the value of the local *internal clock*, and the rest of the *virtual clock* state is computed as indicated in Section 2. The newly created *incarnation* is not yet delivered for use: the purpose of the second phase of the protocol is that of deciding if it is safe to deliver it.

After generating the message `m1`, the server waits for the replies from the clients for a limited time. The deadline is set as follows:

$$deadline = T_0 + 2\varepsilon$$

The server stops processing the replies when all clients have replied, or when the deadline expires.

The second phase of the protocol consists in the transmission of a broadcast message from the server to the clients whose reply was received within the deadline. The receipt of this message authorizes the client to deliver for internal use the *virtual clock incarnation* created during the first phase of the protocol. Otherwise the client concludes that the remote clock reading operation was not successful.

A viable continuation after a failure, that we will not explore in this paper, is that of repeating the remote clock reading operation to give further opportunities to the failing clients. About this subject see [1] and [3].

It is easy to prove that the *virtual clock* of the clients that receive `m3` is correct: in fact, if they sent `m2` before $T_s + 2\varepsilon$ (and this is true since `m2` has been received before that time) then (see figure 2):

$$T_s \in [T_0, T_0 + 2\varepsilon]$$

or, equivalently

$$T_s \in [rcr - rcr_{err}, rcr + rcr_{err}]$$

which matches with the statement of Definition 3.

The second phase of the protocol terminates when `m3` has been sent. The clients that receive `m3` deliver the new *incarnation* and out-date the previous.

From the discussion in Sect. 2, we know that the validity of the *virtual clock* is limited in time: the value of τ obtained using equation (5) indicates the time to live of a given cohort. From that value we may derive the following deadline for the next remote clock reading operation:

$$T_s + \frac{\delta - \varepsilon}{drift_{err}}$$

The deadline depends on some characteristic parameters of the cohort, and on the timing of the past clock reading operations. These pieces of information are known to the server, that can schedule by that time the next `m1` broadcast message, that will trigger the next clock reading operation.

This concludes the description of the remote clock reading operation: it is easy to check that the timing of the operation is entirely under the control of the server, which enforces the required accuracy of the *virtual clock*s, and (implicitly) warns the clients that may have failed to read the remote clock with the required accuracy.

4 Keeping cohorts compact

The performance of the scheme illustrated above is optimal when all the subscribers of the same service belong to a single cohort. In that case we say the the *synchronization protocol* is *coherent*: the server schedules a simultaneous clock reading for all members of the cohort, and optimizes the cost of the communication.

This result is easy to obtain if the membership of the clients of the service is fixed during the duration of the service itself. But this assumption is not realistic, and it is safer to assume that subscriptions occur irregularly: new instances of the same service are activated dynamically, and new clients can join an already active service, while others can leave it for exceptional or ordinary events.

We need to define a merge operation among sub-cohorts, that models the fact that several cohorts may merge into a single one, whose timing is controlled by a coherent *synchronization protocol*:

Definition 5 (Merging sub-cohorts) *Given a set of sub-cohorts, their merge consists in their union in a new sub-cohort, whose next remote clock readings are set as the earliest, among those of each component sub-cohort.*

Indiscriminate merges are not efficient. For instance, when a client first subscribes for a service, its *virtual clock* is initialized with the coarse τ_0: if this (singleton) sub-cohort were merged with an existent sub-cohort, all members would experience a series of very short τs, needed by the new member in order to calibrate the *drift* of its internal clock. This is inefficient, and it is preferable that the joining member merges with a preexistent sub-cohort only after its τ is comparable with that of the sub-cohort.

We want that the system tends to coherency, but limiting the cost of the merges: so we suggest a protocol for merging sub-cohorts, and we prove that its *cost*, in terms of the degradation of the overall τ, has an upper bound:

Definition 6 (Sub-cohorts merging rule) *At time t_a a sub-cohort c_a is performing a remote clock reading operation. Let sub-cohort c_b be a sub-cohort whose current* virtual clock incarnation *have been generated at t_b (with $t_b \leq t_a$), with a time-to-live of τ_b.*

Given appropriate σ and ξ, the two cohorts merge if and only if:

$$\begin{aligned} \tau_a, \tau_b &\in [(1-\sigma)\overline{\tau}, \overline{\tau}] && (9) \\ t_b + \tau_b &\geq t_a + \xi * \tau_a && (10) \end{aligned}$$

Intuitively, the merge is allowed only if:

- the τ of the two cohorts are sufficiently close to the asymptotic value (see the σ parameter) and
- the next application of the clock reading rule of the merged sub-cohort is sufficiently distant (see the ξ parameter).

The following theorem (the proof of its validity is in appendix A.4) gives a criteria for choosing "appropriate" values for ξ and σ:

Theorem 2 (Bounded degradation) *If*

$$0 < \xi \leq \frac{1-\sigma}{2-\sigma} \tag{11}$$

then the merge of two sub-cohorts does not degrade their τ to less than $(\xi - \sigma)\overline{\tau}$

Example:
We want that a merge never produces a sub-cohort with

$$\tau < \frac{\overline{\tau}}{3}$$

From the consequence of the theorem,

$$\xi = \sigma + \frac{1}{3}$$

By substitution in the premise of the theorem we obtain:

$$\sigma + \frac{1}{3} \leq \frac{1-\sigma}{2-\sigma}$$

and, by algebraic manipulation

$$3\sigma^2 - 8\sigma + 1 \geq 0$$

The relevant solution of the equation is

$$\sigma \leq 0.13$$

and, from the consequence of the theorem:

$$\xi \leq 0.46$$

The best choice is $\sigma = 13\%$ and $\xi = 46\%$.

Note that the inequality formalizes a tradeoff between the time needed to merge, and the degradation of the τ of the sub-cohort that "accepts" a merging sub-cohort.

The σ indicates how near to the asymptotic value must be the τ of a sub-cohort, in order to be considered for merging with another: this impacts on the time from the initial join to the merge. In the example illustrated in Figure 1 a σ of 13% corresponds to a time from the initial join amounting to $5500 secs$.

On the other hand, using the value of ξ derived in the example, the fraction of τ that is preserved after the merge is (in the worst case) 33% of the asymptotic value. From the *bounded degradation* theorem, it is easy to see that the lower bound for such degradation cannot be better than 50%, whatever value is chosen for σ.

The appropriate value for the σ strictly depends on the kind of parameter that one wants to optimize: the number of rounds versus the number of sub-cohorts.

We cannot guarantee that a service ever reaches coherency: frequent activations or joins may maintain the time diffusion in-coherent. But the following result proves that the above protocol "tends" to coherency, and will reach it in case the service does not experience new subscriptions (the proof of its validity is in appendix A.5):

Theorem 3 (Self-stabilization) *In the same condition stated for the bound degradation, and if no new members subscribe for the service, it holds that all sub-cohorts will eventually join.*

From Theorem 3 follows that new subscribers eventually join the cohort, if the inequality that binds the parameters σ and ξ is satisfied.

5 Conclusions

We have described the architecture of a (virtual) clock synchronization protocol, that is aimed at providing a uniform timing to the clients of a distributed service using a multi-cast mechanism. This sort of tool is useful in the design of multimedia distributed applications, as well as in other time-bound applications.

The distinctive feature of the protocol we propose is the ability to calibrate the virtual clocks of the clients, so that the frequency of the remote clock reading operations gradually decreases. In order to make this result of practical interest, a merging rule has been defined that tends to a state where all clients obtain a new remote clock reading through a unique broadcast operation, despite new subscription or the activation of new instances of the service.

References

[1] Flaviu Cristian. Probabilistic clock synchronization. *Distributed computing*, (3):146–158, 1989.

[2] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.

[3] Federica Gattai, Roberta Golinelli, and Augusto Ciuffoletti. Clock synchronization in a virtual ring. In *Proc. 6th Euromicro Workshop on Real-Time Systems*, pages 72–77, Vaesteraas (Sweden), June 1994.

[4] Mohamed G. Gouda and Ted Herman. Stabilizing unison. *Information Processing Letters*, (35):171–175, 1990.

[5] J. Y. Halpern and I. Suzuki. Clock synchronization and the power of broadcasting. *Distributed Computing*, 5(2):73–82, 1991.

[6] Herman Kopetz, A. Kruger, D. Millinger, and A. Schedl. A synchronization strategy for a time triggered multicluster real-time system. In *Proceedings of the 14th Symposium on Reliable Ditributed Systems*, pages 154–161, 1995.

[7] Leslie Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Transactions of Programming Languages and Systems*, 6(2):254–280, April 1984.

[8] Dave L. Mills. Internet time synchronization: the network time protocol. *IEEE Transactions on Communications*, COM-39(10):1482–1493, October 1991.

[9] F. Schmuck and F. Cristian. Continuous clock amortization need not affect the precision of a clock synchronization algorithm. In *9th Annual ACM Symposium on Principles of Distributed Computing*, pages 133–143, 1990.

[10] Fabio A. Schreiber. Is time a real time? An overview of time ontology in informatics. In *Real Time Computing.* 1992.

[11] T.K. Srikanth and Sam Toueg. Optimal clock synchronization. *Journal of the Association of Computer Machinery*, 34(3):627–645, July 1987.

[12] Paulo Verissimo, Louis Rodrigues, and Antonio Casimiro. CESIUMSPRAY: A precise adn accurate global time service for large-scale systems. *Real-Time Systems*, (12):243–294, 1997.

A Appendix: proofs of the theorems

A.1 Proofs of the time-stamp and timeout generation rule

- As for the time-stamp generation rule of Definition 1, the time-stamp is given by the midpoint of the interval indicated by inequality (2), and its accuracy is given by half the length of the same interval:

$$\begin{aligned}
vc(T) &= rcr + (t(T) - t_s)\frac{1 + \mathit{drift}}{(1 + \mathit{drift} + \mathit{drift}_{err})(1 + \mathit{drift} - \mathit{drift}_{err})} \\
vc_{err}(T) &= rcr_{err} + (t(T) - t_s)\frac{\mathit{drift}_{err}}{(1 + \mathit{drift} + \mathit{drift}_{err})(1 + \mathit{drift} - \mathit{drift}_{err})}
\end{aligned}$$

Since both values of *drift* are small with respect to 1, we introduce the following approximation, that simplifies the formulas:

$$\begin{aligned}
(1 + \mathit{drift} + \mathit{drift}_{err})(1 + \mathit{drift} - \mathit{drift}_{err}) &\approx 1 + 2\mathit{drift} \\
(1 + 2\mathit{drift})^{-1} &\approx 1 - 2\mathit{drift} \\
\mathit{drift} + \mathit{drift}^2 &\approx \mathit{drift}
\end{aligned}$$

- Concerning the timeout generation, we use the knowledge of the correspondence between the *internal clock* and the real time, embed in the *virtual clock*, to devise a value of the *internal clock* that corresponds to the timeout.

  The inequality (2) can be rewritten as follows:

  $$t_s + (1 + \mathit{drift} - \mathit{drift}_{err})(T - rcr - rcr_{err}) \le t(T) \le t_s + (1 + \mathit{drift} + \mathit{drift}_{err})(T - rcr + rcr_{err})$$

  which returns an interval for the value of the *internal clock* at real time T: the *virtual clock* can return this information as an indication of the timeout to set on the *internal clock* in order to produce the event controlled by the timeout at real time T. Also in this case, we compute the midpoint and the extent of the interval to indicate the estimated value and a bound for its precision. The terms of the kind $\mathit{drift} * rcr_{err}$ are ignored.

A.2 Proof of the rules to compute *drift* and drift_{err}

From Definition 1, the *average* observed drift can be computed as

$$1 + \mathrm{ave}(\rho) = \frac{t_s^n - t_s^{n-z}}{T_s^n - T_s^{n-z}}$$

Using Definition 3, and assuming without loss of generality that both remote clock reading operations have accuracies better than ε, we can compute the two bounds of the average drift:

$$\frac{t_s^n - t_s^{n-z}}{rcr^n - rcr^{n-z} + 2\varepsilon} \le 1 + \mathrm{ave}(\rho) \le \frac{t_s^n - t_s^{n-z}}{rcr^n - rcr^{n-z} - 2\varepsilon}$$

and, the Lagrange theorem again justifies

$$\frac{t_s^n - t_s^{n-z}}{rcr^n - rcr^{n-z} + 2\varepsilon} - \rho_R \le 1 + \mathit{drift}^n \le \frac{t_s^n - t_s^{n-z}}{rcr^n - rcr^{n-z} - 2\varepsilon} + \rho_R \qquad (12)$$

The above inequality binds the value of the systematic drift of the *internal clock* when two remote clock readings are available: this is the situation that arises when the timeout of the $(n-1)$-th *incarnation* of the *virtual clock* approaches, and the client performs a new remote clock reading operation to initialize a new *incarnation.*

The values for the *drift* and the drift_{err} variables in the state of the n-th *incarnation* can be computed as the midpoint and the extent of the interval of inequality (12). Some second order terms are eliminated.

$$\begin{aligned}
1 + drift^n &= \frac{(t_s^n - t_s^{n-z})(rcr^n - rcr^{n-z})}{(rcr^n - rcr^{n-z})^2 - 4\varepsilon^2} \\
&\approx \frac{t_s^n - t_s^{n-z}}{rcr^n - rcr^{n-z}} \\
drift_{err}^n &= \frac{(t_s^n - t_s^{n-z})2\varepsilon}{(rcr^n - rcr^{n-z})^2 - 4\varepsilon^2} + \rho_R \\
&\approx \frac{2(1+\rho_s)\varepsilon}{rcr^n - rcr^{n-z}} + \rho_R \\
&\approx \frac{2\varepsilon}{rcr^n - rcr^{n-z}} + \rho_R
\end{aligned}$$

A.3 Proof of Theorem 1

Let us denote with τ_n the τ of the n-th *incarnation* of a *virtual clock*. When remote clock reading values from distant z *incarnation*s are used, we have that:

$$\tau_n = \frac{\delta - \varepsilon}{\frac{2\varepsilon}{lcs_n - lcs_{n-z}} - \rho_R}$$

Since the time to live τ indicates the maximum time before a new *incarnation* of the *virtual clock* is created, then $\tau_i \geq lcs_{i+1} - lcs_i$. Therefore:

$$\tau_n \leq \frac{\delta - \varepsilon}{\frac{2\varepsilon}{\sum_{i=n-z}^{n-1} \tau_i} - \rho_R}$$

The asymptotic case corresponds to the fixed point where $\tau_i = \tau_{i-z}$: at that time it holds that:

$$\tau_n = \frac{\delta - \varepsilon}{\frac{2\varepsilon}{z\tau_n} - \rho_R}$$

and

$$\begin{aligned}
\overline{\tau} &= \frac{\delta - \varepsilon - \frac{2\varepsilon}{z}}{\rho_R} \\
&< \frac{\delta - \varepsilon}{\rho_R}
\end{aligned}$$

A.4 Proof of Theorem 2

We separate the cases depending on the validity of $t_a + \tau_a > t_b + \tau_b$.

In the case where it holds, we have that the τ of c_b remains unaltered, while the next reading operation of c_a is anticipated to $t_b + \tau_b$ (see Figure 3).

We show that the following holds:

$$(t_a + \tau_a) - (t_b + \tau_b) < (1 - \xi)\overline{\tau} \qquad (13)$$

Instead we prove the following, that is stronger since $\overline{\tau} > \tau_a$:

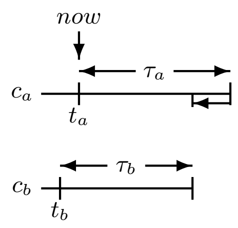


Figure 3. Two sub-cohorts merge, and c_a's τ is shortened

$$(t_a + \tau_a) - (t_b + \tau_b) \quad < \quad (1 - \xi)\tau_a \tag{14}$$

The above can be rewritten as:

$$(t_b + \tau_b) > t_a + \xi\tau_a$$

which is valid, since it corresponds to the merging rule condition.
The next *time to live* of the *virtual clock*of the sub-cohort c_a will therefore be set to

$$\tau_{new} = \tau_a - (t_a + \tau_a) + (t_b + \tau_b)$$

using (13)

$$\tau_{new} > \tau_a - (1 - \xi)\overline{\tau}$$

and, for the merging premises

$$\tau_{new} \quad > \quad (1 - \sigma)\overline{\tau} - (1 - \xi)\overline{\tau} \tag{15}$$
$$\tau_{new} \quad > \quad (\xi - \sigma)\overline{\tau} \tag{16}$$

which proves the assert in the first case.

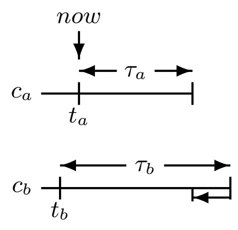


Figure 4. Two sub-cohorts merge, and c_b's τ is shortened

On the other hand, if $t_a + \tau_a \leq t_b + \tau_b$ (see Figure 4), the next reading operation of c_b will be anticipated to $t_a + \tau_a$, and the resulting τ of the current clock of c_b will be reduced to $(t_a - \tau_a) - t_b$. Since $t_a \geq t_b$, the new τ will be longer than τ_a, which in its turn is longer than $(1 - \sigma)\overline{\tau}$, and also longer than $(\xi - \sigma)\overline{\tau}$, which concludes the proof.

A.5 Proof of Theorem 3

We want to prove that as long as there are at least two sub-cohorts, a new merge event will eventually occur.
The successive values of the τ of the cohorts tend to the same asymptotic value: using basic analytical properties of the bounded successions, all of them will eventually enter any interval containing the bound, unless they are perturbed by a merge. So, given any two sub-cohorts, either a merge alters the progression, or the proposition (9) will eventually hold for both of them.

Without loss of generality assume that (9) holds at time t_a for the sub-cohorts c_a and c_b, when c_a is finishing the clock reading, and that c_b performed a clock reading at t_b. We want to prove that, under the condition stipulated by (11), either (10) holds, and c_a merges with c_b, or the the merge will occur at the next reading operation performed by c_b, and c_b will merge with c_a. See Figure 5.

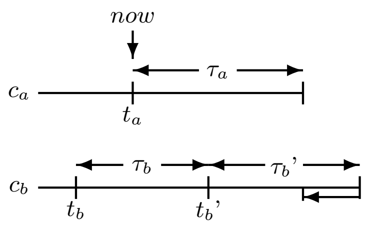


Figure 5. Eventual merge of two sub-cohorts

We start from the hypotheses that inequality (10) is false (otherwise the merge occurs):

$$t_b + \tau_b < t_a + \xi * \tau_a$$

Using the theorem (3), we obtain the following consequence:

$$t_b + \tau_b < t_a + \frac{1-\sigma}{2-\sigma} * \tau_a \tag{17}$$

We prove the following intermediate result: let τ'_b be the next τ of c_b, computed at $t_b + \tau_b$, when the next clock reading is performed by c_b. The proposition (9) will hold for τ'_b too, since the succession of the τ s is monotonic, and:

$$\frac{1-\sigma}{2-\sigma} * \tau_a \leq \tau_a - \frac{1-\sigma}{2-\sigma} * \tau'_b \tag{18}$$

Using some algebraic manipulation, we rewrite it as follows:

$$\frac{\tau_a}{\tau'_b} \geq 1 - \sigma$$

which is valid since

$$\tau_a \geq \overline{\tau}(1-\sigma) \tag{19}$$
$$\tau'_b \leq \overline{\tau} \tag{20}$$

Using inequalities (18) and (17), we obtain that the following holds:

$$t_b + \tau_b < t_a + \tau_a - \frac{1-\sigma}{2-\sigma} * \tau'_b$$

and, from the bound of inequality (11)

$$t_b + \tau_b < t_a + \tau_a - \xi * \tau'_b$$

Now, let us make the substitution $t_b + \tau_b = t'_b$

$$t_a + \tau_a > t'_b + \xi * \tau'_b$$

which, with an appropriate relabeling, matches the condition for merging indicated by inequality (10).

In order to ensure the condition for the merge, we must prove also that $t_a < t'_b$, which is true by the initial construction ($t_a < t_b + \tau_b$ since t_b corresponds to the most recent clock reading of c_b).

This concludes the proof.