Providing a uniform timing to the clients of a distributed service

Augusto Ciuffoletti

Università di Pisa
Corso Italia 40 - 56126 Pisa - ITALY
e.mail: augusto@di.unipi.it
URL: http://www.di.unipi.it/∼augusto

Abstract

We introduce a clock synchronization tool that implements a the gradual tuning of the *virtual clocks* of the clients of a distributed service, in order to compensate the *systematic drift* of their *system clock* with respect to the clock of the server. The algorithm does not affect the *system clock* itself. One distinctive feature of our solution, is that the clients, grouped into dynamic *cohorts*, are synchronized using broadcast communications.

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

1 Introduction

Several timing protocols exist that are structured according to the common hypotheses that the *system clock* is used to ensure a timely coordination of distributed services (for instance, [3, 1, 2, 5, 4]). The enforcement of this assumption is implemented by a separate service, the *timing protocol*, provided by a specialized server that conforms to some internationally agreed standard. We say that the *timing protocol* that originates from this approach is **system clock based**.

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. When the processing of the host is mainly time-dependent these drawbacks must be accepted, but in many other cases the use of the common time reference is limited to a single activity. In these cases, the ability of producing accurate **time-stamps** is sufficient; for this, the synchronization of the *system clock* is not needed since, when the granularity of the time reference is compatible with the speed of the processor, a software *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 synchronization of the *system clock* is avoided, improving the robustness of the service and of the overall system.
- each service may be based on its own time reference, admitting that the timing of distinct services may differ.
- the control of the clock synchronization operation can be shifted from the client to the server.
- for the same reason the server can guarantee a given quality of service to the clients.
- 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*s, 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; a client automatically enters the *synchronization protocol* when it *subscribes* for the service the *synchronization protocol* is associated to.

The algorithm is self stabilizing, in the sense that it tends to the optimal situation (all clients synchronized with one multi-cast) whatever is the initial state: as a consequence, eventual perturbations (caused, for instance, by a new member joining the service) are compensated.

2 Implementation of a *virtual clock*

The local synchronization algorithm consists in the generation of an (infinite) sequence of *virtual clock incarnations* on each client. Each *incarnation* is responsible for the local timing during a certain time interval.

The following definition embeds some basic knowledge about the relation between the value of the *internal clock* t with respect to the *real clock* T:

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

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

We assume that the *drift* $\rho(T)$ is known with some approximation:

Definition 2 (Bounded drift)

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

The hardware constructor usually provides a frequency tolerance ρ_0 and a frequency stability ρ_R of the timing device: the former corresponds to a systematic drift, while the latter describes a term of the drift that depends on time. From these data we derive an initial setup:

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

However, this setup is extremely coarse, and one of the tasks of the *synchronization protocol* is that of refining it.

In order to integrate the definition 1 we need an initial value for t: the host can obtain an approximation of this value by interacting with the server through a *remote clock reading*. Here we define its interface, whose implementation will be detailed later:

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

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

From the definition 1 we obtain:

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*

$$\frac{t(T) - t_s}{(1 + drift + drift_{err})} + rcr - rcr_{err} \leq T \qquad (1)$$

$$T \leq \frac{t(T) - t_s}{(1 + drift - drift_{err})} + rcr + rcr_{err} \qquad (2)$$

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*. Its internal state (summarized in Table 1) is set with the results of the remote clock reading operation, and will never be altered.

The inequalities (1) and (2) state 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. The following lemma 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:*

$$\begin{aligned} vc(T) &= rcr + (t(T) - t_s)(1 - drift) &(3) \\ vc_{err}(T) &= rcr_{err} + (t(T) - t_s) drift_{err} &(4) \end{aligned}$$

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* :

Lemma 2 (Time to live)

$$\tau = \frac{\delta - rcr_{err}}{drift_{err}} \qquad (5)$$

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.

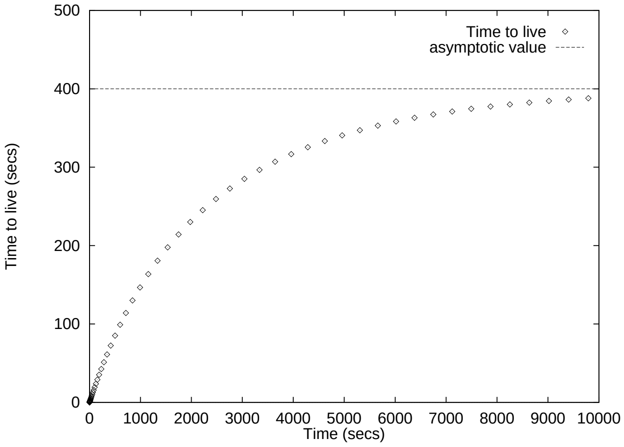

Figure 1: Time to live of successive incarnations

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 :

Lemma 3 (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*

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

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. Figure 1 represents the evolution with the values indicated in Table 2.

3 The remote clock reading operation

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

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.

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 (ρ_0 and ρ_R are typical of an ordinary quartz clock, while δ is tighter than typical multimedia applications requirements)

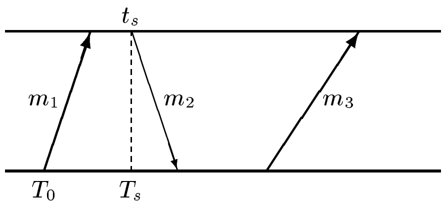

Figure 2: Message exchange during a remote clock reading

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*, that is not yet delivered for use, 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.

After generating the message `m1`, the server waits for the replies from the clients for a limited time, which expires after

$$T_0 + 2\varepsilon$$

The second phase of the protocol consists in the transmission of a broadcast message from the server to the clients whose reply was received in time. The clients that receive `m3` deliver the new *incarnation* and out-date the previous, while the others (if any) conclude that their remote clock reading operation failed.

The server now schedules the next broadcast message, that will trigger another *remote clock reading*, after

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

4 Keeping cohorts compact

To optimize the performance, all the subscribers of the same service should belong to a single cohort: in that case we say the the *synchronization protocol* is *coherent*.

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.

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 4 (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.*

We want that the system tends to coherency, but limiting the cost of the merges. We introduce 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 5 (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:

$$\tau_a, \tau_b \in [(1-\sigma)\overline{\tau}, \overline{\tau}] \quad (6)$$

$$t_b + \tau_b \geq t_a + \xi * \tau_a \quad (7)$$

The following theorem gives a criteria for choosing "appropriate" values for ξ and σ:

Theorem 1 (Bounded degradation) *If*

$$0 < \xi \leq \frac{1-\sigma}{2-\sigma} \quad (8)$$

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

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 :

Theorem 2 (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.*

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 without affecting the *system clock*. This 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] 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.

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

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

[5] 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.