Ordering and Timeliness Requirements of Dependable Real-time Programs

Paulo Veríssimo
paulov@inesc.pt
INESC - IST
Technical University of Lisboa*

Abstract

It is significantly difficult to program dependable decentralised computer control systems, since they need to integrate fault-tolerance and real-time with distribution. In consequence, architects and programmers turn to new distributed system tools and paradigms, such as: synchronised clocks, causal order, groups, membership, replication, and so forth, to assist them in that task. Their inappropriate use, however, may lead to unexpected inefficiency or failure of programs. There is a common link between all these tools in real-time uses, and that is the rôle of time and order in the behaviour of the system. The purpose of this paper is to establish the limits to their use, in order that the models programmers rely on are not used beyond their validity boundaries. This paper equates the time and order problems of real-time programs in a unified manner. It recalls known fundamental limits of time and order in distributed systems, and then it shows that these limits can be drastically relaxed, if the system support (clock granularity, order discrimination) can be matched to the ordering and timeliness requirements of programs. It finalises by giving concrete guidelines about the use of clocks and about what a programmer can expect from an ordering by distributed timestamps.

1 Introduction

Dependable distributed real-time systems are an active field of research [7,19,21]. Not surprisingly, a number of techniques and programming paradigms used in integrated real-time systems display shortcomings in distributed environments. In fact, it has proved to be significantly difficult to integrate real-time and fault-tolerance with distribution. As a consequence, architects and programmers turn to the new tools developed in the recent years in distributed systems, seeking for solutions to build timely, correct and reliable

*Instituto de Engenharia de Sistemas e Computadores, R. Alves Redol, 9 - 6^o - 1000 Lisboa - Portugal, Tel.+351-1-3100281. Fax: +351-1-525843. This work has been supported in part by the CEC, through Esprit Projects BROADCAST and DINAS, and JNICT, through Programs Ciência and STRIDE. Sections 2, 3 and 4 are based on, and an extension of, material appearing in the chapter 17 of Distributed Systems, 2nd Edition, Edited by Sape Mullender, (c) 1993 by the ACM Press in association with Addison-Wesley Publishing Co., by kind permission of the publisher.

programs. Amongst them, are synchronised clocks, causal and total order protocols, group membership management, distributed replication techniques, and so forth.

Their inappropriate use may, however, lead to unexpected inefficiency, or even failure, of programs relying on those paradigms. Synchronous programming languages and environments are popular in critical systems, since they give the programmer the abstraction of a completely synchronous perfect-clock-driven world, advancing in lock-step. In a sense, they ignore the time issue, or assume it is solved by the run-time support, by assuring, for example, that the computation resources have adequate power to fulfil the requirements of the program. However, this model is an abstraction materialised by the operating system, based on clocks and time-triggered lattices, which are imperfect.

On the other hand, perception of external events, reasonably well understoood (metastability notwithstanding) in integrated systems, suffers from the imprecision of clocks, in distributed settings, either time-triggered or event-triggered. In fact, in a newtonian timeframe, events happen in a global total order. Given infinitely precise instrumentation, one could always tell the order of any two events. Since this is not so, the mapping of that order into computer-generated orderings needs to be understood by the programmer. Some events will be considered simultaneous, others may even be ordered inversely. That is, the programmer must understand the limits of the system he/she is working with: what order discrimination can a given system guarantee, depending on the separation of events or messages (can the system give me the order of two events separated by 100ns?); what is the meaning of the ordering of any two events or messages, based on their timestamps (did two events with timestamps $Tb = Ta + 1$ really occur in that order in physical time?).

The purpose of this paper is not to derive new techniques for construction of dependable real-time programs. Although this is still a very active field of research, a number of such techniques exist. However, these tools being made available to programmers, we believe it important to establish the limits to their correct use. Issues of finite precision [4] and granularity [13] have been pointed out earlier on as disturbance factors of theoretical models based on fully-synchronous systems, and perfect and continuous clocks. A model for real-time objects and a discussion on the impact of temporal uncertainties on total order protocols has been advanced in [12]. In [22,26] it was proposed to match the granularity of computer-derived orderings to the pace at which computer or physical processes evolve (δ_t-precedence). A recent work [9] discusses how to match sparse timebases to δ_t-precedent sets of events.

This paper equates the time and order problems of real-time programs in a unified manner. It raises or recalls fundamental limits of time and order in distributed systems, and then, with the help of some innovative concepts (δ_t-precedence and execution granularity), it shows that these limits can be drastically relaxed, if the system support (clock granularity, order discrimination) can be matched to the ordering and timeliness requirements of programs.

The paper is organised as follows. In the next section, we start by motivating the problem of writing programs for decentralised and distributed control of real-time systems, rather than doing it in a centralised way. In sections 3 and 4 we discuss the rôles of time and order in distributed real-time systems, under a unifying perspective, which we believe important for the understanding of the problem by system, protocol and application programmers, and relevant to motivate the rest of the paper. For the purpose, we will

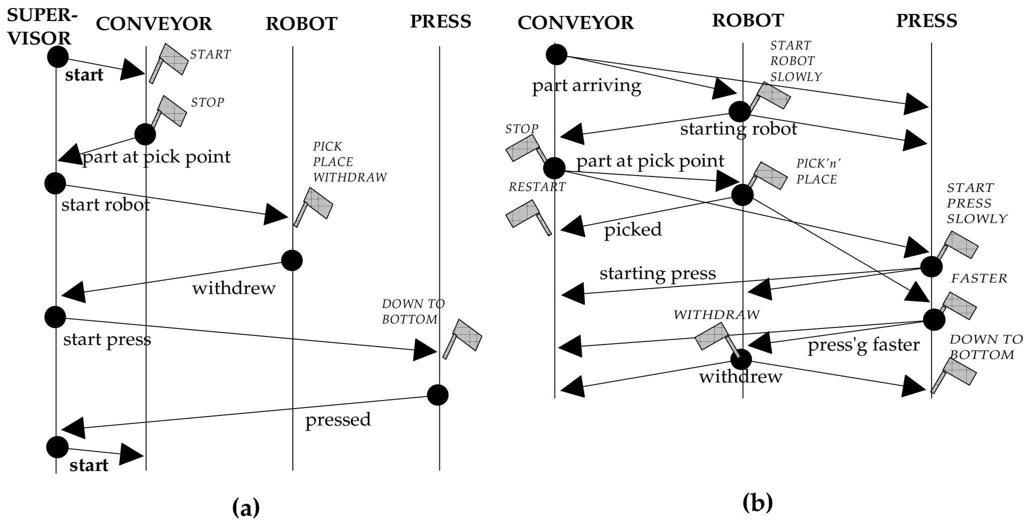

Figure 1: Example of a problem for distributed control: (a) adhoc centralised solution; (b) distributed solution

elaborate on material appearing in [24], expanded with some new results. In order not to distract the reader from the main point, which is the applicability of these techniques, we reduced the formalism to a minimum. We give informal proofs of the results in the appendix. In sections 5 and 6 we discuss how applications can access time and order, and which types of protocols provide them. We assume the reader is familiar with the basic clock synchronisation [20] and causal and temporal ordering [17] concepts. The works cited give an excellent insight into the problem.

2 Distributed control of real-time systems

Traditional computer control of related functions is centralised. The philosophy is to organise the process in a supervisor controlling a set of variables or other controllers. 'Distribution' in computerised control, when it exists, is mostly concerned with replacing point-to-point cabling, yielding the so-called *field buses*, which are a kind of digital system over a long wire: the central unit executes an automaton which reads information from and sends commands to remote units on a polled, synchronised basis.

Figure 1a illustrates a cell composed of a conveyor, a pick-and-place robot, and a press which forms the raw parts brought by the conveyor. For the traditional control engineer, this will be handled by a single automaton, sequentially programmed, things happening one at a time. The centralised control can be done on a network, via blocking RPCs issued by the supervisor: there is a global thread of control which walks through the machines.

Let us imagine how it would be to handle this same task *without* a supervisor. Figure 1b illustrates the idea. The sites are autonomous. They work based on: local algorithms; local information; information they receive from one another, disseminated *(multicasted)* to the group of sites; cause-effect relations; the passage of time.

At this point the reader may ask why distribute, if a centralised solution is simpler and well-known. A distributed system is modular and autonomous. In that sense, it reflects better the natural way an automation system operates. Given the modularity of control, it is easier to expand and modify, thus more versatile. Given the autonomy, it has more parallelism, and is thus more performant and concurrent.

Besides, such a kind of architecture provides the foundation for distributed fault-tolerance. The centralised solution is a single point of failure: if the supervisor fails, the system stops. On the other hand, if it is made reliable, it has to be as reliable as required by the most demanding component of the control system. That may be an overkill in most situations. For example, in figure 1, the press is probably a critical component, because it is capable of damaging the robot. In a centralised solution, the supervisor would have to be made dependable enough to prevent that from happening. In a distributed solution, one might apply what is called *incremental fault-tolerance*, by having replication of hardware and/or software only in the necessary measure [11,25]. In consequence, one could have two or three replicas of the press controller on the network, connected to a voter, to ensure it would act reliably, without having to replicate the other components. Finally, though not in the very simple example of the figure, such a modular distributed system would be capable of reconfiguring and/or exhibiting graceful degradation, upon the failure of some components. With the adequate system support [11,25], the handling of the replicas and the reconfiguration would be transparent to the programmer.

Having made the point about distribution, one must admit, however, that figure 1b also discloses a greater complexity in the interactions between participants. In fact, it is extremely difficult to provide any guarantee concerning dependability of such distributed real-time control software, should it be programmed ad-hoc, over ad-hoc operating and communication system support. In that case, it is most probable that the robot arm will once (and forever more ...) be late getting from under the one-ton press while it goes down, confirming the fears of the control engineer about distribution.

To motivate the reader to our proposal, we will use a few examples in figure 1b to explain very simply how order and timing of events, i.e. knowledge about what caused what and when in the system, can allow its components to operate autonomously. If sites receive sequences of messages about system activity and timing they are able to build a 'picture' of the controlled system state and determine their own actions on a per-message basis. However, with this simple programming technique the press would be utterly confused should it receive the `picked` message before the `part-at-pick-point` message. So, those interactions have to take place in a cause-effect order. Take these examples: if the robot in the figure receives an ordered and timely sequence of messages from the conveyor it can predict its movement and velocity and start moving concurrently; on the other hand, the conveyor can receive the `picked` and the `starting-press` messages interchangeably, without any disturbance. The reason for the different behaviour in the two examples is because messages are causally related in the first, so they can not be delivered out of order, whereas in the second they are not related.

Dependable programming of **distributed** computer control systems should thus rely on concepts and tools pertaining to the realm of:

- a notion of time common to all sites;
- temporal ordering of information;
- real-time and reliability guarantees in communications and processing [10,6,26,23];

- encapsulating constructs for the distributed and cooperating programs, from the timeliness, dependability and functionality viewpoints, such as real-time object groups [12];
- system structure for distributed fault-tolerance and real-time [11,7,25].

The sequel of the paper addresses the first two aspects. The following two sections address the rôles of time and order. The remaining sections discuss the implications of time and order for distributed applications. Since we will be discussing mechanisms, rather than systems, related work will be mentioned as appropriate in the sections concerned. Nevertheless, a few systems of different types are worthwhile mentioning, for having worked with some of the concepts advanced here: MARS, a time-triggered real-time system [11]; the AAS air traffic control system [7]; the DELTA-4 XPA event-triggered real-time system [25]. The reader may find details about reliable real-time communication and processing, and about distributed fault-tolerance and real-time, in the works cited above.

3 The rôle of Time

Computing systems often use some notion of time, for example, for timeouts. These *relative time measurements* can be achieved by local **timers**, even in real-time systems. There is little to be said about the useful rôle of timers in distributed real-time systems, except that the rate of all timers at the several sites must have a bounded deviation from real-time, and thus from each other, during the execution of the longest distributed action or protocol. This attribute is normally considered acquired by programmers, although timer implementations are sometimes inaccurate and thus inadequate for real-time use.

Timers are unfortunately not enough for *absolute time measurements*, that is, the need to locate the position of events in the timeline of a reference timebase. A **clock** representing that timebase is necessary. When talking about distributed real-time systems, clocks become also crucial for relative measurements, since they are the only way of measuring *distributed durations*: something that starts in a site and ends in another. In both cases, the notion of time, to be useful, must be global. To be reliable and precise, it should also be available locally, and in a decentralised fashion, by means of local clocks which are periodically adjusted to remain close to one another and, sometimes, to an external reference. This is called clock synchronisation and will be referred to in section 5.

3.1 Agreement on time

Participant agreement on time, that is, the common knowledge that applications can have about a **global time** in the system, is a fundamental paradigm of distributed real-time programs, let alone centralised ones. It is best achieved through synchronised local clocks, for the reasons referred above, and it has two aspects:

- agreement on the time at which events occurred.
- agreement on the time to trigger actions;

The first is concerned with the *distributed recording (logging) of events*. Such a distributed log is instrumental to determine precedence relations through temporal order of sets of events (further discussed in section 4). This is used, for example, for the a posteriori study of alarm situations in power stations/networks or in process control plants, in order to determine causes, trace propagations, discover spurious triggers.

The second is concerned with the *synchronisation of the concurrent progress of a system*. For example, to disseminate the time to trigger both a change of points and a change of lights in a railway crossing, by a distributed group of actuator representatives. The specified time may be absolute, or it may be relative: the sender may specify "change points at 5:03:35" or "[...] within 1sec of change of lights". The latter distributed duration is easily handled by global time, provided that the lights-change instant is disseminated.

Slightly different but important to mention is *replica agreement on time*. A replicated computation which reads clocks to make time-dependent decisions, should do so in a manner that all replicas get the same time, since synchronised clocks are not bit-by-bit identical.

3.2 Timestamps

Events may be ordered with timestamps from a physical clock of granularity g_p — the clock 'tick'. The latter is normally a small time interval, in consequence, we would end-up with a very fine-grained ordering. However, timestamps must be allocated in a distributed way, so this must be done from a *virtual* global clock, formed by the set of local clocks. These are not exactly equal: they differ at most by the clock *precision*, π [20], that is, the greatest amount by which any two clocks may be separated at any time. This leads us to the following remark, with regard to the use of clocks for distributed timestamping by real-time programs:

Remark 1 *Given two events a and b in different sites, timestamped by a global clock of granularity g_p, precision π, their temporal order can be guaranteed to be derived from the respective timestamps*[1] *only if $|t_b - t_a| \geq \pi + g_p$.*

This result is a generalisation of Lamport's result in [15], from continuous clocks to events timestamped from granular clocks.

A *granularity condition* was introduced in [9]. If the virtual global clock were to have the same (fine) granularity of the physical clock, the same event might receive timestamps distant more than one tick, due to the effect of precision. While not incorrect per se, any granularity below precision is not significant, very much like a weighing-machine accurate to the gram giving weights in milligrams. A virtual clock is a software representation of the physical clock, so it can easily have any granularity $g_v = p.g_p$, p integer. So the granularity condition is translated into the following rule:

Rule 1 *To ensure distributed timestamps of the same event are at most one tick apart, the granularity g_v of the global timebase must not be smaller than precision π: $g_v \geq \pi$.*

Although stated as $g_v > \pi$ in [9], it also holds for $g_v = \pi$. We leave the demonstration to the reader. Fulfilling the granularity condition is equivalent to obliging the same virtual clock tick interval to have some overlap in all sites.

[1] If $|t_b - t_a| < \pi$, there may even be order inversion.

4 The rôle of Order

Consider a distributed system such as the one in figure 1b: it is formed by processors with no shared memory, no centralised control or clock, communicating only by exchanging messages. It would be easy for an external (omniscient) observer to monitor the evolution of the whole cell, so to speak *visually*. However, individual participants, being *inside* the system, are subject to the uncertainty of its observation, since the latter is performed by exchanging messages whose transmission delay and delay variation are not negligible.

This fact is a scaled-down version of the phenomena explained by the Relativity Theory. In fact, given two events a and b, a can only be causally related to b, in the measure where an information departing from a site where a occurred, arrives at the site of b before b occurs. In other words, in a three-dimensional space-time diagram where a occupies the vertex of an inverted "light cone" disposed along the time axis, for a to be said to precede b, b must be inside the cone, as very well exemplified in [8]. An event a in a computer system is therefore said to **precede** an event b $(a \rightarrow b)$ in those conditions [15]. Given this space-time relation, it may occur that neither of them can potentially cause the other, in which case we say they are *concurrent*, i.e. $\neg(a \rightarrow b \vee b \rightarrow a)$. The consequence is that they can appear as simultaneous, for example, by receiving the same timestamp, to a computer program. If the events represent message arrivals, they can also be delivered to the computer program in different orders. None of these cases would mean an incorrect behaviour, from the point of view of precedence.

Note that a genuinely causal order is impossible to establish a priori. So the best that can be done is order what we think *may* have a causal relationship, that is, to secure a *potential* causal order, which is exactly the one defined by the precedence relation. There are several ways of guaranteeing precedence or *potential causal* order — sometimes just called *causal* order, for simplicity — in a distributed system. The most widely known in distributed computing is through a *logical* order on the messages exchanged between participants in a distributed computation [1]:

> **Logical Order:** A message $m1$ is said to logically precede, $(\xrightarrow{l})$, $m2$ **if**: $m1$ *is sent before* $m2$*, by the same participant* **or** $m1$ *is delivered* [2] *to the sender of* $m2$ *before it sends* $m2$ **or** $m1 \xrightarrow{l} m3$ *and* $m3 \xrightarrow{l} m2$.

In consequence, each participant will observe system evolution as a sequence of events coming from the various sites (space-time view). The sequence may not be the same for all, due to the relativity of their positions: a **partial ordering** on events. This is not disturbing though. The cause-effect relation is the natural order of events in a system [3].

Going back to figure 1b, the sites can operate autonomously if they know what caused what and when in the system, especially about events that concern their own activity. We hope to have made clear that, for this purpose, information exchange must be reliable, and must preserve the ordering of any two messages such that one *could* have caused the other, that is, messages that are potentially-causally related. The most well known logical precedence implementations are the "causal broadcast/multicast" type of protocols using: logical clocks [15], piggybacking [1], context graphs [18], or vector clocks [2].

[2] We distinguish *received*, the act of a message arriving at a site, from *delivered*, the act of it being given to the participant, since those instants may not be equivalent, for some ordering protocols.

[3] Or in the universe, for that matter.

These protocols are non real-time though, and subject to anomalies, as discussed in the next section. Such a system, besides the concern about its dependable programming, must also have system-level fault-tolerance, which is outside the scope of this paper, but relevant examples may be found in [11,7,19].

4.1 Anomalous behaviour

Logical order is based on a simple observation: if participants only exchange information by sending and receiving messages, they can only define causality relations through those messages. Two situations where this principle is violated and an anomalous behaviour may occur:

- when participants exchange messages outside the ordering protocol;
- when the system interacts with the outside world.

In both cases there are *hidden* channels, that is, information flowing between participants which is not controlled by the ordering discipline, so to speak, taking place in a *clandestine* manner. The first anomaly is well-known, and its most common example is the mixed use of a protocol for logical order and another protocol, for example, RPC. It is shown in figure 2a: m_2 is issued because of the RPC that the top sender executed, so it is preceded by m_1; nevertheless, for the protocol they are concurrent and m_2 happens to be delivered before m_1; m_3 may carry the undesirable effects of this violation.

A less known but extremely common situation typical to real-time systems [25] is concerned with the interaction between the computer system and the outside world, when the hidden channels develop by means of feedback paths through the controlled process. Distributed real-time systems are obviously prone to this anomaly, since the effect of one controller's actuator can propagate through the environment to, say, a sensor located in another controller, competing, and possibly conflicting, with regular messages passed between controllers. *There is no way that logical order implementations know about these paths*. Figure 2b presents such an example. R_1 and R_2 represent some real-time entities residing in the physical process under control, PHY: for example, R_1 controls a valve actuator, R_2 handles a fluid sensor. CE and SUP are computing elements, SUP being a supervision unit which detects anomalies and handles alarms. CE issues an output command message to R_1, which in turn issues the physical actuation command (c_1). A notification event is generated in some other part of PHY in consequence (n_1, in this case, fluid detection). This event is read by R_2, who in response sends m_2 to notify SUP. Suppose the original message m_1 from CE meant `open-valve`, and that it got delayed in communication, arriving later than m_2 to SUP (the logical order protocol does not recognise any relationship between them). Given that m_2 means `fluid-flowing`, SUP will most probably issue a `leakage!` alarm, whereas the system is functioning perfectly.

4.2 Temporal order

The solution for these situations consists in securing precedence through *temporal order*, for example with timestamps, as discussed in section 3. Applying Rule 1 affects Remark 1 as follows:

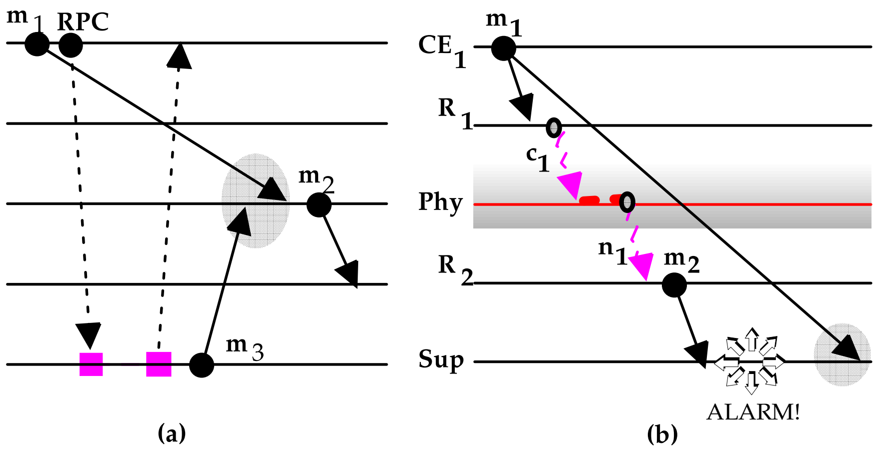

Figure 2: Anomalous behaviour examples: (a) RPC; (b) physical feedback

Remark 2 *Given two events a and b in different sites, timestamped by a global clock of precision π, and granularity $g_v \geq \pi$, their temporal order can be guaranteed to be asserted from the respective timestamps only if $|t_b - t_a| \geq \pi + g_v$.*

The tightest value for g_v, $g_v = \pi$, yields $|t_b - t_a| \geq 2g_v$, the condition stipulated by the authors of [9] to remove what they called *imprecision effect*. Remark 2 generalises this formulation for values of g_v other than π, making it precise for all situations. This is relevant when, as will be shown ahead, granularity gets to be much greater than precision. In that case, i.e. considering $g_v \gg \pi$, our expression shows that the order is as fine as $g_v + \pi \simeq g_v$, instead of $2g_v$. Figure 3 illustrates Remark 2, by depicting the limit situation: the clocks $C1$ and $C2$ of two sites are separated by π; a occurs immediately after $tick$ 1 in $C1$. For b to be ordered after a, it must receive a timestamp in $tick$ 2. In consequence, the figure displays the minimum separation between events so that they are ordered, when clocks are worst-case separated by π. An informal proof is given in appendix.

At this point, the temporal order problem is formulated, and the limitations of its solution in distributed systems, established in Remarks 1 and 2. In consequence, the programmer can obtain simple orderings of events by means of timestamps, for example in instrumentation, and extract conclusions about their order of occurrence in physical time, depending on their minimum separation, as will be discussed in section 6.

However, another very important use of temporal order in distributed control, as we pointed out in section 2, is to determine potential causal relationships among a particular type of events: messages exchanged by the participants in a distributed computation. Given the precision achievable by clock synchronisation in real-time systems, which are normally LAN-based (order of milliseconds, can reach order of microseconds), one will be ordering too much in most of the settings, vis-a-vis the objective of representing potential causality. In a distributed computer system or in a physical process, it takes a finite amount of time for an input event to *cause* an output event: the time for a piece of information to travel from one site to the other; the response time of a computer process; the feedback time of a control loop in a physical process. Supposing δ_t is a lower bound for such intervals in a given system, only two messages separated by more than δ_t may be causally related in that system.

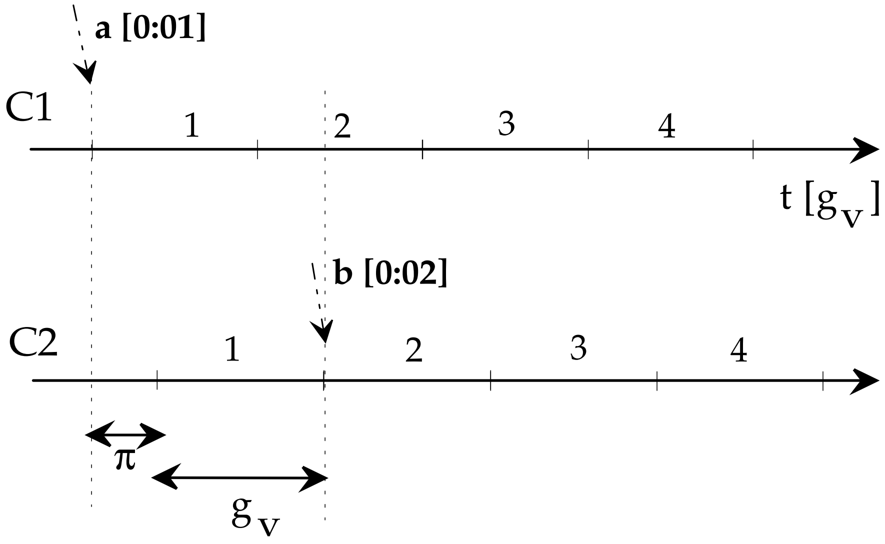

Figure 3: Limit situation for Remark 2

In consequence, it is of no utility to order messages with a shorter separation, to represent potential causality. A less stringent ordering on messages is benefitial for system performance and concurrency. To understand why in a brief explanation, consider firstly that ordering protocols incur a cost, in message complexity and/or delay, to achieve ordering. The finer the ordering granularity, the greater that cost. Secondly, for the control program, it is different to receive ten unrelated (concurrent) messages, which can then be processed by independent threads, or to receive the same ten in some order, which requires them to be processed one at a time.

4.3 The δ_t-precedence order

We recall a definition that will help us show how to take advantage of these facts in real settings [22,26]:

> δ_t**-precedence order** ($\stackrel{\delta_t}{\rightarrow}$): *An event a is said to δ_t-precede an event b, $a \stackrel{\delta_t}{\rightarrow} b$,* **if** $t_b - t_a > \delta_t$.

Since δ_t is the minimum real time interval for causal relations to be generated, δ_t-*precedence* is the criterium for potential causality (which will not exist for smaller differences). The definition is helpful on the system support and on the application requirements side, as we explain below.

On the system support side, the δ_t-precedence concept is a suitable definition of how fine-grained is the temporal order provided by a protocol. Let us call *discrimination* to the minimum real time interval between the sending of two messages so that a protocol is capable of correctly ordering them, by their order of occurrence in physical time. Clock-driven protocols, such as the one in [4], are known to achieve total temporal order using a global timebase. However, since the clocks are digital, and thus granular, they have a limitation, sometimes overlooked, which derives from Remark 1:

Remark 3 *The discrimination of clock-driven protocols cannot be better than $\pi + g_p$.*

If Rule 1 is followed, we may also conclude:

Remark 4 *Clock-driven protocols of granularity $g_v \geq \pi$ enforce* $2g_v$-precedence, $\stackrel{2g_v}{\rightarrow}$.

On the application side, δ_t-precedence is a suitable definition of ordering requirements. In a message-passing model of computation, such as the one depicted in figure 1, distributed real-time programs receive input messages, perform execution steps within bounded delays (deadlines), and exchange messages in result of those computations. In consequence, one can define the minimum time interval it takes for a process to issue a reaction to an input. Generalising for a system:

> **Execution granularity:** Execution granularity of a system, x_g, is the minimum time for an input to *cause* a response, in any process of the system.

Let us consider a set of events δ_t**-precedent**, if for any non-ordered pair of events, either they are concurrent, or they satisfy the relation $\stackrel{\delta_t}{\rightarrow}$. For the purpose of ordering messages produced by x_g-granular real-time programs, one can consider they form an x_g-precedent set[4]: for any message to cause another, it has to undergo a processing step, so that the interval between those messages is greater x_g.

The execution granularity of computer processes is normally much coarser than clock granularity. In consequence, one can use a new virtual clock whose granularity may be relaxed, improving the concurrency of the system, and that leads us to a major result of this paper, Rule 2 below:

Rule 2 *Given any two potentially causally related events a and b deriving from program computations in a system[5] with execution granularity x_g, timestamped by a global clock of granularity g'_v and precision π, their temporal order is guaranteed to be asserted from the respective timestamps if $g'_v \leq x_g - \pi$.*

Figure 4 illustrates the situation secured by implementation Rule 2: the clocks $C1$ and $C2$ at two sites are separated by π; a occurs immediately after $tick$ 1 in $C1$; the causality path (through the communication system or through the environment) is, by definition of granularity, x_g. Granularity implies that the earliest event that may be causally related to a must occur at t_b or later. Now notice that all events occurring at or after t_b are ordered with regard to a: virtual clock granularity is chosen in order that $x_g = g'_v + \pi$, so event b is indeed timestamped with the tick after the one of a. It is easy to see that the rule holds for events in the same site. An informal proof of this rule is given in appendix.

As mentioned before a propos Remark 2, this result is more precise than the one presented in [14] and [9]. Bringing application requirements and system support characteristics together around the unifying δ_t-*precedence* concept, yields the important new result formulated by Rule 2. The latter comment is made more evident by the following remark applying to message ordering protocols:

---

[4]For the sake of modeling, and without loss of generality, we consider the minimum communication delay as zero, and we consider a finite message set-up time after x_g elapses. We also consider a system-wide x_g, to make our explanation simple. In real settings, one may have several groups of non-causally related entities, each one with a different x_g.

[5]For example, message `sends` and `receives`.

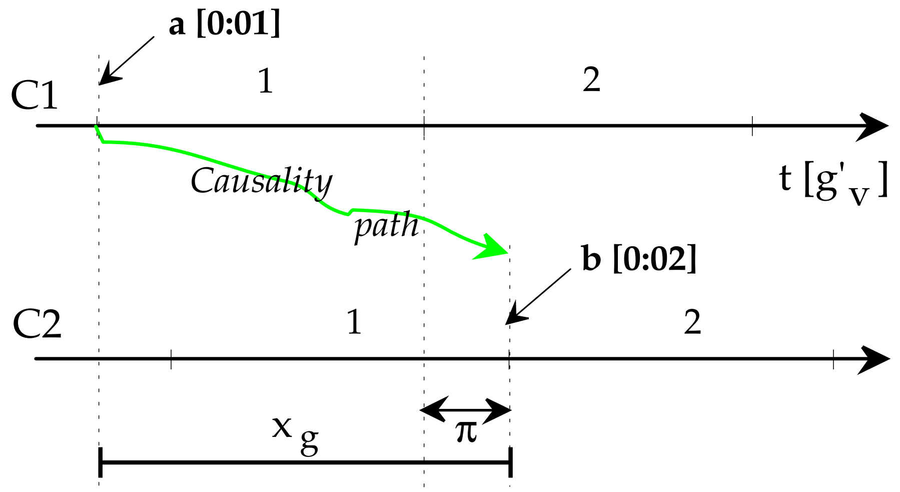

Figure 4: Limit situation for Rule 2

Remark 5 *The discrimination of temporal order protocols need not be better than execution granularity,* x_g*.*

Temporal order protocols have been disputed on the grounds of ordering too much and thus reducing system concurrency, since only messages with the same timestamp are considered concurrent. Informally, one might say that they do not represent causality *faithfully*. That would be true of conventional protocols, using clock granularities that are either the physical ones, or of the order of precision. However, by taking execution granularity into account, the x_g*-based* implementation of precedence has the following advantages:

- it is a more faithful representation of causality than approaches with granularity g_p (physical) or π (precision);
- it is free from anomalous behaviour, unlike logical order implementations [15,1,2].

In conclusion, a system implemented according to Rule 2 yields correct orders and improves concurrency of the system, because it does not order unnecessarily, with regard to the application requirements.

4.4 Non-sparse timebases

In [9], the concept of δ_t-precedence is applied to time-triggered systems. In these systems, activity occurs in synchrony with *global ticks*. For events under the control of the system, it may be possible to make them happen between global ticks, so that ordering becomes trivial in such a sparse timebase. The cited paper details that situation.

When timing of events is not that well controlled, a denser timebase results. But it is still tractable, if the results of the previous sections are followed, as we show below.

A dense timebase occurs as a typical situation of sensor perception when the real-time entities [14] to be measured are not controllable by the computer system. However, physical processes have a process (or *environment*) granularity as well. This allows us to extend the results of [9] to a non-sparse timebase which is not infinitely dense, if it has the mapping to granularity developed in the previous section. The lattice is still tractable in terms of ordering: if the lattice has a period of g_l, a $\stackrel{g_l+\pi}{\rightarrow}$ temporal order is secured.

In consequence, the results of the previous sections allow us to define how large the lattice can be: considering x_g as the granularity of the process being handled by the lattice, we can enlarge the period of the lattice up to $g_l = x_g - \pi$.

Remark 6 *A lattice with a period of $g_l = x_g - \pi$, secures an $\stackrel{x_g}{\rightarrow}$ temporal order.*

The larger g_l, the less significant the skew of the global tick (π) becomes, in the overall period. That is, the percentage of events that occur between global ticks — and thus receives the same timestamp — increases. This brings the lattice nearer to the ideal sparse timebase situation mentioned above.

4.5 Total order

If to temporal order we add the fact that actively replicated processing requires total order by default, the general ordering requirement for dependable and distributed real-time computing would be *total temporal order*. Kopetz *et al.* do an analysis of the timing implications of achieving total temporal order in several types of systems [12]. There are a number of existing real-time total temporal order protocols [4,10,6]. However, total temporal order can be expensive, both in performance and concurrency. Messages are either restrained to fit in the nodes of a lattice, and thus they can be delayed for a whole period, or have delivery delays of the order of the worst case transmission time. Despite that, it is often required in time-critical applications, and namely in the periphery of the system, in what concerns arriving events and some time-triggered actuations.

However, part of the computation in some distributed applications, consists of exchanging messages between replicated components inside the system. In some cases, it may even entail producing outputs to a set of replicas of a component (a server, a display) that need not be tightly synchronised. Furthermore, at that level there is frequently no causal relationship between the computation modules producing the messages, or if there is, it is confined to those interchanged messages (no hidden channels thus). The communication requirements of these special cases are confined to: (i) real-time (bounded delay) message delivery; (ii) total order; no precedence order or, if needed, just a logical order. A clock-less real-time protocol providing total logical order and bounded termination delay will prove effective in these cases, exhibiting faster termination in the no-fault cases [26].

Programmers should be aware of the simplifications that can sometimes be made on the ordering needs of applications. Relaxing the order requirements will be further discussed in section 6.

5 Time as seen by applications

We will now discuss how real-time programs can or should access time in a distributed system: timers, clock synchronisation, global time, and replicated reading.

Concerning timers, most systems have support for a `start` and `kill` service. Additionally, they should provide applications with the possibility of linking the execution of a function to the occurrence of timeout. This is the most precise way of taking an action upon a timeout. Finally, the correctness of some real-time programs depends on the ability of *pausing* timers. This is frequent in adaptive real-time protocols and requires the additional `suspend` and `resume` services.

Before talking about clock services, we will discuss clock synchronisation. There are a number of protocols for clock synchronisation, displaying several attributes. Writers of real-time programs must be concerned about the quality of the time service provided by the system. In principle, all clock synchronisation algorithms have the property of never setting back any clock, and of spreading whatever adjustment needed during the next period [20]. Some algorithms are probabilistic or statistic [5,3], with possible implications in the correctness of hard real-time programs. Most existing algorithms perform *internal* synchronisation, guaranteeing a precision bound. However, a real-time system is one that interacts with the environment. The real world uses standard references of absolute time and whenever the system operation must take them into account, the system must guarantee that its timebase is also *externally* synchronised [13]. This is called accuracy. One purpose of keeping clocks accurate is to be able to relate (and order) events in two separate real-time systems which are accurate to the same external time reference. As a matter of fact, clocks that are accurate are also precise, since the relative precision of clocks with α accuracy is $\pi = 2\,\alpha$.

Discussing now the global time services, the basic one available to programs is obviously `get_time`. However, the correctness of real-time programs can be influenced by the way that is implemented. For best results with regard to freshness, time should in fact be acquired by *reading shared memory*. Dedicated systems achieve the best results by reading time and timestamping buffers and messages via hardware coprocessors [13]. Finally, real-time programs, especially control ones, must have the ability to efficiently schedule the execution of functions at a given future time, such as `exec_at(hour,function)`.

Finally, replica agreement on time, in fault-tolerant real-time systems. Several approaches may be taken on this, depending on the replication model and semantics. A detailed discussion is given in [14]. Actively replicated programs can perform local `get_time` readings and run a consensus protocol, to reach a single value. In semi-active or passive replication, the primary or leader can impose its local value on the other replicas. With some system support for distributed replication, such as the one described in [19], program replicas can access time transparently from replication, though not transparently from clock access. This means that clock access by replicas is not made through the default system primitive, but through a local *time server* which offers the same interface as the system's `get_time`. All the processing concerning replica agreement is performed underneath and thus transparently.

One of the main uses of time by programs is timestamping for ordering. This is a delicate issue in distributed systems, since the programmer must always be aware of the limitations of timestamp-derived orders. In section 6 below, we will discuss the implications of clock precision in the order of events and messages.

6 Order as seen by applications

In this section we discuss how real-time programs can or should obtain order in a distributed system: logging order of events; ordered dissemination of messages and/or actions.

Logging the time at which events occur is easily done by timestamping as described in section 5. However, the a posteriori analysis deserves some precautions by the programmer, deriving from the Remarks made in the previous sections. The same is true of the relative order of message interrupts[6] disseminated in temporal order.

The order of the messages processes exchange when communicating with one another is also relevant. In a time-triggered system [11] message dissemination takes place in the nodes of the timebase lattice, hopefully sparse. In an event-triggered system [25], messages occur without any instantaneous constraint. By instantaneous, we mean that a single observed message can indeed take place at any time. However, if truly real-time behaviour is to be observed, then a limit on the message sending pattern must be observed by definition and design of the system, to avoid the well-known *event showers*. Such a pattern is called sporadic. One way of imposing a discipline for the dissemination of a sporadic arrival pattern is to select sending points over the nodes of a time-triggered lattice, albeit in an irregular fashion. The subject is further discussed in [14].

The time-triggered sparse timebase with large granularity is also very useful for real-time output: while temporal order is secured, tightness (i.e., simultaneity) of action is improved, since the system acts at synchronised points with small skew.

Temporal order in an event-triggered system is normally achieved with clock-driven protocols, for example with a -protocol [4]. -protocols derive their name from the fact that messages, timestamped T_s when they are sent to the network, are delivered to all destinations at $T_s +$. The delay is a constant of the system, and is made greater than the sum of the maximum transmission time and the precision of the clocks.

In time-triggered systems, temporal order is trivially obtained by having all sites send their messages at the nodes of the lattice. Messages sent at a given time are all concurrent, and precede the messages to be sent at the next node [9], and so forth.

The programmer should be aware of what the order protocols can provide to him, to avoid any surprises. In the next sections we discuss what the programmer can conclude from a timestamp-derived ordering, how to relax order, and how to benefit from sparse timebases.

6.1 Telling order from timestamps

Next, we are going to analyse what can the programmer conclude from a distributed-timestamp derived order, on several occasions. Two views can be taken on this: (i) what order discrimination can a given system guarantee, depending on the separation of events or messages; (ii) what is the meaning of the run-time ordering of any two events or messages, based on the respective timestamps.

The tables in figures 5 and 6 describe the two decision processes. The reader will note

[6] An event collected in the periphery of an event-triggered system which is converted into a message with temporal information about it and delivered to the centre of the system [14].

Event separation (δ_t)	Ordering guarantee
$\delta_t \geq \pi + g$ approx. ($\delta_t \geq 2g$)	ALWAYS IN CORRECT ORDER
$\pi \leq \delta_t < \pi + g$ approx. ($g \leq \delta_t < 2g$)	EITHER SIMULTANEOUS OR IN CORRECT ORDER
$\delta_t < \pi$ approx. ($\delta_t < g$)	ANY ORDERING POSSIBLE $\mid$ TIMESTAMP DIFFERENCE $\mid \leq 1$
$\delta_t = 0$ (or same event!)	ANY ORDERING POSSIBLE $\mid$ TIMESTAMP DIFFERENCE $\mid \leq 1$

Figure 5: A priori ordering of events (or messages) separated by varying intervals, with distributed timestamps

Relation between timestamps (ts_i)	Possible event separation ($\delta_t^{21} = t_2 - t_1$)	
$Ts_2 = Ts_1$	$0 \leq \lvert\delta_t^{21}\rvert < g + \pi$ approx. $0 \leq \lvert\delta_t^{21}\rvert < 2g$	NEVER $2g$-PRECEDENT
$Ts_2 = Ts_1 + 1$	$-\pi < \delta_t^{21} < \pi + 2g$	
$Ts_2 = Ts_1 + 2$ $Ts_2 = Ts_1 + 3$	$g - \pi < \delta_t^{21} < \pi + 3g$ $2g - \pi < \delta_t^{21} < \pi + 4g$	ALWAYS IN PHYSICAL ORDER
$Ts_2 = Ts_1 + 4$	$3g - \pi < \delta_t^{21} < \pi + 5g$	ALWAYS $2g$-PRECEDENT

Figure 6: Run-time ordering of events or messages, based on distributed timestamps

the importance of both of them: the first is concerned with the fundamental limits on order discrimination, it tells the programmer how two given events may be ordered by the system, depending on their separation; the second is concerned with what the system allows the programmer to know about the way it orders events, with regard to their physical order. This means that although the programmer has guarantees about ordering of events, he/she may not be able to tell their physical order precisely from timestamps, in run-time.

The meaning of *correct order* in the tables is: events have different timestamps, in the order of their physical occurrence. The meaning of *simultaneous* is: events have the same timestamps. The approximate expressions in parenthesis account for the limit situation where $g = \pi$. Precision is denoted by π, and the granularity assumed for the clocks, whatever its value, by g. For simplicity, the legends and the text below refer to events only, although all applies to timestamped messages as well.

Some conclusions may be drawn from the figures. If precision is of the order of magnitude of granularity ($g \simeq \pi$), one can make some easy-to-remember simplifying assumptions:

- ANY TWO EVENTS SEPARATED BY AT LEAST $2g$ ARE CORRECTLY ORDERED;
- ANY TWO EVENTS SEPARATED BY AT LEAST g BUT LESS THAN $2g$ ARE NEVER INVERSELY ORDERED, BUT THEY MAY RECEIVE THE SAME TIMESTAMP;
- TWO EVENTS SEPARATED BY LESS THAN g MAY RECEIVE THE SAME TIMESTAMP OR BE ARBITRARILY ORDERED BY CONSECUTIVE TIMESTAMPS.
- THE SAME EVENT OBSERVED AT TWO SITES MAY RECEIVE THE SAME TIMESTAMP OR BE ARBITRARILY ORDERED BY CONSECUTIVE TIMESTAMPS.

Conversely, when events can have any possible separation, there are fundamental limits to order discrimination by a program, which one — simplifying again — can draw from the table in figure 6:

- EVENTS WITH THE SAME TIMESTAMP ARE GUARANTEED TO BE CONCURRENT (NOT $2g$-PRECEDENT);
- ONLY EVENTS WITH TIMESTAMPS SEPARATED BY AT LEAST 2 ARE GUARANTEED TO BE IN THEIR PHYSICAL ORDER.
- ONLY EVENTS WITH TIMESTAMPS SEPARATED BY AT LEAST 4 ARE GUARANTEED TO BE $2g$-PRECEDENT.

Events with timestamps separated by one may be anything, from $2g$-precedent to ordered inversely to their physical order, but *never inverting* the $2g$-precedence order[7]. In consequence, timestamp ordering has a limitation similar to that of the logical clocks of Lamport [15]: δ_t-precedent events receive different timestamps in correct order, but the inverse implication is not true, concurrent events may be ordered by timestamps.

Logical clocks are sequence counters maintained by each site, used to order messages. Vector clocks [16] consist of vectors of logical clocks. Protocols using vector clocks [2] include the local vector in each message sent, and messages are delivered in an order resulting from the comparison of the local vector with the one in the received message. Vector clocks establish the equivalence relation for logical order as defined in section 4, that is, they order *exactly* the messages potentially causally related, and not more. But this only works for non-real-time systems, and plus, in systems without hidden channels, as pointed out in that section.

There is no solution for this problem with temporal order without constraining the system activity. This is not incorrect, since by definition of partial order, concurrent events may be ordered in any manner. However, it limits concurrency of the system. The results of the previous sections allow relaxing temporal order as much as possible without more knowledge about applications than its granularity. The next section discusses other possibilities of further relaxing order.

[7] That is, if ordered contrary to physical order of occurrence, it is because their difference is not greater than $2g$.

6.2 Relaxing order

There are several opportunities for the programmer to relax the ordering needs for the messages exchanged in the system, and they should be taken whenever appropriate. Below, we give some examples of situations when the programmer can make tradeoffs between the several ordering disciplines available from the system support:

- *specific knowledge about the application semantics*;
- *particular computational model*;
- *particular replication model*;
- *several ordering paths to the same destination*.

Application semantics can be used by the programmer in the choice of the protocol. For example, knowledge that the senders are a priori concurrent, allows using a FIFO order protocol instead of a causal one, if the underlying system provides several qualities of communication service [26].

The computational model may be restrictive enough to make a causal protocol unuseful. If the system is single-threaded, only evolving through blocking RPCs, for example those coming from the supervisor in figure 1a, the ordering is established by the trajectory of the thread of control, and is purely sequential, no concurrency exists. Multi-threads, even if used with RPC, create opportunities for trading-off concurrency with order.

In real-time fault-tolerant systems, active replication is normally used. For the replicas to be deterministic, i.e. produce the same outputs, they should receive the same inputs in the same order [19]. This would require a total order protocol, as described earlier. However, in some replication models, a privileged replica exists (the primary, coordinator or leader). One of these models, the leader/follower [25], is particularly well-suited for dynamic real-time applications. The leader imposes its own processing order on the other replicas through a private replication management protocol, obviating the need to secure a total order in communication among processes. That is, when sending competing requests to a replicated leader/follower controller, there is no need for ordered communication.

The last point is one of the most important issues for protocol builders. There may be several unrelated ordered flows of messages into the same recipient, for example, urgent alarm messages, themselves ordered, but which can overtake others freely, breaking an otherwise global order into *incomplete* orderings, that is, orderings on subsets of the messages received. In logical order protocols, one may restrict the ordering discipline by using *groups*, or special *labels* in the send request, so that only messages for the same group, or with the same label, are mutually ordered [1]. Temporal order protocols will tend to reason in terms of ordering all messages by timestamps. However, every system has concurrent activities which have no ordering relationship. Messages concerning these different activities may reach the same set of participants. Since the fact that they are not related is not known to an underlying temporal order protocol, it will order all of them. Although the thorough discussion of a protocol would be outside the scope of this paper, we suggest a very simple protocol whereby outgoing messages belonging to different activities (temporal order flows) are labeled differently (one label per activity). Thus, the protocol might maintain several queues, one per label, and in each queue follow the ordering discipline, independently from the other messages.

Relation between timestamps (Ts_i)	Possible event separation ($\delta_t^{21} = t_2 - t_1$)	
$Ts_2 = Ts_1$	$0 \leq \|\delta_t^{21}\| < g - \pi$	NEVER g-PRECEDENT
$Ts_2 = Ts_1 + 1$	$\pi < \delta_t^{21} < 2g - \pi$	
$Ts_2 = Ts_1 + 2$	$\pi + g < \delta_t^{21} < 3g - \pi$	ALWAYS g-PRECEDENT

Figure 7: Run-time ordering of events or messages, based on distributed timestamps from sparse timebases

6.3 Improving order precision with sparse timebases

Another track is the one followed by pure time-triggered systems [11]. They constrain occurrence of events and messages to specific instants in a timebase lattice synchronised with the clock. This leads to a sparse timebase as discussed earlier. Using the results in [9], we derive a new content for the table in figure 6, which we present in figure 7. Citing [9]: suppose events and actions only occur between the last local clock tick of node i of the lattice, and the first tick of node $i+1$. Then, all events in a same cycle of the lattice receive the same timestamp, and are concurrent. Likewise, any two events of consecutive cycles are separated by one tick. The consequences are depicted in figure 7.

As already pointed out by Kopetz [9], this programming style drastically simplifies the temporal relationships in a distributed real-time system. However, it is not always possible for the computer system to control the environment, in order to condition perception and actuation times. In those occasions, the system must take that event-triggered nature into account, and the programmer, event if using a time-triggered lattice, as discussed under non-sparse timebases in section 4, must rely on the contents of the table in figure 6. We anticipate that distributed real-time systems will tend to have a hybrid time- and event-triggered behaviour, to cope exactly with the varying requirements just enumerated. In that case, we hope the programmer will find the unifying perspective given in this paper useful, for its generality.

7 Conclusions

Writers of programs for dependable distributed real-time systems have to deal with programming abstractions which, for the purpose of tractability, hide the complex timing and ordering relationships between events and messages. The limits of these abstractions are sometimes ignored by the programmers. On the other hand, when considered without taking into account the requirements of the environment (physical or computational), they sometimes lead to too demanding, and thus inefficient, solutions.

The paper has equated the fundamental limits of time and order in distributed systems in a unified manner. It has extended previous results in this domain, giving precise formulations of the limits to temporal order in distributed systems with granular and imprecise clocks. Programs built with these limits in mind, will work correctly from the time and order viewpoints, be the abstractions they use time- or event-triggered.

These limits however, tend to compromise efficiency and concurrency of systems. With the help of innovative concepts such as δ_t*-precedence* and *execution granularity*, the paper has shown that the limits can be drastically relaxed, if the system support (clock granularity, order discrimination) can be matched to the ordering and timeliness requirements of programs. In consequence, system protocols will order events and messages more faithfully, though correctly. Until now, there has been a tendency to consider that the more fine-grained a temporal order or the closer the precision of a clock set, the better. The paper has shown that this is not so. Surprisingly, time-critical systems can be reliably controlled by coarse temporal order protocols and clocks with "lousy" precision, provided they still respect the limits imposed by the requirements of the processes they are controlling (cf. Rule 2 and Remark 5).

Finally, the paper discussed how applications can access time and order. Namely, it discussed what can a program deduct, from timestamps on events or messages, about their temporal order — and in consequence, causality — relations. By program, we mean a control application, or a system support protocol: we hope these results are as useful to real-time application programmers, as to system programmers, for example the ones building the system support for lock-step time-triggered systems, or for event-triggered acquisition and control systems. In appendix, we give a consolidated list of the Rules and Remarks made along this paper, together with an informal proof of the most relevant results.

Acknowledgements

The author wishes to thank Doug Seaton and Luís Rodrigues for the many discussions on causality and real-time. Many of the ideas expressed here were debated with Hermann Kopetz, whose previous work on the impact of clock granularity was of utmost importance. Finally, a word of appreciation to Michel Raynal and to the anonymous reviewers, who provided a number of useful comments on earlier versions.

List of Rules and Remarks

Rule 1: To ensure distributed timestamps of the same event are at most one tick apart, the granularity g_v of the global timebase must not be smaller than precision π: $g_v \geq \pi$.

Rule 2: Given any two potentially causally related events a and b deriving from program computations in a system[8] with execution granularity x_g, timestamped by a global clock of granularity g'_v and precision π, their temporal order is guaranteed to be asserted from the respective timestamps if $g'_v \leq x_g - \pi$.

Remark 1: Given two events a and b in different sites, timestamped by a global clock of granularity g_p, precision π, their temporal order can be guaranteed to be derived from the respective timestamps[9] only if $|t_b - t_a| \geq \pi + g_p$.

[8]For example, message `sends` and `receives`.

[9]If $|t_b - t_a| < \pi$, there may even be order inversion.

Remark 2: Given two events a and b in different sites, timestamped by a global clock of precision π, and granularity $g_v \geq \pi$, their temporal order can be guaranteed to be asserted from the respective timestamps only if $|t_b - t_a| \geq \pi + g_v$.

Remark 3: The discrimination of clock-driven protocols cannot be better than $\pi + g_p$.

Remark 4: Clock-driven protocols of granularity $g_v \geq \pi$ enforce $2g_v$-precedence, $\stackrel{2g_v}{\rightarrow}$.

Remark 5: The discrimination of temporal order protocols need not be better than execution granularity, x_g.

Remark 6: A lattice with a period of $g_l = x_g - \pi$, secures an $\stackrel{x_g}{\rightarrow}$ temporal order.

Informal proof of some results

Rule 1: The proof is obvious, if we think that, if clocks differ by $\pi \leq g_v$, the same tick interval in all clocks has at least an overlap of a single point in time (when $\pi = g_v$). That is all that is necessary to avoid that any timestamp of the same event is more than one tick apart. □

Rule 2: According to Remark 2, for order to be recognised, events must be separated by at least $\pi + g_v$. Substituting $g'_v = x_g - \pi$ for g_v, we have a new order condition: $|t_b - t_a| \geq x_g$. Since x_g is the execution granularity, no two events separated by less than it, in the same or different sites, are causally related, so the order can indeed be correctly relaxed and still reflect all interesting relations. □

Remark 1: a formal proof for continuous clocks was given by Lamport. For granular clocks, imagine first the clocks are in phase: for two events to have different timestamps, it is enough that they be g_p apart. Now imagine that one of the clocks starts drifting from the other, until reaching a difference of π: to keep the timestamp difference, the separation between events must also increase by π, to $\pi + g_p$. □

Remark 2: Derives from the proof of Remark 1 and the transformation of g_p in g_v. Refer to figure 3 to illustrate the limit situation. □

Remark 3: The proof is obvious, given g_p and π, the granularity and precision of the clocks serving the -protocol, and Remark 1. □

Remark 4: The tightest value that satisfies $g_v \geq \pi$ is $g_v = \pi$. Thence, the ordering condition in Remark 2 becomes $\pi + g_v = 2g_v$. □

Remark 5: Follows from Rule 2.

Remark 6: Follows from Remark 2 and Rule 2. Note that if $g_l = x_g - \pi$, then $g_l + \pi = x_g$, which according to Rule 2, is enough to guarantee order. □

References

[1] K. Birman and T. Joseph. Reliable Communication in the Presence of Failures. *ACM, Transactions on Computer Systems*, 5(1), February 1987.

[2] Kenneth Birman, Andre Schiper, and Pat Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems*, 9(3), August 1991.

[3] 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.

[4] F. Cristian, Aghili. H., R. Strong, and D. Dolev. Atomic Broadcast: From simple message diffusion to Byzantine Agreement. In *Digest of Papers, The 15th International Symposium on Fault-Tolerant Computing*, Ann Arbor-USA, June 1985. IEEE.

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

[6] Flaviu Cristian. Synchronous atomic broadcast for redundant broadcast channels. *The Journal of Real-Time Systems*, 2(1):195–212, 1990.

[7] Flaviu Cristian, Robert D. Dancey, and Jon Dehn. Fault-tolerance in the Advanced Automation System. In *Digest of Papers, The 20th International Symposium on Fault-Tolerant Computing*, Newcastle-UK, June 1990. IEEE.

[8] S. Hawking. *A Brief History of Time - from the Big Bang to Black Holes*. Gradiva, December 1988.

[9] H. Kopetz. Sparse Time versus Dense Time in Distributed Systems. In *Proceedings of the 12th International Conference on Distributed Computing Systems*, Yokohama, Tokyo, June 1992. IEEE.

[10] H. Kopetz, G. Grunsteidl, and J. Reisinger. Fault-tolerant Membership Service in a Synchronous Distributed Real-time System. In *Proceedings of the IFIP WG10.4 Int'l Working Conference on Dependable Computing for Critical Applications*, Sta Barbara - USA, August 1989.

[11] Hermann Kopetz, Andreas Damm, Christian Koza, Marco Mulazzani, Wolfgang Schwabl, Christoph Senft, and Ralph Zainlinger. Distributed Fault-Tolerant Real-Time Systems: The Mars Approach. *IEEE Micro*, pages 25–41, February 1989.

[12] Hermann Kopetz and K.H.(Kane) Kim. Temporal Uncertainties in Interactions among Real-time Objects. In *Proceedings of the Ninth Symposium on Reliable Distributed Systems*, pages 165–174, Huntsville, Alabama, October 1990. IEEE.

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

[14] Hermann Kopetz and Paulo Veríssimo. Real-time and Dependability Concepts. In S.J. Mullender, editor, *Distributed Systems, 2nd Edition*, ACM-Press. Addison-Wesley, 1993.

[15] Leslie Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *CACM*, 7(21), July 1978.

[16] F. Mattern. Time and Global States in Distributed Systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*. North-Holland, 1989.

[17] S.J. Mullender, editor. *Distributed Systems, 2nd Edition*. ACM-Press. Addison-Wesley, 1993.

[18] Larry L. Peterson, Nick C. Buchholdz, and Richard D. Schlichting. Preserving and Using Context Information in Interprocess Communication. *ACM Transactions on Computer Systems*, 7(3), August 1989.

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

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

[21] John Stankovic and Krithi Ramamritham. The Spring Kernel: A New Paradigm for Real-time Systems. *IEEE Software*, May 1991.

[22] P. Veríssimo and L. Rodrigues. Order and synchronism properties of reliable broadcast protocols. Technical Report RT/66-89, INESC, Lisboa, Portugal, December 1989.

[23] P. Veríssimo, J. Rufino, and L. Rodrigues. Enforcing real-time behaviour of LAN-based protocols. In *Proceedings of the 10th IFAC Workshop on Distributed Computer Control Systems*, Semmering, Austria, September 1991. IFAC.

[24] Paulo Veríssimo. Real-time Communication. In S.J. Mullender, editor, *Distributed Systems, 2nd Edition*, ACM-Press. Addison-Wesley, 1993.

[25] Paulo Veríssimo, P. Barrett, P. Bond, A. Hilborne, L. Rodrigues, and D. Seaton. The Extra Performance Architecture (XPA). In D. Powell, editor, *Delta-4 - A Generic Architecture for Dependable Distributed Computing*, ESPRIT Research Reports. Springer Verlag, November 1991.

[26] Paulo Veríssimo, L. Rodrigues, and J. Rufino. The Atomic Multicast protocol (AMp). In D. Powell, editor, *Delta-4 - A Generic Architecture for Dependable Distributed Computing*, ESPRIT Research Reports. Springer Verlag, November 1991.