Critical issues for the development of distributed real-time computing systems

Gerard Le Lann

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 1274

Programme 3
Réseaux et Systèmes Répartis

CRITICAL ISSUES FOR THE DEVELOPMENT OF DISTRIBUTED REAL-TIME COMPUTING SYSTEMS

Gérard LE LANN

Août 1990

QUESTIONS ESSENTIELLES POUR LE DEVELOPPEMENT DES SYSTEMES INFORMATIQUES REPARTIS TEMPS REEL

Gérard Le Lann
INRIA, Projet Reflecs
BP 105
78153 Le Chesnay Cedex, France

gll@score.inria.fr

RESUME

De nombreux problèmes sont posés par les systèmes informatiques répartis temps réel. Ceux qui sont jugés primordiaux font l'objet de cet article. A partir des définitions des termes "réparti" et "temps réel", on explore les concepts, les problèmes, les solutions correspondantes, qui sont directement liés à l'obtention des propriétés de répartition et de ponctualité.

L'informatique répartie temps réel est l'objet d'un intérêt croissant dans la communauté scientifique ainsi que dans le milieu commercial. Afin de permettre aux utilisateurs potentiels de comparer ce qui est réputé faisable et ce qui est actuellement commercialisé sous le nom de "système d'exploitation réparti temps réel", les points de vue biaisés ou erronés les plus répandus sont examinés.

Mots-clés : analyse d'hypothèse, langages, protocoles de communication, contrôle de concurrence, tolérance aux fautes, consensus, ordonnancement, ponctualité, temps réel, systèmes répartis, systèmes d'exploitation.

CRITICAL ISSUES FOR THE DEVELOPMENT OF DISTRIBUTED REAL-TIME COMPUTING SYSTEMS

ABSTRACT

Among the numerous issues involved with distributed real-time computing systems, those which are viewed as being central to designing such systems are presented. Based upon definitions of the terms "distributed" and "real-time", this article concentrates on concepts, problems and corresponding solutions that are directly related to the achievement of distributiveness and timeliness properties.

Distributed real-time computing is gaining increased interest in scientific circles as well as in the business oriented community. In order to help potential users to get a clearer picture of what is currently known to be achievable and what is currently marketed as "distributed real-time operating system", most common misconceptions and shortcomings are also reviewed.

Keywords : assumption analysis, languages, communication protocols, concurrency control, fault-tolerance, consensus, scheduling, timeliness, real-time, distributed systems, operating systems.

CONTENTS

1. INTRODUCTION

Thanks to quasi-continuous technological advances, the growth rate of computer based embedded systems has become one of the highest in the computing industry. Microelectronics and software are being inserted at the core of such conventional devices as washing machines, cars, toys, ovens and so on. Of course, embedded systems also play an increasingly vital role in the aerospace industry, in flight/transportation systems, in factory automation and so on. It is widely recognized that the trend in favor of embedded systems can only strengthen in the foreseeable future.

This fast expanding market has attracted and is attracting a great many vendors (computer manufacturers, software houses, system integrators, etc.) which deliver products and/or services indifferently quoted as being "embedded" or "on-line" or "transactional" or "automated" or "reactive" or "real-time". One particularly interesting example in the operating system arena is "real-time Unix", which we will examine further in this article.

The selection of a new/old term to refer to a particular class of computing systems is easy. The term "real-time" is the one selected in the framework of this article. What is more difficult, but necessary, is the elaboration of an acceptable rigorous definition.

Computer scientists have faced this kind of difficulty in the mid 70's with the notion of "distributed computing". Many religious wars have been fought before our community clearly acknowledged the need to discriminate between **physical dispersion** (as found in existing computer-communication networks) and **logical distribution of control** (as found in a few commercial offerings today). Pioneering work in this area was conducted between 1975 and 1977 at Bolt, Beranek and Newman [39], Massachusetts Computer Associates [18] and IRIA, now INRIA [21].

Although there is now a commonly agreed definition of "distribution" (see section 2.1), it is still the case that many systems which are currently marketed as being distributed include no instance of distributed control. There is no problem with this marketing ploy, except that users may learn the hard way that such systems enjoy none of the properties which come with truly distributed computing.

We are witnessing the same commercial strategies with real-time systems. For reasons quoted above, it is now more fashionable than before to be in the "real-time" business. However, the state of confusion is even higher than with distributed systems because it is only recently that the research community has shown signs of interest, on a significant scale, for real-time computing. The elaboration of a widely accepted definition of real-time computing still is an open subject. Nevertheless, a definition of the concept of real-time which meets our requirements and seems to be accepted by many other researchers is given in section 2.1.

Trying to shed some light on most important concepts in the area is the major objective of this article. There is definitely a need to clarify the issues involved [38]. Other objectives pursued with this article are as follows :

- to investigate the basic nature of distributed real-time computing
- to review and dispel the most common misconceptions about distributed real-time computing
- to identify the most critical issues in the area
- to quote relevant work in the area.

2. CRITICAL ISSUES AND MISCONCEPTIONS

2.1. Definitions

We define the terms "distributed" and "real-time" as follows.

Distributed system

A computing system whose behaviour is determined by algorithms explicitly designed to work with multiple loci of control.

Examples of such algorithms are those used to control concurrent threads of computation (e.g. concurrency control). We will say that a system is entrusted with distributiveness properties to mean that its design is in accordance with the definition given above.

Real-time system

A computing system where initiation and termination of activities must meet specified timing constraints. Time-dependent values are associated with activity terminations. System behaviour is determined by algorithms designed to maximize a global time-dependent value function.

A system designed in accordance with this definition is said to be entrusted with timeliness properties.

2.2. Basic facts

A distributed real-time system must behave as prescribed by its specifications, i.e. it must be **predictable** in the **logical** domain (correctness properties) and in the **time** domain (timeliness properties).

Correctness requirements (e.g. safeness, liveness) are stated via time-independent expressions or invariants, e.g. consistency constraints for a data structure. Timeliness requirements are stated via time-dependent expressions, e.g. maximum system response time for a given set of inputs.

The design and implementation of correct and timely distributed real-time systems is rendered difficult by the existence of the following **physical facts** :

- multiple asynchronous hardware elements
- occurence of faults
- finite space, speed and energy levels
- passing of time.

As a consequence (but it is only a consequence of those basic facts), in the general case, delays for computing, for communicating, cannot be known or predicted with certainty. This is a major impediment to the realization of distributed real-time systems. Furthermore, in general, system loads are variable and cannot be fully anticipated, this being particularly true with distributed systems. Consequently, all problems derive from the following basic dilemma : how to build a **predictable** system when (i) the system environment (ii) the system constituents, exhibit non fully predictable behaviours ?

Quite obviously, under the most general assumptions, there is (demonstrably) **no** solution to this problem. Hence the importance of the first critical issue, referred to as the assumption analysis.

2.3. Assumption analysis

The goal pursued with an assumption analysis is to estimate the level of predictability of a given system. An assumption analysis consists in the following :

- to state the assumptions (internal fault patterns, hardware speed, arrival laws, resource conflicts, external "aggressions", multiplexing ratio, etc.) either probabilistically or deterministically

- to express the coverage factor of each assumption (probability that an assumption is not violated at run time), in a way similar to coverage factors used for fault-tolerant systems [33].

Based on such an analysis, intrinsic performance indicators are expressed as probability distributions which show how likely it is that system performance lies within some given bounds. Consequently, merits of various solutions (architectures and algorithms) can be rigorously computed and compared.

It is clear that the **predictability** of any given solution can only be expressed as a probability.

In particular, one should not expect to do away with the basic dilemma stated above by resorting to what is commonly called a "worst case" analysis. There is no such thing as an absolute "worst case" analysis for it is always possible to imagine something worse than any stated "worst case". An ultimate worst case consists, for example, in having all system constituents down. We must admit that, with live and usable systems, we are indeed dealing with probabilities of certain "bad cases" to occur.

It is appropriate at this point to address the following common misconception : synchronous/deterministic designs and asynchronous/probabilistic designs are antagonistic. In fact, each approach delimits an end-point of a continuous spectrum of assumption sets. One widely publicized advantage of a synchronous/deterministic approach is that it is amenable to formal proving and/or to mechanical verification. This is certainly true with not too complex assumption sets, less true when assumptions get elaborate. Unfortunately, except for the simplest systems, assumption coverage factors are smaller with the former case than with the latter.

It is thus not clear at all that a synchronous/deterministic approach always yields the highest predictability level. Many asynchronous/probabilistic designs also are mechanically verifiable. For reasonably complex systems, it is highly likely that a "good" approach lies somewhere within the spectrum, never at the end points.

2.4. Languages, proofs and verification

This second critical issue is somewhat related to the first one. Indeed, we currently have no entirely satisfactory solution at hand. As far as languages are concerned, many of those constructs that are needed to obtain correctness properties (see sections 2.6, 2.7 and 2.8) cannot be expressed with such languages as Ada, Occam or Concurrent C. This is not meant to dismiss the importance of such languages. They represent steps in the right direction. However, we must be aware of their limitations for distributed real-time systems. To give just one example, it is obvious that deadlocks can occur in multi-client/multi-server systems programmed in Ada.

Conversely, many languages allow for the expression of those timing constraints needed to enforce timeliness properties (see section 2.9). These languages are general languages with a time dimension "added", or specific asynchronous languages (e.g. Jovial, LTR, Ada) or specific synchronous languages (e.g. Lustre, Esterel). However, the major difficulty does not lie in expressing the individual timing requirements (one set of requirements per process) but rather in proving or verifying that, for a given set of hardware elements and algorithmic solutions, prescribed **system-wide** timeliness properties are guaranteed for given process sets and for given arrival laws.

The complexity of establishing proofs or running thorough verifications strongly depends on the outcome of the assumption analysis. Comments made under section 2.3 for synchronous/deterministic versus asynchronous/probabilistic approaches fully apply to languages, proofs and verifications.

As far as proofs are concerned, contrary to expectations, temporal logic has not proven yet to be the ultimate formalism that some of us thought it would be. With respect to verification, most techniques currently in use are based on the exploration of state spaces (e.g. timed Petri nets). The well known problem encountered is that of state explosion even when assumptions are reasonably restrictive.

State reduction techniques are used to ease the problem in the case of non real-time systems. It remains to be seen how satisfactory such techniques are with reasonably complex time constrained systems. In other words, one needs to carefully assess how much is lost with state reduction techniques in abstracting away from the implementation and the environment (yielding smaller assumption coverage factors).

As a final comment, let us observe that the problem of developing software for distributed **and** real-time systems is not very well addressed with existing languages for they are either biaised towards distribution (e.g. Concurrent C) and unable to allow for the expression of timing constraints or biaised towards timeliness (e.g. Esterel) but unable to allow for the expression of general parallel constructs.

2.5. Communication

This third critical issue probably is one of the best understood for non time contrained systems. Over the last 15 years, many communication protocols have been standardized by ISO/OSI Technical Committees whose primary (almost unique) goal is to develop standards to allow for **interworking in heterogeneous environments**.

One major misconception, which can be illustrated with statements found in documents produced by some of the MAP working groups (MAP is intended to be the standard for factory local area networks), is that existing ISO/OSI standards can be viewed as appropriate solutions for such real-time systems as automated factories. This is somewhat surprising for it is obvious that heterogeneity hidding might be necessary but is certainly not sufficient to build a real-time system.

The ability of existing ISO/OSI standards to solve real-time communication problems should be seriously questioned. A number of undertakings show that the problem has not gone unnoticed to such bodies as SAE in the USA and DEI in France, which contribute to the elaboration of NATO standards for real-time communication systems (Stanags), e.g. LTPB (SAE) and GAM-T-103 (DEI), as well as IEC (Fieldbus) and ANSI (XTP). It can be speculated that some of these real-time communication protocols will eventually turn into ISO/OSI standards.

From a strictly technical viewpoint, a number of interesting results have been established in the area. In comparison with current ISO/OSI standards, these results constitute significant advances in terms of timeliness and distributiveness properties. Let us elaborate on two sub-areas, namely multiaccess protocols and end-to-end protocols.

Multiaccess

For both slotted or unslotted communication channels, one basic question to be answered is whether self-adaptive protocols (e.g. ISO/OSI 8802/3) are better or worse than static protocols (e.g. static TDMA). The controversy between contention-based protocols and contention-free protocols is a truncated exposure of the more general question stated above. Correctness of the answer depends on the types of assumption sets considered.

At one extreme end, one finds fully deterministic assumption sets (e.g. periodical arrivals, no fault, no system modification). Tailor-made static TDMA or some token-passing protocols (e.g. ISO/OSI 8802/4, FDDI) are appropriate if the assumption coverage factor is close to 1. This of course not always the case in the real world.

At the other extreme end, one finds probabilistic assumption sets (e.g. bursty arrivals, random fault patterns, mobile contenders). Only those protocols which include some form of contention are acceptable in that case. Contrary to token-passing, contention-based protocols enjoy a property known as channel transparency. Channel transparency means that average access delays are O(1) for all sets of arrival laws such that the channel load does not exceed some unstability threshold. Translated into the notions used in this paper, this means that contention-based protocols yield average access delays, which are O(1) for assumption sets which are much larger than those matched by token-passing protocols.

A class of contention protocols which is of particular interest is that of tree protocols. They have been thoroughfully analyzed [11, 12]. Contrary to token-passing, exact analytical models have been established for tree protocols. They exhibit excellent stability thresholds and their deterministic variants outperform token-passing protocols for a vast majority of assumption sets corresponding to realistic situations.

End-to-end

Existing ISO/OSI standards have not been designed to handle time constrained message-passing rigorously.

For example, existing ISO/OSI standards (connectionless or connection-oriented) cannot provably handle periodical arrivals and achieve some a priori given quality of service. Sampled connections in GAM-T-103 [28] provide a solution to this problem.

With existing ISO/OSI standards, fault tolerance is obtained exclusively via error detection and recovery. This is not necessarily the best solution for real-time systems. Masking is the other approach, which has been more recently explored. Similarly, rate control, as used in XTP [31], is a flow control mechanism which is often more appropriate for real-time systems than credit-based mechanisms.

Finally, conversation constructs involving more than 2 members (2 is the rule with most existing ISO/OSI standards) have been defined for real-time systems in GAM-T-103. The general problem of reliable broadcast, not currently taken into account by ISO/OSI working groups, has been very much investigated and solved over the last 7 years by the research community (see section 2.8).

2.6. Concurrency control

This fourth critical issue is also fairly well understood for non real-time systems.

The first area of Computer Science to directly address the problems of concurrency control in distributed systems was that of distributed databases [3]. The issue of maintaining the "consistency" of a set of distributed data structures that can be "simultaneously" created, destroyed, read, updated by a possibly unknown number of processes has been at the origin of fundamental results and concepts such as that of serializability [9, 30] and atomic transaction [9, 15].

Most decentralized synchronization algorithms known today are refinements of those solutions that were devised to enforce the atomicity property for concurrently executing distributed transactions. Examples of these early algorithms are two-phase locking [9], timestamp ordering [35] and ticket ordering [22]. Atomic transactions with replicated databases were shown to be feasible with quorum protocols [14, 39]. Since then, many solutions to the concurrency control problem have been published. In [4], existing concurrency control algorithms are categorized in three classes, namely locking protocols, non-locking schemes and multiversion concurrency control. Many of these algorithms resort to blocking or rolling back transactions in case the desired consistency properties would be endangered. Although this is acceptable with non real-time systems, one might have to look for different solutions under the following constraints which are typical of real-time systems :

- wait states or roll-backs are not allowed for they are antagonistic with timing constraints

- intermediate outputs produced while executing transactions must be triggered in time, i.e. made visible, and cannot be cancelled later on (in case of a roll-back).

Several teams are currently working on the concept of real-time transactions. An example of the issues involved is that of timely atomicity and how algorithms used for fault-tolerance, concurrency control and scheduling relate to each other [23].

There is one major common misconception with concurrency control in real-time systems, that is as follows : concurrency control is not needed, for real-time systems should be structured in such a way that interprocess conflicts never occur ; should conflicts occur, system behaviour would be unpredictable.

Designs derived from this type of biaised perspective rest on such assumptions as preclaimed resources, deterministic arrivals, no fault occurence, static system configurations and so on. The level of intellectual effort induced by such designs is close to zero, which is good news. The bad news are that, for many real systems, the coverage factor of such assumptions also is close to zero.

2.7. Fault-tolerance

This fifth critical issue is of particular relevance with distributed systems for fault-tolerance has to do with comparing the actual behaviour (of a system, of a constituent) with some intended behaviour. This implies that the observee and the observer are distinct entities, with independent fault modes if possible. Existence of multiple entities precisely is a pre-requisite for distributed systems. This issue is also of direct concern to designers of real-time systems. Indeed, the best way to violate timeliness requirements consists in paying no attention to occurence of faults. Conversely, time is the only means whereby one can tell whether an entity is faulty or whether it behaves correctly but slowly.

State-of-the-art in the area of fault-tolerance is presented in [20]. Here, we briefly review those notions and models useful to reason about fault tolerance with distributed real-time systems. Fault-tolerance is obtained out of redundant (hardware, software, data) entities which cooperate via decentralized algorithms. A correct entity produces **correctly valued** outputs **in time**, in response to inputs.

In the value domain, possible failures are as follows :

- undefined value, i.e. detectable
- defined value (i.e. considered correct) but incorrect vis-a-vis the related input.

In the time domain, possible failures are as follows [8] :

- crash failures (eternal silence)
- omission failures (transient losses)
- early/late timing failures.

When entities interact with each other, it is necessary to distinguish between two failure mode assumptions, namely :

- consistent behaviour, whereby a failed entity behaves indentically for all other entities,
- arbitrary behaviour, whereby a failed entity exhibits different behaviours to different entities.

This latter case was thoroughfully examined first in [19].

All of the above applies to software fault tolerance as well as to hardware fault tolerance. Passive redundancy (recovery blocks or stand-by-sparing) fits well with sequential execution models

whereas active redundancy (N-version programming or N-modular redundancy) fits better with parallel execution models.

Under specific assumptions (e.g. independent fault modes), it has been shown that a system can tolerate up to f simultaneously erroneous entities if it comprises at least :

- $f+1$ entities, to conduct error detection (assuming no arbitrary behaviour)
- $2f+1$ entities (assuming consistent behaviour) or $3f+1$ entities (assuming arbitrary behaviour), to keep functioning correctly.

Achieving fault-tolerance in a distributed system subsumes the possibility for multiple entities to reach agreement (e.g. whether to passivate/activate a replicate). Reliance on one single entity would be in contradiction with the general principles of distributed computing.

This leads us to focus on an important sub-area of fault-tolerance, that of reliable distributed consensus.

2.8. Consensus

This sixth critical issue has received much attention recently. It is of paramount importance to distributed systems and to real-time systems.

A primitive form of consensus is that of reliable broadcast, a facility that enables a process to send a given message to a set of processes. The simplest property of a broadcast algorithm is atomicity, that is, a message is received by all destinations that do not fail or by none of them. At first glance, this seems to be a trivial problem. It is not, even under simple fault assumptions. "Obvious" solutions either do not work or have communication costs that grow exponentially with the number of processes involved [1]. The "reliable commit" algorithms known in the distributed database area are examples of "solutions" to this problem (see further).

Assuming crash failures only, even more complex algorithms have been devised that ensure ordered broadcasts, that is, different broadcasts are delivered to different destinations with the guarantee that messages are delivered in the same order everywhere [6]. A great deal of work has also been conducted to devise reliable broadcast algorithms that could cope with consistent or arbitrary failures. Two classes of solutions have been investigated, namely deterministic solutions [7, 19] and randomized or probabilistic solutions [2]. Conversely, few results have been published so far for the problem of ordered broadcasts in the presence of omission failures [32].

A more elaborate form of consensus is raised by the need to get distributed processes agreeing on some unique decision, (e.g. the state of a given processor, should a reconfiguration be initiated, current time value) in spite of the existence of different or contradictory initial views. Reliable broadcasts help but do not suffice.

For reasons that will become obvious later on, it is appropriate to investigate the relationships that may exist in real-time systems between timeliness constraints, timing errors and reliable broadcast or consensus.

With real-time systems, algorithms for reliable broadcasts should guarantee that all deliveries take place within a prescribed time window. Similarly, real-time reliable consensus protocols should guarantee that those times at which each non-faulty process discovers that consensus has been reached lie within some prescribed time window.

All known deterministic solutions to these problems take into account worst-case behaviours for some given assumption set. For example, it is assumed that one knows timing bounds to schedule a process, to transmit a message and so on, so that a bound Δ can be computed for any end-to-end value exchange under given fault assumptions [7].

A value broadcast at time t (timestamped with t) is released at every non-faulty destination at time $t+\Delta$ (even though it might have been received some time earlier than $t+\Delta$). Clearly, synchronized physical clocks are needed to implement such protocols. Clock synchronization [17, 25, 26, 27, 37] is one important instance of a reliable consensus protocol (see further). Consequently, in real-time systems, it is the case that reliable consensus is a basic (low level) construct. Such a construct is needed to establish some degree of synchronicity, i.e. a global time reference, which can then be used to build such other constructs as time bounded reliable broadcasts.

Interestingly enough, it is also the case that a time bounded reliable broadcast is sometimes assumed to exist in order to achieve reliable clock synchronization. Quite clearly, unless assumption sets are rigorously stated, there is a potential danger of being trapped in "recursive" solutions or tautologies (solutions are "hidden" in the assumptions). Unfortunately, this turns out to be the case in some published papers.

More generally, the need for relying on some form of synchronicity to achieve deterministic reliable consensus has been established a few years ago. It has been shown that distributed consensus cannot be reached in a finite number of steps in an asynchronous system subject to failures [10]. However, distributed consensus can be obtained in asynchronous systems with probabilistic protocols that may never terminate, but this would occur with probability 0 [5].

For the particularly important problem of clock synchronization, the quality of the consensus (precision and accuracy) depends not only on fault patterns but also on how accurately message transmission delays are estimated. This is due to the fact that values transported (clock values) are time dependent. All solutions published so far are either deterministic (lower and upper bounds to schedule a process and transmit a message are assumed to be known) or probabilistic (no upper bound is assumed to be known). Only recently has work been reported on statistical solutions.

Whatever approach is chosen, again, one has to carefully assess assumption coverage factors. For example, under a deterministic approach, one might have to explicitly enforce the assumption that upper bounds cannot be violated. The corresponding coverage factor should be estimated. Similarly, under a probabilistic or a statistical approach, it does not suffice to compute the performance of an algorithm under those conditions reflected in an assumption set (e.g. system load is always average or null when needed). It is also important to assess the assumption coverage factor and compute performance under adverse conditions (e.g. in order to show that a given algorithm converges for any value of system load).

The major misconception is this area is that it suffices to provide processes with access to some centralized clock to solve the problem of establishing a consistent distributed time reference.

Even under the assumption that the central clock is fully reliable, all the problems due to transmission delay uncertainty, types of faults and density of faults to be tolerated need to be addressed. This is precisely what is done with algorithms referenced in the above.

2.9. Scheduling

This seventh critical issue is directly related to the achievement of timeliness properties.

It has been amply demonstrated that scheduling problems are NP-complete in the general case, even for a uniprocessor (see [13] and subsequent work). Therefore, known usable scheduling algorithms are based on probabilistic schemes or heuristics [16, 29, 34].

Under the particular assumptions of deterministic arrivals (e.g. periodical arrivals), absence of resource conflicts (besides the CPU resource) and absence of faults, exact analytical results have been established for scheduling algorithms based on fixed priorities and preemption [24]. These results have been generalized (e.g. [36]).

It has also been amply demonstrated that fixed priorities should not be used to schedule time-dependent processes under more general assumptions (e.g. asynchronous arrivals). We get back to this issue in section 2.10.

Let us now review two very common misconceptions.

Real-time computing is fast computing

Let us consider two computing systems, one called the Tortoise and the other called the Hare*. The Tortoise has the following features :

- speed : 1
- task scheduling and context-switch latency : 1
- scheduling policy : earliest deadline first, no preemption.

The Hare has the following features :

- speed : 10
- task scheduling and context-switch latency : 0
- scheduling policy : first come-first served, no preemption.

Gain ratios of the Hare over the Tortoise therefore are :

- 10 in terms of raw processing power
- ∞ in terms of task scheduling/switching latency.

To some people, the Hare is obviously "more" real-time than the Tortoise.

Let us now consider an application comprising two tasks with the following situation, typical of real operational conditions. At time t, task A is pending, about to be scheduled and run. However, also at time t, a request for activating task B is triggered. Tasks A and B are equally critical tasks.

Let us assume that task attributes are as follows :

Task A

- duration : 270, at speed 1
- deadline : t + 320

Task B

- duration : 15, at speed 1
- deadline : t + 21

Execution patterns would be as shown figure 1, for the two systems considered.

Conclusion is obvious. The Tortoise, which is slow but which processes tasks in correct sequence, meets both deadlines. This is a safe real-time system for the application considered. The Hare, which runs tasks very fast, but in the wrong order, does not meet both deadlines. The Hare is definitely unsafe for the application considered.

Real-time computing is not equivalent to fast computing. Quite clearly, it is possible to build counter-examples, showing that the Hare can win against the Tortoise. However, such examples would do no more than demonstrating that with a brute force approach (over-dimensioned systems), one can do as well as with a clever approach and rightly dimensioned systems.

* after the tale "The Hare and the Tortoise", from the French writer Jean de la Fontaine (1621-1695)

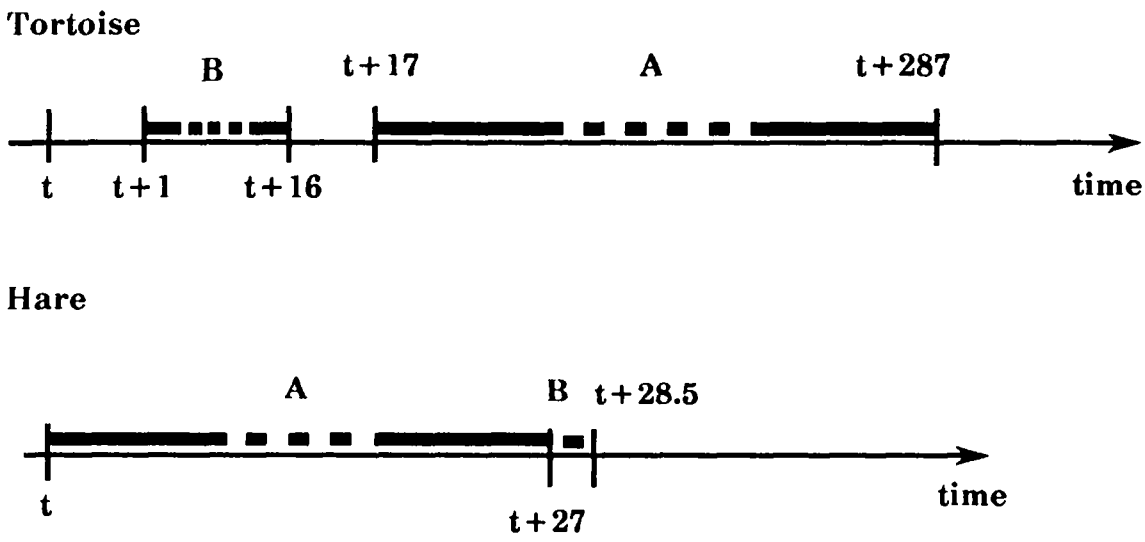

Figure 1 : Execution patterns

System over-dimensioning is not a solution but rather demonstrates refusal to look for a solution. Futhermore, there are cases where over-dimensioning is antagonistic with weight and energy requirements (e.g. space applications).

Fast computing might help but does not suffice. Appropriate scheduling algorithms are necessary.

Real-time computing is fast priority interrupt handling and fast context switch

In the example given above, the Hare system was assumed to have a zero time context switch. And it was shown to lose against the Tortoise for the application considered. What if preemption would be allowed ?

Many commercial systems are coined "real-time" because they can handle hardware generated interrupts and do task context switch in, say, 2 microseconds rather than, say, 50. Let us call this delay the preemption latency.

To see how trifling this type of argument may be, it suffices to compute the savings induced with small preemption latencies.

For any "reasonably well" constructed system, s, the average preemption latency, and D, the average task duration, are such that $s/D << 1$.

The savings achieved by an ideal system ($s = 0$) compared to a non ideal system would then be measured by the ratio $(D + ns)/D$, if at most n consecutive preemptions can occur.

Firstly, let us observe that any system running at a speed $1 + ns/D$ times higher than the ideal system would perform as well in average. For example, with $D = 50$ ms and $n = 10$, a system where $s = 50$ microseconds and running 1% faster than the ideal system would perform as well. And, by the way, is it not the case that raw MIPS are cheaper than specialized hardware ?

Secondly, the ultra fast preemption argument misses the real issue again.

Indeed, in order to obtain impressively small preemption latencies, scheduling algorithms implemented within such "real-time" systems usually are the dumbest ones (yielding fast scheduling decisions).

For example, scheduling schemes used in most current commercial systems make use of fixed priorities such as hardware interrupts. There is a major problem with these schemes (see section 2.10). Basically, fixed priorities are alien to the notion of time. Those time dependent attributes which are associated with tasks in real-time systems are simply ignored.

Therefore, it does not matter so much whether wrong scheduling decisions or, at best, poor scheduling decisions, are made quickly or not. What matters is that such fast preemption based systems do not make the right decisions. Using the example given above again, one obviously sees that if B is (a priori) attributed a priority level inferior to A's level, preemption does not help.

Furthermore, the ultra fast preemption argument demonstrates a confusion of ends and means, in that it is based on the belief that starting a process as fast as possible is equivalent to guaranteeing timely completion of that process and of other processes it is contending with.

Again, the identification of a time dependent priority based scheduling algorithm is the key issue. Raw hardware performance or nominal speed of a particular hardware primitive is of secondary importance.

2.10. Overall design

2.10.1. Principles

One basic question, if not **the** basic question for designers of distributed real-time systems is as follows : for a given set of assumptions, what is, for every design problem (and solution), the appropriate level of complexity to be sought ?

For the sake of simplicity, let us assume that designers have to choose between "simple" and "complex" solutions.

Common pros and cons are as follows.

Simple solutions

Pros : They are easy to understand, to implement. They are amenable to mechanical verification. Overhead incurred is limited. They are "robust".

Cons : System behaviour is unpredictable (i.e. solutions may just do not work) in case some of the assumptions are violated at run time. When the original assumption set is modified (e.g. some of the arrival laws are changed), mechanical verification must be re-run.

Complex solutions

Pros : Solutions keep working for run time situations which match a superset of the original assumption set. For any given assumption set, if a simple solution is mechanically verifiable, then mechanical verification of a general solution is also feasible.

Cons : They are difficult to understand, to implement. Overhead incurred is significant. This overhead could better be used by a specific and simple solution so as to either provide better performance or cope with richer assumption sets.

Before discussing these statements, let us give two examples of such arguments. In the area of real-time multiaccess protocols, token-passing schemes are sometimes believed to be better suited than contention schemes. They are believed to be simpler and "deterministic". In the area of time constrained process scheduling, fixed priority based algorithms (e.g. integers) are sometimes

believed to be well adapted. They look simple compared with time-dependent priority based algorithms (e.g. least laxity first) which are believed to be unnecessarily sophisticated and expensive.

Human nature is such that simple, easy to understand solutions are spontaneously favoured (and selected). However, with distributed real-time applications, the question is whether such an instinctive attitude is appropriate when it is known that the selection of a wrong or "weak"solution might entail loss of human lives or/and huge financial penalties. There are many examples of computerized/automated time critical systems which have caused significant damages to people, to ecosystems, simply because their specifications had been left incomplete (assumption sets were too restrictive and system behavior was unpredictable under the occurrence of "unlikely" events) or because (too) simple solutions had been retained.

Most people responsible for proper operation of critical applications based on distributed real-time systems would certainly feel more comfortable if they could be shown that, for some reasonable overhead, the solutions selected have the potential of matching situations corresponding to a superset of the original assumption set, i.e. system behavior is correct even under some situations that were not anticipated.

What kind of overhead are we considering, after all ? Hardware resources ? Processing power, channel bandwidth have become the cheapest resources used in computing systems. Why save on them ? Manpower ? This takes us back to the issue of complexity/simplicity. It is not at all clear that "simple" solutions perform better than complex solutions.

Let us use the same two examples to illustrate this.

At first glance, token-passing seems simple. Let us concentrate on token passing busses (e.g. ISO/OSI 8802/4). Contenders are polled in sequence. That sequence and maximum token sojourn times are derived from the assumption set (arrival laws for every contender). Apparently, no conflict or collision can ever happen. Wrong if token losses can occur. Wrong if contenders become faulty or may fail. Wrong if contenders are repaired. Thus the need to supplement an initially simple scheme (devised to avoid conflicts and collisions) with extra protocols meant to cope with conflicts and collisions !

This strange intellectual recursion has an analogy in Mathematics : demonstration of a theorem which violates one of the original axioms.

Indeed, the final result is a protocol standard (ISO/OSI 8802/4) which is definitely much more cumbersome than the standardized contention based protocol (ISO/OSI 8802/3) derived from Ethernet.

The Fieldbus profile was initially based on a similar "good simple" idea : centralized polling is the best way to go for cheap time constrained multiaccess control. However, more recent proposals include those extra protocols needed to detect and recover from a faulty or failed bus master, thus yielding higher complexity and higher costs.

As indicated under section 2.5, deterministic tree protocols which may look sophisticated at first sight are simpler in fact, they perform better under similar assumption sets and remain predictable under assumption sets larger than those matched by token-passing protocols.

Similarly, at first glance, fixed priority based scheduling seems simple and easy to use. The intuitive idea is that it suffices to assign a priority level (an integer) to every process (called priority mapping) prior to letting a system run. At run time, the scheduler always select the pending process with the highest priority level.

Of course, in real-time systems, processes have timing constraints attached to them (e.g. deadlines, frequencies, time-value functions). As indicated before, for general assumption sets, even for a uniprocessor system, corresponding scheduling problems are known to be NP-complete.

Then the questions : How can time dependent constraints be rigorously translated into time independent integers ? Is it not the case that the priority mapping problem is NP-complete as well ? How then can it be suggested that the priority mapping problem can be easily "solved" by users (of such primitive systems) ? So much for the "simplicity". Is it not the case that fixed priority based scheduling yields possible starvation and probabilistic service for all processes except those mapped onto the highest priority level ? How then can this be considered as a sufficient solution towards the construction of deterministic systems ? So much for the "efficiency".

As indicated under section 2.9, probabilistic algorithms or heuristics based on time dependent attributes are simpler to use (no need to perform fixed priority mapping), they have been shown to perform better under similar assumption sets and remain predictable under assumption sets larger than those matched by fixed priority based scheduling algorithms.

2.10.2. Where do we stand ?

Operational real-time systems have a rather long lifetime. Systems in use today have been designed years ago. They are not distributed. They are proprietary.

The trend in favour of distributiveness and portability (or openness) is beginning to make inroads within the real-time computing arena, which is not surprising (see Introduction). This trend can be witnessed with the mushroom growth of advertisements for "distributed real-time Unixes".

Let us be clear : Unix, which has been designed more than 15 years ago, was not meant to be a distributed real-time operating system. Therefore, any "distributed Unix" or any "real-time Unix" must include significant extensions to the original Unix, which raises interesting technical and standardization issues.

Technical issues

What is the nature of these extensions ? For the vast majority of announced "distributed Unixes", distribution is restricted to mean physical dispersion. At best, processes can interact through client-server patterns. Exceptionally, data location transparency is provided. But there is currently no "distributed Unix" which includes algorithms which would support system-wide concurrency control, transaction processing, fault-tolerance or consensus. Similar comments apply to the Mach operating system which has been selected as its basic technology by the Open Software Foundation.

Similarly, there is currently no "real-time Unix" which would include a time dependent priority based scheduler and those appropriate fault-tolerant algorithms required to handle those arbitrary degrees of redundancy needed to do timely error masking or/and timely error detection and recovery, even for the simplest type of failures (crash).

In most cases, some existing Unixes are called "real-time" because they do interrupt handling, they manipulate fixed priorities and they support multiprocessing ! How can this be taken seriously ? IBM OS 360 did just that 25 years ago, and was not considered to be a real-time operating system.

The usual argumentation of promoters of such so-called distributed real-time Unixes consists in stating that these (missing) "high-level" capabilities ought to be provided by server processes, whose cooperation is obtained via specific "protocols". Although it is interesting to know that current operating systems are architectured after a kernel + servers model, such arguments miss the real issue again : these specific "protocols" precisely are those algorithms needed to entrust operating systems with distributiveness and timeliness properties. And they are not currently implemented in commercially available "distributed real-time Unixes".

Standardization issues

The marketing strategies behind the improper use of the terms "distributed" and "real-time" are quite obvious. With Unix, we are witnessing commercial tricks similar to those used in the area of communication protocols : anything that is standardized or portable (open) is good for all kinds of applications !

In the context of communication protocols (section 2.5), it has been pointed out to the fact that heterogeneity hidding has nothing to do with real-time.

The same observation holds with operating systems. Unix, OSF offerings and others still have some way to go before they turn into international standards. They will be (good) standards for message-based time-sharing operating systems.

Full stop. Unless major changes would occur in the foreseeable future, such standards will have nothing to do with timeliness, nothing to do with distributiveness. Such standards could then be challenged by commercial offerings derived from research work specifically targetted at timeliness and distributiveness issues.

3. CONCLUSION

There are many interesting challenges lying ahead for designers of distributed real-time systems. These challenges must be overcome before distributed real-time computing can become a real scientific discipline rather than just ad-hoc engineering. Fortunately, quite a significant number of solutions have been devised by the research community, which make it possible to put **quantifiably** trustable distributed real-time systems into operation.

Emerging concepts, formalisms, models and solutions which show the way for building such systems have been sketched out in this article through the investigation of most critical issues as well as the presentation of major misconceptions in the area.

Users of distributed real-time computing systems should not be forced to get involved into such issues. System experts bear full responsibility for developing and bringing to market those technical solutions that do meet real distributiveness and timeliness constraints, rather than those solutions that do not take too much intellectual and/or financial effort.

ACKNOWLEDGEMENTS

This article is based on a presentation given at the European Space Agency Workshop on "Communication Networks and Distributed Operating Systems within the Space Environment", October 24-26, 1989, Noordwijk, The Netherlands. Comments made by Flaviu Cristian, IBM Almaden, on a previous version of this article are gratefully acknowledged.

REFERENCES

[1] O. Babaoglu, R. Drumond, "Streets of Byzantium : Network architectures for fast reliable broadcast", IEEE Trans. on Software Engineering, SE-11(6), June 1985, 546-554.

[2] M. Ben-Or, "Another advantage of free choice : completely asynchronous agreement protocol", 2nd Symp. on the Principles of Distributed Systems, 1983, 27-30.

[3] P.A. Bernstein, N. Goodman, "Concurrency control in distributed database systems", ACM Computing Surveys, 13 (2), June 1981, 185-221.

[4] P.A. Bernstein, V. Hadzilacos, N. Goodman, "Concurrency control and recovery in database systems", Addison-Wesley Pub., ISBN 0-201-10715-5, 1987, 370 p.

[5] G. Bracha, S. Toueg, "Asynchronous consensus and broadcast protocols", J. ACM, 32 (4), October 1985, 824-840.

[6] Jo-Mei Chang, N.F. Maxemchuck, "Reliable broadcast protocols", ACM Trans.on Computer Systems, 2 (3), August 1984, 251-273.

[7] F. Cristian et al., "Atomic broadcast : from simple message diffusion to Byzantine agreement", 15th IEEE Symp. on Fault-Tolerant Computing", Ann Arbor, June 1985, 200-206.

[8] F. Cristian, "Understanding fault-tolerant distributed systems", IBM Research Report RJ 6980, April 1990, 30 p.

[9] K.P. Eswaran et al., "The notions of consistency and predicate locks in a database system", Com. ACM, 19 (11), November 1976, 624-633.

[10] M.J. Fischer, N.A. Lynch, M.S. Paterson, "Impossibility of distributed consensus with one faulty process", J. ACM, 32 (2), April 1985, 374-382.

[11] P. Flajolet, P. Jacquet, "Analytical model for tree communication protocols", NATO Advanced Study Institute on Flow Control of Congested Networks, Capri, Springer-Verlag, 1987.

[12] R.G. Gallager, "A perspective on multiaccess channels", IEEE Trans. on Information Theory, IT 31, 1985, 124-142.

[13] M.R. Garey, D.S. Johnson, "Complexity results for multiprocessor scheduling under resource constraints", SIAM Journal of Computing, 4, 1975, 397-411.

[14] D.K. Gifford, "Weighted voting for replicated data", 7th ACM SIGOPS Symp. on Operating Systems Principles, Pacific Grove, December 1979, 150-159.

[15] J.N. Gray, "Notes on database operating systems", in "Operating Systems : An Advanced Course", Lecture Notes in Computer Science n° 60, Springer-Verlag Pub., 1978, 393-481.

[16] E.D. Jensen, C.D. Locke, H. Tokuda, "A time-driven scheduling model for real-time operating systems", IEEE Real-Time Systems Symp., 1985, 112-122.

[17] H. Kopetz, W. Ochsenreiter, "Clock synchronization in distributed systems", IEEE Trans. on Computer, 36 (8), August 1987, 933-940.

[18] L. Lamport, "Time, clocks and the ordering of events in a distributed system", Com. ACM, 21 (7), July 1978, 558-565.

[19] L. Lamport, R. Shostak, M. Pease, "The Byzantine Generals problem", ACM Trans. on Programming Languages and Systems, Vol. 4, 1982, 382-401.

[20] J.C. Laprie, "Dependable computing and fault-tolerance : concepts and terminology", 15th IEEE Symp. on Fault-Tolerant Computing, Ann Arbor, June 1985, 2-11.

[21] G. Le Lann, "Distributed systems - Towards a formal approach", IFIP Congress, Toronto, 1977, North-Holland Pub., 155-160.

[22] G. Le Lann, "Algorithms for distributed data-sharing systems which use tickets", 3rd ACM/IEEE Berkeley Workshop on Distributed Databases and Computer Networks, August 1978, 259-272.

[23] G. Le Lann, "Distributed real-time processing", in "Computer Systems for Process Control", Plenum Press Pub., 1986, ISBN 0-306-42386-3, R. Güth Ed., 69-88.

[24] C.L. Liu, J.W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment", J.ACM, 20 (1), January 1973, 46-61.

[25] J. Lundelius, N. Lynch, "A new fault-tolerant algorithm for clock synchronization", 3rd ACM Symposium on Principles of Distributed Computing, Vancouver, August 1984, 75-88.

[26] S.R. Mahaney, F.B. Schneider, "Inexact agreement : Accuracy, precision and graceful degradation", 4th ACM Symposium on Principles of Distributed Computing, Minaki, August 1985, 237-249.

[27] K. Marzullo, S. Owicki, "Maintaining the Time in a distributed system", 2nd ACM Symposium on Principles of Distributed Computing, Montreal, August 1983, 295-305.

[28] P. Minet, "Performance evaluation of GAM-T-103 real-time transfer protocols", INFOCOM'89, IEEE Pub., Ottawa, April 1989, 766-773.

[29] A.K. Mok, M.L. Dertouzos, "Multiprocessor scheduling in a hard real-time environment", 7th Texas Conf. on Computing Systems, Houston, October 1978, 5.1-5.12.

[30] C.H. Papadimitriou, "Serializability of concurrent database updates", J.ACM, 26 (4), October 1979, 631-653.

[31] "Draft Project proposal to develop a new X3 standard", Protocol Engines Inc., report PEI 89-12, February 1989.

[32] K. Perry, S. Toueg, "Distributed agreement in the presence of processor and communication faults", IEEE Trans. on Software Engineering, 12 (3), March 1986, 477-482.

[33] D. Powell, "Fault assumptions and assumption coverage - a contribution to the fundamental concepts of dependability", LAAS Research Report n° 90074, February 1990, 16 p.

[34] K. Ramamritham, J.A. Stankovic, P.F. Shiah, "Efficient scheduling algorithms for real-time multiprocessor systems", IEEE Trans. on Parallel and Distributed Systems, 1(2), April 1990, 184-194.

[35] D.P. Reed, "Implementing atomic actions on decentralized data", ACM Trans. on Computer Systems, 1 (1), February 1983, 3-23.

[36] B. Sprunt, L. Sha, J. Lehoczky, "Aperiodic task scheduling for hard real-time systems", Real-Time Systems, Kluwer Academic Pub. (Boston), 1 (1), June 1989, 27-60.

[37] T.K. Srikanth, S. Toueg, "Optimal clock synchronization", J.ACM, 34(3), July 1987, 626-645.

[38] J.A. Stankovic, "Misconceptions about real-time computing - A serious problem for next-generation systems", IEEE Computer, October 1988, 10-19.

[39] R.H. Thomas, "A majority consensus approach to concurrency control for multiple copy databases", ACM Trans. on Database Systems, 4 (2), June 1979, 180-209.

Imprimé en France
par
. l'Institut National de Recherche en Informatique et en Automatique .

ISSN 0249-6399