

DISTRIBUTED REAL-TIME COMPUTING:

WHERE DO WE STAND?



Gerard.Le_Lann@inria.fr

31st Spring School
in Theoretical Computer Science
5-8 May 2003, Porquerolles, France

1. Prolegomena (slide 3)

2. Design Solutions and Computational Models (slide 16)

3. Coverage Issues with Real Systems (slide 42)

Note: This tutorial is based upon acknowledged state-of-the-art in scheduling theory and distributed algorithms theory. Conclusions drawn derive from logical reasoning. Some may look “counter intuitive” to those who have had no exposure to scheduling theory (in particular). The objective of this tutorial is to provide a fair overview of what is involved with “solving a distributed real-time computing problem”.

1. Prolegomena

- Let $\langle X \rangle = \{\langle m.X \rangle, \langle p.X \rangle\}$ stand for the specification of a distributed real-time (RT) computing problem
- Goal: To solve $\langle X \rangle$, and build a system (S) correct % $\langle X \rangle$

$\langle m.X \rangle = \text{models}$ (assumptions) = “adversary” of S = (future) run-time conditions relative to environment and technology, i.e. (1) **postulated models** for {event arrivals, process structures, data sharing, failures, ...}, (2) **postulated bounds** on {“loads”, densities of failure occurrence, of intrusion occurrence, process execution times (wcet), number of processes, ...}

$\langle p.X \rangle = \text{properties}$ sought, i.e. some combination of **Safety (SafeP)**, **Liveness (LiveP)**, **Timeliness (TimeP)**, **Dependability (the above, assuming failures)**, properties

Typical example: for event/request that activates process q , $\langle m.X \rangle$ specifies an arrival model/law (periodic, sporadic, aperiodic, etc.), and $\langle p.X \rangle$ specifies a latest termination deadline $T_q(z)$ (z is the application level of abstraction/implementation of $\langle X \rangle$).

$T_q(z)$ is imposed, dictated by application level considerations. The “user” (problem $\langle X \rangle$ specifier) has no idea of how S is going to be built so as to meet $T_q(z)$ – user does not even know whether this is doable. **$T_q(z)$ is the expression of a “wish”.**

For example, **$T_q(z) = 300 \text{ ms}$** . It is the duty of a scientist/designer to show, with proofs, if and how a system (solution) S may exist.

Consider S' level of abstraction/implementation $z-1$. Clearly, any level $z-1$ operation ω needed by process q cannot take more than **$D_q(z-1) = W_q(z-1) T_q(z)$, $W_q(z-1) < 1$.**

Example: to meet $T_q(\mathbf{z}) = 300 \text{ ms}$, it is **proved** that one must have $D_q(\mathbf{z}-1) < 41 \text{ ms}$. And so on, for every level $h < z$, until reaching level 1.

$W_q(\mathbf{h})$ is a (usually complex) function of (1) q 's level h adversary, i.e. other processes + failures, (2) how q is scheduled – against its adversary – at level h . In fact, given that “low level” operations are accessible to many processes, $D_q(\mathbf{h}) = \min \{D_Q(\mathbf{h})\}$, where Q is **the set of processes that may use ω (q is not the only one)**.

It is the duty of a scientist/designer to establish, with proofs, an analytical expression of every function $W_q(\mathbf{h})$, for every process q , for every level h . Indeed, there is no “trivial” relationship between “given” $T_q(\mathbf{z})$ – the problem – and the $D_q(\mathbf{h})$'s – the solution to be found. The only level where such proof obligations are relaxed is level 1 – basic hardware is assumed to “guarantee” $D_q(\mathbf{1})$.

Solving $\langle X \rangle$ implies establishing (see **Scheduling theory**):

- **$[\Delta]$** , an implementable specification (a “blueprint”) of a system design – $[\Delta]$ specifies a set of [modules] mapped on some [architecture] running under the control of [distributed algorithms], such as mutual exclusion, deadline-driven process scheduling, uniform consensus, algorithms
- **Proofs** of properties $[p. \Delta]$ ensured by $[\Delta]$ in the presence of $\langle m.X \rangle$
- **[FC]**, a specification of **feasibility conditions for pair** $\{\langle X \rangle, [\Delta]\}$, i.e. conditions under which $[p.\Delta]$ dominate (imply) desired properties $\langle p.X \rangle$

Under [FC], **S**, an operational system that implements $[\Delta]$ correctly, will always “win” against “adversary” $\langle m.X \rangle$.

Proofs of properties $[p, \Delta]$ are the $W_q(h)$ functions.

[FC]? Established by writing: for every q , $D_q(z-1) < W_q(z-1) T_q(z)$,
 $D_q(z-2) < W_q(z-2) D_q(z-1)$, ..., $D_q(1) < W_q(1) D_q(2)$.



There are two classes of “approaches” to RT problems, those based upon proofs, those based upon claims or tricks (i.e. no proofs). In fact, there is no choice (see Scheduling theory):

➤ What is essential with RT computing problems?

TimeP proofs!

➤ How do we prove TimeP? By **selecting/specifying** (in $[\Delta]$) **scheduling algorithms**, and by conducting

worst-case schedulability (WCS) analyses!

➤ Are WCS analyses easy? No!

They raise NP-hard combinatorial problems ☹

For WCS analyses, see scheduling theory, queuing theory, (max, +) algebra, constraint programming, combinatorial analysis, graph theory, integer arithmetic, analytical calculus, etc.

Schedulability analyses rest on simplified models of {systems, software processes, COTS, ...}, on approximated {processor architectures, internal “adversaries”, worst-case execution times, contention and/or queuing phenomena, ...}, etc.

⇒ computing the coverage of a WCS analysis may turn problematic

⇒ the simpler a WCS analysis, the higher its coverage

The fact that WCS analyses are not “easy” may explain why some proposals (to solve distributed RT computing problems) rest on claims or tricks (rather than proofs) ...

Nevertheless, WCS analyses cannot be ignored.

First conclusion (C1):

Any design or “approach” claimed to be a “solution” (to a RT computing problem) which is not accompanied with TimeP proofs is a non solution.

(This conclusion is part of the basics in RT computing)

System abstraction levels (simplified view)

ENVIRONMENT		← Assumptions <m.X>
APPLICATIONS	z	← ← Properties <p.X>
M/W	m	← Assumptions <m.X>
OS/MONITORS	...	
.../...	k	← Assumptions <m.X>
.../...	...	
END-TO-END COMs	c	← Assumptions <m.X>
.../...		
NETWORK AND I/O	2	← Assumptions <m.X>
BASIC H/W (PHYSICS)	1	← Assumptions <m.X>

➤ What is essential with distributed RT computing problems?

End-to-end TimeP proofs!

Consider $\text{Thread}(j)$, a **distributed operation** (a thread of computation and communication) encompassing:

- $p(j)$ and $q(j)$, **two level j processes** on two separate processors,
- the **network** interconnecting these processors.

Whenever $\langle X \rangle$ specifies a **distributed** RT computing problem, **any $[\Delta]$ rests on $D(j)$** , an upper bound on (end-to-end) $\text{Thread}(j)$ delays, for all levels j of interest.

(C1) → an analytical expression must be established for every end-to-end bound $D(j)$.

$D(j) = P(j) + N(j) + Q(j)$, where:

$P(j) = \text{wcet}\{p(j)\} + \mathbf{W}\{p(j)\}$, $Q(j) = \text{wcet}\{q(j)\} + \mathbf{W}\{q(j)\}$,

$N(j) = \text{wcet}\{\text{message}(j)\} + \mathbf{W}\{\text{message}(j)\}$,

where $\mathbf{W}\{x(j)\}$ = highest waiting delays experienced by $x(j)$ due to (1) schedulers, (2) “favored” concurrent processes and messages.

To find the wcet’s (worst-case execution times) is “easy” (they are problem X -independent and design Δ -independent). Conversely, the $\mathbf{W}\{x(j)\}$ ’s are:

(1) problem X -dependent! (2) design Δ -dependent!

Second conclusion (C2):

Any design or “approach” claimed to be a “solution” (to a distributed RT computing problem) which is based on assuming a priori knowledge of bounds $D(j)$, or assuming that bounds $D(j)$ are problem X -independent, is a non solution.

Most problems in distributed RT computing are open. There are only a few solutions (i.e. TimeP proofs), even less solutions proved optimal.

State-of-the-art (examples):

- ▣ **Distr. access to a shared communication channel** (bus, ...)
 - Late 70's, Pippenger, Gallager, Capetanakis, ... proved that optimal solutions are tree protocols (e.g., deterministic Ethernets)

- ▣ **Holistic scheduling** (sender, network, receiver)
 - York Univ., Pisa, CMU, ...

- ▣ **Timely Serializability**: Graph-shaped processes, each spanning several processors, sharing updatable persistent variables
 - Classified work for DGA/DSP (French Darpa)

▣ Distr. access to a shared communication channel

In $[\Delta]$, multiaccess algorithm = deterministic balanced tree search.
Analytical expression of FC (worst-case “loads”, for any message of priority j) \Rightarrow analytical expression of message delay bound (j)
 \Rightarrow TimeP proofs for deterministic Ethernets

$$\xi_k^t = \begin{cases} \frac{\lceil \log_m \left(m \lfloor \frac{k}{2} \rfloor \right) \rceil - 1}{m-1} + m \lfloor \frac{k}{2} \rfloor \left\lceil \log_m \left(\frac{t}{m \lfloor \frac{k}{2} \rfloor} \right) \right\rceil - \left(k - m \lfloor \frac{k}{2} \rfloor \right) & \text{if } k \in \{2, \dots, t\}, \\ 0 & \text{if } k = 1, \\ 1 & \text{if } k = 0. \end{cases}$$

$$t = m^n, m \in \mathbb{N}^* \setminus \{1\}, n \in \mathbb{N}^*.$$

Worst-case time for processing k colliding messages, with t -leaf trees of branching degree m

Recommended reading:

IEEE/ACM Transactions on Networking
Real-Time Systems Journal (Kluwer)
Proc. IEEE Real-Time Systems Symposium

Reasons for scarce results? With few exceptions:

- The “distributed algorithms” community (PODC, DISC, ...) ignores issues of waiting queues, WCS analyses, ... ignores scheduling theory, queuing theory,
- The “real-time scheduling” community (RTSS, ...) ignores issues of distribution (“distribution” is used to mean physical separation),
- It is believed that designs aimed at solving distributed RT computing problems must inevitably be synchronous.

Which computational model \mathcal{M} for establishing $[\Delta]$?

2. Design Solutions and Computational Models

Modules, algorithms in $[\Delta]$ perform **operations (computation, communication steps)**. Given that we have to solve **distributed (RT) computing problems**, it would make no sense to provide ourselves with an **(assumed) distributed operation** that would amount to solve these problems “magically”.

Hence, operations part of the computational models that can be considered are **local operations**, such as e.g., successful write in cache-memory, OS or M/W call/service, send/receive message, execution of a CPU instruction.

If **distributed operations** are needed, they must be **designed out of (assumed) local operations (rather than “assumed”)**.

For example, one may need a “reliable broadcast” operation. Such an operation can be constructed out of local operations (see published state-of-the-art). Therefore, **assuming such distributed operations as Thread(j) is not acceptable. One must show how Thread(j) is constructed.**



Computational model $\mathcal{M} \Leftrightarrow$ timing assumptions (TA(\mathcal{M}))

TA(\mathcal{M}) = postulated upper and lower bounds for delays of some local operations, at some levels.

These bounds appear in design solutions (in $[\Delta]$). Note that with RT computing problems, postulated bounds must hold true from time 0.

\mathcal{M} = Pure Asynchronous:

No TA(\mathcal{M}) at all postulated at any level \Rightarrow no coverage issue involved.

Unfortunately, many problems in distributed fault-tolerant computing (p.X = SafeP + LiveP + DepP) have no deterministic solution in this model.

\Rightarrow “augmented” (pure asynchronous) models

\mathcal{M} = Asynchronous (Async):

Pure asynchrony “augmented” with some **time-free semantics**

\Rightarrow algorithmic solutions in $[\Delta]$ are time free

\mathcal{M} = Partially Synchronous (ParSync) (various models):

- $t_ParSync$ = Pure asynchrony “augmented” with timed semantics, $TA(\mathcal{M})$ stated at some levels, for some operations,
 - $tf_ParSync$ = Pure asynchrony “augmented” with time-free functions of $TA(\mathcal{M})$,
 - etc.
- \Rightarrow algorithmic solutions in $[\Delta]$ use $TA(\mathcal{M})$ or functions of $TA(\mathcal{M})$

Pure Synchronous (Sync) is a particular case of $t_ParSync$:

$TA(\mathcal{M})$ stated for every level, for every operation.

Note: The TT approach [Kopetz 1998] is a particular case of Sync. Most delays are assumed to be constant (rather than variable).

\mathcal{M} = Globally Synchronous (G_Sync):

A subset of Sync models or t_ParSync models where $TA(\mathcal{M})$ include end-to-end bounds $D(j)$, for some level(s) j .



Whenever the $TA(\mathcal{M})$ encompass such bounds as $D(j)$, there is a ***circularity problem***. Assuming a priori knowledge of bound $D(j)$ implies assuming that distributed operation $Thread(j)$ is available, without showing how it is constructed. This is not acceptable – see slide 17.

Furthermore, recall (C1): one **solves** a **distributed RT computing** problem X by **giving analytical and computable expressions of end-to-end bounds $D(j)$** , established out of some **$TA(M)$ local to a processor or a network node**.

Therefore, assuming a priori knowledge of such bounds (analytical expressions missing) is strange – problems central to X are assumed to be solved a priori, and magically ☺ ...

Third conclusion (C3):

It is self-contradictory to consider G_Sync models for solving a distributed RT computing problem (the “solution” is in the assumptions).

A popular example of Async models:

Time-free semantics “added to” pure Async are **Chandra/Toueg’s “unreliable failure detectors”** – FDs [Chandra/Toueg 1991].

Such semantics match “low” levels, exactly like (physical) clocks.

These Async models are considered by many authors (including ourselves – see [Hermant/Le Lann 2002]).

The beauty of Async designs is that they preserve SafeP (LiveP as well sometimes) when time bounds (postulated with non Async designs) are violated.

A popular belief:

Async design solutions and TimeP are antagonistic...

That belief is unfounded. For more on this, see (further) the design immersion principle.

Two categories of approaches proposed for addressing distributed RT computing problems:

① Proposals based upon G_Sync models

These proposals rest on design models that are extensions of the pure Sync model, i.e. solutions rest on “early binding” to synchrony assumptions.

② Proposals based upon the “design immersion” principle

These proposals rest on distinguishing design models (proofs) and implementation models (development), i.e. solutions rest on “late binding” to synchrony assumptions.

(As is well known, synchrony assumptions are harmful/risky).

① Proposals based upon G_Sync models

→ models \mathcal{M} considered consist in postulating (in particular) a priori knowledge of some bound(s) $D(j)$ (end-to-end Thread(j) delays). These bounds are used explicitly in designs $[\Delta]$. Hence, conclusions (C2) and (C3) apply.

In the G_Sync category:

- ◆ The TA model [Cristian/Fetzer 1999] – the $W\{x(j)\}$'s are ignored
- ◆ The TCB approach [Verissimo/Casimiro/Fetzer 2000] – the $W\{x(j)\}$'s are assumed to be zero thanks to “special” resources

Note: With The TT approach, the $W\{x(j)\}$'s are declared “ruled out” (the single lane highway “theorem” 😊)

Excerpts from [Cristian/Fetzer 1999]:

- ◆ End of Section II: “TA does not make any assumptions about load patterns”.
- ◆ Subsection B1: “TA assumes the existence of a one-way time-out (message) delay δ ”. “There is a known ratio k such that δ is equal to D/k ” (D here is a $T_q(z)$ TimeP specified in $\langle p.X \rangle$).
 - Reality: Neither δ nor k are “known”. An analytical expression of δ must be established, via a WCS analysis. Ratio k is a $W\{x(j)\}$ function, which must be established, via a WCS analysis (k cannot be “known” a priori).
- ◆ Subsection B1: “The choice of a good δ is not always easy since ... delays increase with ... network loads”.
 - An obvious contradiction with the first excerpt 😊
 - If no load assumptions, no bounds such as δ (obvious)!!!

Assertions such as “**a system alternates between “good” and “bad” periods**” are just wishes.

It may well be that a system designed in the TA model ends up being always either incorrect or mute (muteness resulting from “timing failures detection”).

Counter-argument: pick up some “really big” $D(j)$ (e.g., 1 hour), and that $D(j)$ will never be violated!

Wrong!! In the absence of FC, you cannot predict whether system S will or will not be “overloaded” or entering thrashing “too often”. Whenever the case, real $D(j)$ is infinite!!!

Furthermore, you are not entitled to pick up any arbitrarily big value for $D(j)$ if you want to meet those TimeP specified in $\langle p.X \rangle$.

Recall the example on slide 5: to meet $T_q(\mathbf{z}) = 300 \text{ ms}$, one must have any $D_q(j) < \text{some fraction (analytical expression required) of } 300 \text{ ms}$.

Note:

Recall that it suffices to consider an Async model for proving SafeP and LiveP. Assuming timed semantics is an overkill. Considering the TA model when SafeP and LiveP only must be achieved is a bad strategy, which leads you to address the (very complex) issues raised with proving TimeP – every timing assumption that underlies TA must be transformed into a timeliness bound, analytical expression given!

Excerpts from [Verissimo/Casimiro/Fetzer 2000]:

- ◆ Section 4.2: “There exists an known upper bound T_D on the delivery delay of messages exchanged between TCB modules”.
→ Reality: T_D has no “existence”, unless proven. An analytical expression of T_D must be established, via a WCS analysis (which involves giving a $W\{x(j)\}$ function).
- ◆ Subsection 6.1: “By definition, the timely execution service is ... for short-lasting time-critical application functions. These functions should not reside in the application address space”.
→ Reality: There are many systems where almost every application process is, in its entirety, a time-critical function, even those which are not “short-lasting”. TCB does not apply. Functions “out of the process address space” → processes post “calls” (to TCB). Any difference with systems where processes post “calls” to a RT operating system/monitor?

The “TCB/wormhole” approach

Consider a set of application processes and data, a set of system processes and data (operating systems, middleware, ...) running on some (distributed) computing system S . The whole thing is referred to as a payload. State that the payload is supposed to solve some problem X (safety, liveness, timeliness, dependability, properties), and observe that this is impossible, because of the assumptions made regarding the payload (full asynchrony, omission or Byzantine failures, etc.).

Claim that you have a solution (even for problems X shown to be unfeasible). The solution is ... buy yourself a “TCB/wormhole”, some “small” set of computers and networks (“added to” S) which are claimed to be always timely, available, reliable, etc.

How is that possible? Simple! You just have to change the assumptions made for the payload, and state that “TCB/wormhole” resources may fail by crashing only, and that they are conformant to a pure Sync model. How can these assertions be trusted? No proofs, just claims or assumptions. Regarding timeliness, there are a number of pending questions.

It must be the case that every “TCB/wormhole” processor is shared by some number of payload processes – otherwise, every payload process owns its private “TCB/wormhole”, which does not make sense in a (distributed) system. Hence, varying and possibly unbounded loads are experienced by every “TCB/wormhole” processor. How can it be that **bound T_D (see slide 28)** may exist, no matter what? As is well known, delays/response times depend on loads!

Furthermore, a “TCB/wormhole” is stated to be “small”. Let’s imagine that a “small” “TCB/wormhole” is timely for up to 10 payload processes. Is a “TCB/wormhole” capable of delivering timely services to 2,000 payload processes going to be “small” as well?

If there is a “solution” (?) for designing timely “TCB/wormhole” processors/networks, why not using it for designing timely payload processors/networks?

Obvious: Any “wormhole” is also subject to contention and waiting queue phenomena.

Therefore, WCS analyses are mandatory! No difference whatsoever with what may/should be done for a payload!!!

The “wormhole approach” to distributed RT computing problems belongs to the class of approaches based upon “claims” and “tricks” – see slide 7.

The difficult issues (computation of the $W\{x(j)\}$) are believed to be solved thanks to ???

The “TCB/wormhole” claims are similar to the following claim. Whenever you are driving your car in a city at rush hour, and you have to meet a strict arrival deadline, you just have to drop your car and drive one of the few Ferraris provided by the city (the “wormhole” cars) on some “small” streets, that are never congested 😊 Anything wrong?

In a traffic jam, Ferraris are not faster than any other car! Streets that are “never congested” at rush hours are ... a dream!

By definition, every driver who is nearing his/her deadline would like to drive on such streets, hence making them ... congested 😊

The “simple” idea that one can ignore the work accomplished by armies of very respectable scientists in such areas as, e.g., complexity theory, who have established a few optimality results after having demonstrated that scheduling problems are NP-hard, should be treated highly suspiciously.

In fact, the “trick” behind the “TCB/wormhole” approach consists in hiding the fact that what is called the payload is what is usually referred to as the “environment”, and that the (real) system of concern (called the “TCB/wormhole”) is built out of very conventional solutions, that work for “nice” assumptions.

Nothing new!!!

Assertions such as “*timing failures can be detected and transformed into stop or omission failures*” (TA, TCB) are just wishes. **To be trusted, such assertions should rest on proofs that timing failures are detected and transformed into stop or omission failures in due time (otherwise, a “late” message can be read by a destination process before being “cancelled”).**

This raises non trivial scheduling issues – **doing “timing failure detection” is not just a matter of setting and resetting timers.** Whenever these issues are not addressed (the case with TA and TCB), **either such assertions are unfounded, or they amount to assume infinite computational power.**

One “great idea” behind TA and TCB is that one can use timers that awake** whenever a timing assumption is violated.

Timers don’t suffice. It is necessary to activate special processes that “measure-compare-and-(abort or commit)” every end-to-end activity. This does not come for free. Obviously, performance that can be achieved with a TA/TCB design “augmented” with such special processes can only be worse than that achieved with a design that does not need such processes.

**TA and TCB systems can be telling (most of) the “bad news”.
But scientists are asked to show how to build systems that
ensure “good news” 😊**

** eventually

Furthermore, one finds the following statement (U): “There is no upper bound assumed for failure occurrence”.

Hence, the **number of payload processor crashes** which may be artificial, i.e. caused by a poor guess about δ (too many detections of so called “timing failures”), **is unbounded**.

- **Who is interested in using systems that can turn mute, arbitrarily often, for arbitrarily long time intervals?** (assuming “timing failure detection” works always, which is not the case)
- **Algorithms** that provably solve **distributed fault-tolerant computing problems explicitly use a variable f , an assumed bound on failure occurrence**, and f must be valued before usage (in a real system). **Hence, statement U cannot hold true. The trick is – as noted before – that U is “cosmetics” for the payload, and does not apply to the TCB/wormhole.**

② Proposals based upon the “design immersion” principle

[Le Lann 1995], [Hermant/Le Lann 2002], [Le Lann/Schmid 2003]

→ models M considered for designs $[\Delta]$ are Async models.

Rationale

First observation: It is “risky” to trust proven TimeP bounds (it is even more dangerous to trust timing assumptions)

→ it is risky to choose $M(\Delta) = \text{Sync}$ or $t_ParSync$ or G_Sync

Second observation: a Sync model need be considered only when time has come to conduct WCS analyses (to prove TimeP).

Third observation: design model $M(\Delta)$ and implementation model $M(S)$ need not be the same.

→ A design Δ conducted in model $M(\Delta)$ can be immersed into/bound to a model $M(S)$ iff $M(S)$ is not weaker than $M(\Delta)$

Of particular interest is the case where $M(\Delta)$ is Async and $M(S)$ is Sync ☺

If S is capable of making any step in finite and bounded time, any Async algorithm A runs in finite and bounded time, as does the “adversary” of A (other algorithms). Hence, a WCS analysis can be conducted, so as to establish the TimeP bounds/proofs for A .

The “design immersion” principle (also called “late binding”):

For establishing some design Δ aimed at some problem X

Step 1: $M(\Delta)$ = Async. Give $[\Delta]$. Prove SafeP and LiveP ensured by $[\Delta]$, for every level of interest.

Step 2: immerse Δ in $M(S)$, the Sync model that matches (physical) system S . Prove TimeP – bounds $D(\cdot)$ – ensured by $[\Delta]$ run on S (establish [FC]), for every level of interest.

Essential observation:

Timeliness bounds $D(h)$ – TimeP proofs – are not “wired in” design Δ (pure asynchronous algorithms only in Δ).

Of course, the “*design immersion*” principle can also be applied to any design aimed at implementing a computational model – which is a problem of its own, not just “implementation details” or a problem for “engineers”.

For establishing some design $\nabla(\mathcal{M}(\Delta))$ aimed at implementing model $\mathcal{M}(\Delta)$

Step 3: give $[\nabla(\mathcal{M}(\Delta))]$, a specification of a design for implementing the time-free semantics of $\mathcal{M}(\Delta)$.

Step 4: in case $[\nabla(\mathcal{M}(\Delta))]$ is time-free, immerse $[\nabla(\mathcal{M}(\Delta))]$ in $\mathcal{M}(\mathbf{S})$. Establish those TimeP proofs needed to demonstrate implementation correctness of the time-free semantics of $\mathcal{M}(\Delta)$.

This is in contrast with approaches based on **“early binding” to some timed semantics** at level k – timings are assumed or, better TimeP proofs ([FC]) are established for level k – in order to prove SafeP and LiveP at level $k+1$. Timeliness bounds $D(h)$ – TimeP proofs – are “wired in” design Δ .

In case timing assumptions (the no proof approaches), or proven timeliness bounds, are violated at run-time:

➤ **Timed semantics based approaches (Sync, t_ParSync, G_Sync)**
TimeP are lost, SafeP and LiveP may be lost.

➤ **Design immersion based approaches (design model is Async)**
TimeP are lost, SafeP cannot be lost, LiveP may not be lost.

3. Coverage Issues with Real Systems

We are now concerned with **real RT applications and real operational systems**. With real RT applications, $\langle X \rangle$ is supplemented with $C(S)$ – a “user” defined requirement.

$C(S)$ = lower bound set for $Cov(S)$, the coverage of (future) S .

$Cov(S)$ = likelihood or probability that S ' behavior is “correct”.

For example, with safety-critical systems, $C(S) \approx 1 - 10^{-9}$.

Note that $Cov(S) < Cov(X)$, even if (1) $[\Delta]$ provably solves $\langle X \rangle$, (2) S implements $[\Delta]$ provably correctly (our assumption here).

This is due to the fact that:

- ◆ $\langle m.X \rangle$ has some coverage (who knows the future exactly?)
- ◆ $M(\Delta)$ has some coverage.

For violations of $\langle m.X \rangle$ that lead to violations of proven TimeP, see conclusions slide 48: Async designs dominate non Async designs. For violations of $\langle m.X \rangle$ regarding assumed bounds on failure occurrence, that Async designs preserve SafeP at no extra cost (no need to do “timing failure detection”) is well known.

Let us now concentrate on the other (often neglected) issue.

Besides having to show how to implement $[\Delta]$, we are asked to show: (1) how $M(\Delta)$ can be implemented, (2) which $M(\Delta)$ maximizes $Cov(M(\Delta))$.

More precisely, two requirements derive from requirement C(S):

- **(R1): $Cov(M(\Delta))$ should be computable,**
- **(R2): $Cov(M(\Delta)) > C(S)$.**

How do we meet requirements **(R1) and (R2)**?

➤ $\mathcal{M}(\Delta) = \text{Sync}$ or t_ParSync (hence G_Sync)

One must transform every postulated bound $\text{TA}(\mathcal{M})$ into a TimeP proof, which involves conducting **WCS analyses from level 1 up to level z, the level of problem X (even if $\langle X \rangle$ is not a RT computing problem)**. Alas – recall (C2) – such analyses are:

- (1) problem X-dependent,
- (2) design Δ -dependent.

➤ $\mathcal{M}(\Delta) = \text{Async}$

One must prove those TimeP needed to implement $[\nabla(\mathcal{M}(\Delta))]$, the design specification of $\mathcal{M}(\Delta)$'s time-free semantics, which involves conducting **WCS analyses from level 1 up to level c only, the exposed level of time-free semantics** – see steps 3 or 4 slide 40 **(even if $\langle X \rangle$ is a RT computing problem)**.

Fortunately, analyses in the latter case are:

- (1) **almost fully** problem X -**in**dependent (“adversary immunity”),
- (2) **fully** design Δ -**in**dependent.

Furthermore, **$c < z$** .

\Rightarrow The complexity of WCS analyses involved with
implementing $\mathcal{M}(\Delta) = \text{Async}$ is necessarily smaller than
that of analyses involved with
implementing $\mathcal{M}(\Delta) = \text{Sync}$ or $t_ParSync$ (hence G_Sync)

\Rightarrow the coverage of WCS analyses involved with
 $\mathcal{M}(\Delta) = \text{Async}$ is necessarily higher than that of WCS analyses
involved with **$\mathcal{M}(\Delta) = \text{Sync}$ or $t_ParSync$ (hence G_Sync)**

For example, in [Hermant/Le Lann 2002], one shows how to build **Fast FDs – Strong or Perfect semantics** – provably correctly.

Tight timeliness bounds for level c FD messages are established out of basic H/W level timing assumptions (level 1). Weak complexity is due to the choice of “head-of-the-line” scheduling policy – see Queuing theory. Such a choice is **fully** problem X-**in**dependent.

If immediate resource preemption is feasible, WCS analysis is **fully** problem X-**in**dependent. Otherwise, WCS analysis is **almost fully** problem X-**in**dependent. The only dependencies with problem X are (1) largest level c process wcet, (2) largest level c message duration (“blocking factors” in Scheduling theory).

Reality?

The worst-case time optimal **Async Uniform Consensus algorithm** [Hermant/Le Lann 2002] and its extension (**Uniform Coordination**) based upon **Fast FDs** have been implemented in 2002 by a software company, delivered to ASTRIUM (a European satellite manufacturer) as **M/W “building blocks” for new spaceborne applications**. Measured timeliness bounds match those predicted following the “design immersion” principle.

Fast Perfect FDs are implemented “atop” a link protocol (OSI level 2) → very small failure detection latency.

Demonstrator installed at European Space Agency/ESTEC premises in July 2003. M/W “building blocks” to be made available to European industry by ESA/ESTEC.

Fourth conclusion (C4):

Async models dominate other computational models

⇒ Regarding computability of $\text{Cov}(\mathcal{M}(\Delta))$

⇒ Regarding achievable values of $\text{Cov}(\mathcal{M}(\Delta))$

⇒ Regarding tightness of demonstrated timeliness bounds $D_{\mathcal{M}(\Delta)}$
(or system “performance” in general) for any given $C(\mathcal{M}(\Delta))^{**}$

⇒ Either $\{D_{\text{non Async}} = D_{\text{Async}} \text{ and } \text{Cov}(\text{non Async}) < \text{Cov}(\text{Async})\}$
or $\{\text{Cov}(\text{non Async}) = \text{Cov}(\text{Async}) \text{ and } D_{\text{non Async}} > D_{\text{Async}}\}$

** Demonstration given in a forthcoming paper.

Nancy Lynch, 1996:

“It is impossible or inefficient to implement the synchronous model in many types of distributed systems.”

Early binding to synchrony is particularly risky with real systems built out of COTS products (Have you ever tried to get [FC] or TimeP proofs from vendors of “real-time monitors” 😊 ?)

Example: the Mars Path Finder mission (1997) almost failed due to lack of [FC] (for VxWorks). Cause? PI, the “priority inheritance” option, had been set to “off” (should have been “on”). Official diagnosis? A software fault (as usual!). Why? Because boolean PI is “software” implemented! Imagine PI implemented as a mechanical switch. Diagnosis would have been “it’s a mechanical fault”. Stupid. Lack of [FC] is a design and validation fault – lack of proofs (nothing to do with software/hardware implementation).

Performance/efficiency issues: an illustration

Commonly held view (in some circles):

Sync/TT (time-triggered) solutions are faster, more efficient, more predictable (???), than Async/ET (event-triggered) solutions.

Example: a shared multi-access communication medium

- 10 message sources
- Every 10 messages, each source generates 9 messages of duration 1 each, and 1 message of duration 10
- Up to x messages may be submitted concurrently, $1 \leq x \leq 10$

TTP multi-access scheme = conventional Static_Sync_TDMA

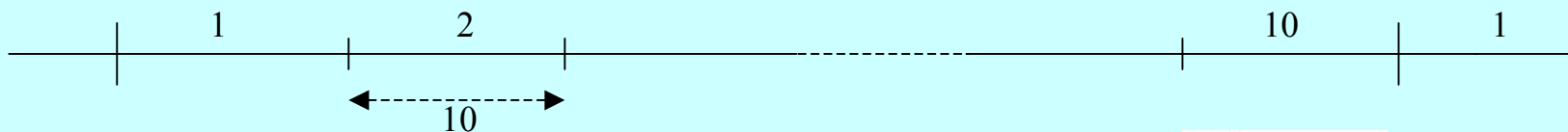
⇒ a pre-computed cyclic frame of 10 consecutive time slots,
each of duration 10

w = protocol overhead per frame + 10 inter-slot “guard times”

Performance figures:

T = worst-case waiting time(message) = $100 + w$

Efficiency ratio (w ignored) = 0.19



Deterministic Ethernets = conventional Async_TDMA

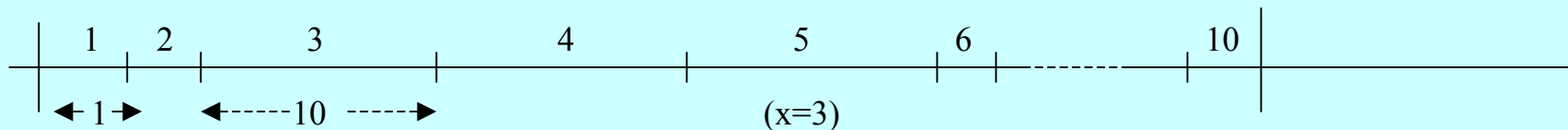
⇒ CSMA + deterministic contention resolution

w = protocol overhead per 10 messages

Performance figures:

$T = \text{worst-case waiting time(message)} = 10 + 9x + w$

Efficiency ratio (w ignored) = 1



Not surprisingly, deterministic Ethernets win over TTP channels.

This simple example is an illustration of well-known results in **computer networking**.

For problems in **distributed computing**, similar conclusions hold.

Reason?

TT/Sync performance figures are $\max\{\max\}$ functions,
whereas ET/Async performance figures are $\min\{\max\}$ functions

The ultimate quest

How “far” can we go in reducing reliance upon timed semantics? (Such reliance being “risky”, inevitably)

A recent result (that illustrates steps 3 and 4 of the “design immersion” principle):

It is possible to design a (time-free) solution in a `tf_ParSync` computational model for implementing the (time-free) Perfect Failure Detector assuming any failure model (from stop to Byzantine) [joint work with Ulrich Schmid, T.U. Vienna]