# Fast Asynchronous Uniform Consensus in Real-Time Distributed Systems

Jean-François Hermant and Gérard Le Lann, *Member*, *IEEE Computer Society*

**Abstract**—We investigate whether asynchronous computational models and asynchronous algorithms can be considered for designing real-time distributed fault-tolerant systems. A priori, the lack of bounded finite delays is antagonistic with timeliness requirements. We show how to circumvent this apparent contradiction, via the principle of "late binding" of a solution to some (partially) synchronous model. This principle is shown to maximize the coverage of demonstrated safety, liveness, and timeliness properties. These general results are illustrated with the Uniform Consensus (UC) and the Real-Time UC problems, assuming processor crashes and reliable communications, considering asynchronous solutions based upon Unreliable Failure Detectors. We introduce the concept of Fast Failure Detectors and we show that the problem of building Strong or Perfect Fast Failure Detectors in real systems can be stated as a distributed message scheduling problem. A generic solution to this problem is given, illustrated considering deterministic Ethernets. In passing, it is shown that, with our construction of Unreliable Failure Detectors, asynchronous algorithms that solve UC have a worst-case termination lower bound that matches the optimal synchronous lower bound, that is, $(t+1)\,D$, where $t$ is the maximum number of processors that may crash and $D$ is the maximum interprocess message delay. Finally, we introduce *FastUC*, a novel solution to UC, that is based upon Fast Failure Detectors. *FastUC* has a worst-case termination time that is sublinear in $t\,D$. For most practical cases and common values of $t$, *FastUC* terminates in $D$, making it a worst-case time optimal solution to Real-Time UC.

**Index Terms**—Asynchronous computational models, partially synchronous computational models, coverage, uniform consensus, real-time distributed fault-tolerant computing, safety, liveness, timeliness, unreliable failure detectors, schedulability analysis.

## 1 INTRODUCTION

MANY real computer-based applications raise combined distributed, real-time, dependable computing issues, even if not explicitly stated as such in application semantics. Therefore, algorithms that solve distributed fault-tolerant computing problems and scheduling problems altogether are central to the design and development of provably correct systems supporting such applications. Without fast algorithms, many time-constrained fault-tolerant services that are mandatory with such applications cannot be contemplated. Examples are consensus, atomic broadcast, leader election, group membership, system reconfiguration, mission (re)planning, online (re)validation or verification, distributed scheduling, replicated data consistency, distributed coordination.

In the presence of processor crashes and reliable communications—our assumptions throughout this paper—a well-known optimal worst-case termination time for such services as consensus or atomic broadcast in synchronous models is $(t+1)\,D$, where $t$ is the maximum number of processors that may crash and $D$ is the maximum interprocess message delay. The $(t+1)$-rounds lower bound was first proven for Byzantine failures [6] and was later extended to crash failures. In systems where safety is of particular concern, postulated bound $t$ should not be violated at runtime. Consequently, "high" values of $t$ must be considered. Moreover, in real systems, many processes contribute to network and processor "loads," which may result into "high" values of $D$. Therefore, worst-case termination times may be too high, which hinders the use of existing solutions in real systems.

In this paper, we introduce the concept of Fast Failure Detectors, which builds upon Unreliable Failure Detectors [1], denoted FDs. With Fast FDs, worst-case failure detection times may be (very) small compared to $D$, which permits designing algorithms that have worst-case termination times sublinear in $t\,D$. We introduce such an algorithm, referred to as *FastUC*, that solves the uniform consensus problem—denoted UC. *FastUC* is worst-case time optimal. Under some (realistic) conditions, *FastUC* terminates in exactly $D$, the absolute worst-case lower bound.

In order to establish these new results, we have departed from approaches that are traditional in the area of real-time computing, namely, assuming timed semantics or some (partially) synchronous computational model for designing a solution. We discuss merits and drawbacks of various models regarding coverage and performance. Two basic observations underlie our work. First, the coverage[1] of runtime correctness is highest with asynchronous solutions. Second, the coverage of timeliness properties is "modest," necessarily (much) smaller than the coverage of safety or liveness properties. This leads to the principle of "late binding" designs to some (partially) synchronous model. With this principle, the apparent contradiction between 1) retaining asynchronous solutions when designing real-time systems and 2) the need to consider some (partially)

• *The authors are with INRIA, Domaine de Voluceau, B.P. 105, 78153 Le Chesnay Cedex, France.*
*E-mail: {Jean-Francois.Hermant, Gerard.Le_Lann}@inria.fr.*

*Manuscript received 14 Apr. 2001; revised 2 Jan. 2002; accepted 1 Feb. 2002.*
*For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 115805.*

1. The coverage of an assertion is the probability or the likelihood that this assertion holds true.

synchronous model to conduct schedulability analyses, vanishes.

Our results permit us to conclude that, for the class of problems considered, real-time distributed systems built out of asynchronous solutions are "safer" and more efficient than systems based upon (partially) synchronous solutions, contrary to intuition or belief.

In Section 2, we examine coverage and efficiency issues related to computational models and we present the "late binding" principle. A generic architectural model of systems under consideration is given in Section 3. In Section 4, we introduce the Real-Time UC problem. In Section 5, we introduce Fast FDs and show that the implementation of (Strong or Perfect) FDs or Fast FDs can be stated as a generic distributed message scheduling problem, which we solve. In passing, we show how to match the $(t+1)$-rounds lower bound with asynchronous algorithms. A numerical illustration of these results is given in Section 6, considering deterministic Ethernets. In Section 7, we introduce *FastUC*, a worst-case time optimal algorithm for Real-Time UC.

## 2 On Computational Models

Let $\langle X \rangle$ be the specification of some problem in computer science, $[\Delta]$ be the specification of a design solution, and *Proofs* be the set of demonstrations proving that $[\Delta]$ solves $\langle X \rangle$ under some *design assumptions*. For our purposes, it suffices to consider that a design solution consists of a set of (virtual or physical) modules, structured after some architecture and equipped with a set of algorithms. We restrict our scope of attention to deterministic algorithms. Let $S$ denote a physical system that implements $[\Delta]$.

In order to specify some *real* problem $X$, one must specify the (future) operational environment of $S$ as accurately as possible. This is achieved via subset $\langle m.X \rangle$ of $\langle X \rangle$, which states which models (*problem assumptions*) are considered, such as, e.g., an arrival model for an event ("load" model), a failure model for a processing module. Actually, $\langle m.X \rangle$ specifies the adversary of $S$. Given that an adversary must be an accurate modeling of some reality, an adversary cannot be "simplified."[2] $[\Delta]$ should be such that the specification of desired properties, denoted $\langle p.X \rangle$, cannot be violated as long as 1) the adversary behaves as specified and 2) *design assumptions* are not violated. Properties of interest are safety, liveness, and timeliness ("real-time"), denoted SafeP, LiveP, and TimeP, respectively.

When conducting theoretical work, any computational model may be considered. Theoretical works are essential in that they permit establishing optimality and/or impossibility results. However, when considering real systems, that is, when issues raised with implementing a model must be addressed, then the choice of a model is constrained by a number of requirements. Let $\mathcal{M}(\Delta)$ stand for a model considered at design time and $\mathrm{C}(a)$ denote the *highest achievable* coverage of assertion $a$. Computational models range from pure synchronous—denoted Sync, to pure asynchronous—denoted pure Async, characterized by their intrinsic *timing assumptions* [21]. Most often, systems are structured after a number of levels of abstraction or implementation. Consider, in $[\Delta]$ and *Proofs*, variables that represent durations of computational or communication steps, for every level of interest. As defined in [4], pure synchrony means that every such variable is assumed to have an upper bound and these upper bounds are known at design time, while pure asynchrony means that no upper bound is assumed for any of these variables. Hence, $\mathrm{C}(\mathcal{M}(\Delta))$ is the highest achievable probability or likelihood that none of those timings postulated via $\mathcal{M}(\Delta)$ can be violated at runtime.

2. Such "simplifications" underlie a number of approaches, notably synchronous designs, valid only in the absence of waiting queue phenomena. Obviously, such "simplifications" are illegal, for they amount to falsifying real $\langle X \rangle$.

### 2.1 Computational Model Coverage Issues

In case $X$ is some real problem, $\langle X \rangle$ also specifies $\mathrm{cov}(\langle p.X \rangle)$, a lower bound set for the coverage of $\langle p.X \rangle$ under $\langle m.X \rangle$. Hence, for a correct design to be an acceptable solution, it must be that $\mathrm{C}(\textit{design assumptions}) > \mathrm{cov}(\langle p.X \rangle)$. One essential ingredient of *design assumptions* is $\mathcal{M}(\Delta)$. It follows that the choice of $\mathcal{M}(\Delta)$ must meet the following coverage requirements:

- (R1) $\mathrm{C}(\mathcal{M}(\Delta))$ can be accurately computed or estimated,
- (R2) $\mathrm{C}(\mathcal{M}(\Delta)) > \mathrm{cov}(\langle p.X \rangle)$.

A nonexisting assumption cannot be violated. Therefore, coverage issues involved with (R1) and (R2) do not arise with the pure Async model. Unfortunately, many problems of interest do not have deterministic solutions in this model [7]. This has motivated work directed at "augmenting" this model with some semantics so as to circumvent impossibility results. One can identify two classes of "added semantics," namely, timed semantics and time-free semantics. FDs [1] are an example of time-free semantics. Models that match the pure Async model augmented with timed semantics are known under the name of partially synchronous models—denoted ParSync, where some modules, or some levels, are assumed to match pure synchrony assumptions, whereas others match pure asynchrony assumptions [3], [4]. In the Sync model, every module or level matches pure synchrony assumptions. Sync being an extreme instance of ParSync models, every result established for ParSync models applies to Sync a fortiori. Models that match the pure Async model augmented with time-free semantics will be referred to as asynchronous models—denoted Async.

A model that can be implemented (and its correctness proven) by postulating a limited set of "hardware" or "low" level timings, referred to as *residual* timings—i.e., timings that are considered to have a coverage arbitrarily close to 1—will be referred to as a *weak* model.

Obviously, the higher the number of timing assumptions and/or the higher the levels considered, the smaller the resulting coverage and the more inaccurate the estimate of that coverage. Consequently:

$$\mathrm{C}(\text{weak Async}) > \mathrm{C}(\text{Async}) > \mathrm{C}(\text{weak ParSync}) > \mathrm{C}(\text{ParSync}) > \mathrm{C}(\text{Sync}),$$

and the likelihood of meeting (R1) and (R2) is highest with weak Async models.

It follows trivially that $\mathrm{C}(\langle p.X \rangle)$, the highest achievable coverage of $\langle p.X \rangle$, decreases when moving from weak Async models to the Sync model. Hence, the following conclusions can be drawn:

$\mathcal{MC}\mathbf{1}$: For any level $k$, $\mathrm{C}(\mathrm{SafeP}^k_{Async}) > \mathrm{C}(\mathrm{SafeP}^k_{ParSync})$ and $\mathrm{C}(\mathrm{LiveP}^k_{Async}) > \mathrm{C}(\mathrm{LiveP}^k_{ParSync})$.

*Best* $\mathcal{M}(\Delta)$: Whenever $X$ is not a real-time computing problem, choosing a (weak) Async model maximizes $\mathrm{C}(\langle p.X \rangle)$.

Let us illustrate $\mathcal{MC}1$ with SafeP. Many asynchronous algorithms that preserve SafeP, regardless of $\langle m.X \rangle$ being or not being violated, have been published—see, e.g., ACM PODC proceedings. It is also the case that asynchronous algorithms may preserve SafeP despite violations of "added" time-free semantics. Such algorithms are called "indulgent" in [8]. An example with UC is the $\diamond S$ rotating coordinator algorithm of [1], which preserves SafeP even if $\diamond S$ semantics are violated. Consequently, for some problems, SafeP hold true *under no conditions* with asynchronous algorithms, not the case with (partially) synchronous algorithms.

Then, the question: Why is it that ParSync models are sometimes considered when SafeP and LiveP only must be demonstrated?[3] Before proceeding with the examination of this question, let us recall what is implied with proving TimeP (stricto sensu, needed only if $X$ is a real-time computing problem).

## 2.2 Real-Time Computing Issues

### 2.2.1 Facts

Any (proven) solution to a real-time computing problem $X$ comprises (see scheduling theory):

- some solution $[\Delta]$ based upon scheduling algorithms, such as, e.g., HPF, EDF [20] or more complex schemes, traditionally designed in some ParSync model,
- schedulability analyses valid for pair $\{\langle X \rangle, [\Delta]\}$, under worst-case "load" and failure scenarios, conducted considering some ParSync model, necessarily,
- computable analytical expressions of time bounds T (matching those specified as per TimeP), such as, e.g., termination deadlines, as well as feasibility conditions—denoted FCs.

The ability to predict values of time bounds T before $S$ is turned on is an absolute requirement. Unfortunately, as is well-known, schedulability analyses are quite involved (NP-complete, most often), despite the fact that, in many instances, they rest on simplified models of internal processor architectures, internal system architectures, external and internal event arrivals ("loads"), processes, faulty behaviors. Moreover, in order to bring the inherent complexity of such analyses down to some tractable level, approximations are often resorted to. Although sound from a mathematical viewpoint, such approximations add to the inaccuracy due to considering simplified models of technology, of reality. Therefore, most often, bounds T have a modest coverage.

3. Again, theoretical works are not concerned with this question.

### 2.2.2 The "Late Binding" Principle

Note that it is necessary to consider some ParSync model only when the time has come to conduct schedulability analyses. Let $\mathcal{M}(S)$ be the implementation model retained for system $S$. That $\mathcal{M}(S)$ might be some ParSync model does not imply that $\mathcal{M}(\Delta)$ has to be a ParSync model as well. In fact, $\Delta$ may well be designed in some Async model, even when $X$ is a real-time computing problem.

The idea of deferring the consideration of some $\mathcal{M}(S)$ until after having devised and proved some design $\Delta$ in some $\mathcal{M}(\Delta)$ has been stated first in [16], echoed in [9] and [12], and detailed in [17], under the name of "design immersion" (in a computational model). This is equivalent to the concept of "late binding" (of a design to some computational model), a well-known concept in the field of programming languages.

Let $\mathrm{P}^k$ stand for a level $k$ property. According to this principle, bounds $\mathrm{T}^k_{Async}$ are established only after $\mathrm{SafeP}^k_{Async}$ and $\mathrm{LiveP}^k_{Async}$ have been proven, which is done without assuming any bounds $\mathrm{T}^j_{Async}$, $j < k$. This has definite advantages (see below) that are inaccessible to those approaches based upon "early binding" to a ParSync model, where one must first establish bounds $\mathrm{T}^k_{ParSync}$, prior to proving $\mathrm{SafeP}^k_{ParSync}$ and $\mathrm{LiveP}^k_{ParSync}$.

### 2.2.3 Conclusions

In any model, bounds $\mathrm{T}^k$ depend on $\mathrm{SafeP}^k$ and $\mathrm{LiveP}^k$. Indeed, $\mathrm{T}^k$ proofs rest on assuming some worst-case execution time for every process under consideration, *in addition to* assuming such properties as, e.g., absence of process deadlocks. Hence:

$\mathcal{TC}$: For any model, for any level $k$, $\mathrm{C}(\mathrm{T}^k) < \min\{\mathrm{C}(\mathrm{SafeP}^k), \mathrm{C}(\mathrm{LiveP}^k)\}$.

$\mathcal{TC}$ reinforces the observation made previously, i.e., C(TimeP) is modest compared to C(SafeP) or C(LiveP). From $\mathcal{MC}1$, we trivially derive:

$$\min\{\mathrm{C}(\mathrm{SafeP}^k_{ParSync}), \mathrm{C}(\mathrm{LiveP}^k_{ParSync})\} < \min\{\mathrm{C}(\mathrm{SafeP}^k_{Async}), \mathrm{C}(\mathrm{LiveP}^k_{Async})\}.$$

Compounding this with $\mathcal{TC}$ permits us to conclude:

$\mathcal{MC}\mathbf{2}$: For any level $k$, $\mathrm{C}(\mathrm{T}^k_{Async}) > \mathrm{C}(\mathrm{T}^k_{ParSync})$.[4]

Consequently, *Best* $\mathcal{M}(\Delta)$ can be generalized as follows:

*Best* $\mathcal{M}(\Delta)$: Choosing a (weak) Async model maximizes C(SafeP), C(LiveP), and C(TimeP).

The only way to increase $\mathrm{C}(\mathrm{T}^k_{ParSync})$ consists of adding "time margins" to bounds $\mathrm{T}^k_{ParSync}$. Hence:

$\mathcal{TCC}$: For any level $k$:

- either $\mathrm{T}^k_{ParSync} = \mathrm{T}^k_{Async}$ and $\mathrm{C}(\mathrm{T}^k_{ParSync}) < \mathrm{C}(\mathrm{T}^k_{Async})$,
- or $\mathrm{C}(\mathrm{T}^k_{ParSync}) = \mathrm{C}(\mathrm{T}^k_{Async})$ and $\mathrm{T}^k_{ParSync} > \mathrm{T}^k_{Async}$.

Obviously, "time margins" being added, partially synchronous solutions are necessarily less efficient and/or slower than asynchronous solutions for identical achieved coverage of timeliness properties.

Conclusions $\mathcal{MC}2$ and *Best* $\mathcal{M}(\Delta)$ are the foundations of the "late binding" principle, which underlies our approach.

4. $\mathrm{C}(\mathrm{T}^1)$ being the highest achievable coverage, $\mathrm{T}^1$ being proven out of *residual* timing assumptions.

### 2.3 Our Approach and Related Work

Under the "late binding" principle, Step 1 precedes Steps 2 and 3. Steps 2 and 3 may be concurrent.

Step 1) Given some $\langle X \rangle$, select as $\mathcal{M}(\Delta)$ the most appropriate weak Async model, specify $\Delta$ (selecting asynchronous algorithms only) as well as, in case $X$ is a real-time computing problem, time-free predicates stating activation conditions for schedulers. For example: "Service waiting queue W whenever W is nonempty" or "Make local scheduling decisions whenever distributed consensus has been reached." Prove SafeP and LiveP.

Step 2) Design a solution for implementing the time-free semantics of $\mathcal{M}(\Delta)$ out of residual (timing) assumptions, i.e., in a weak ParSync model. Provide FCs and time bounds T proper to that solution (e.g., failure detection latency).

Step 3) In case $X$ is a real-time computing problem, or in case one wants to predict some physical "performance" figures regarding $\Delta$, do "late binding" of $\Delta$ to some ParSync model so as to establish FCs and time bounds T for pair $\{\langle X \rangle, [\Delta]\}$.

Selection of a ParSync model for designing some solution $\Delta$ amounts to "early bind" $\Delta$ to some synchrony assumptions, which suffers from the drawbacks discussed above. Either timings are postulated (guessed) and then none of the requirements (R1), (R2) can be met. Or, assumed timings are demonstrated to hold as time bounds $\mathrm{T}_{ParSync}$ and then conclusions $\mathcal{MC}2$ and $\mathcal{TCC}$ apply. In order to prove that a ParSync model is correctly implemented, one has to face those difficulties raised with "high-level" schedulability analyses, *even if X is not a real-time computing problem*. Note that proving implementation correctness for a weak ParSync model involves "low-level" schedulability analyses.

The belief according to which contemplating some ParSync design model results into "better performance" is unfounded, as shown with conclusion $\mathcal{TCC}$, in accordance with observations made previously by many researchers.[5]

One may attempt to increase the coverage of ParSync models by resorting to the (classical) idea of "enforcing" timing assumptions at runtime—such attempts being (implicit) acknowledgments of the well-foundedness of $\mathcal{MC}1$ and $\mathcal{MC}2$. This idea, which underlies the Timed Asynchronous Distributed System (TA) model [2] and the Timely Computing Base (TCB) approach [24], is as follows: Design $\Delta$ is supplemented with measure-compare-and-kill algorithms that serve to 1) timestamp every significant state transition (e.g., message departures and arrivals) with current global or local time, 2) measure every actual delay value, for every delay variable that appears in $\Delta$ or *Proofs*, 3) compare every measured delay with its postulated bound. In case a "timing failure" occurs (a postulated bound is violated), that failure is transformed into a provoked abort, such as an omission failure (discard a late incoming message) or a stop failure (crash a processor that has received a late message).

However, the "enforcement" of timing assumptions is guaranteed only if every timing failure is always detected and reacted to in due time, which implies proving that processes which implement measure-compare-and-kill algorithms are always scheduled appropriately at every level where they are implemented, i.e., that their intrinsic bounds T are met. This implies conducting schedulability analyses accounting for the existence of these processes in addition to other "regular" processes. From the above, it follows that the best one can hope for is some coverage of the assertion "timing assumptions are enforced," which coverage is poor with "high" level timing assumptions. Consequently, this approach does help in increasing the likelihood of meeting (R1) and (R2) with ParSync models.

Note in passing that, regarding TimeP, a TCB does not provide better "guarantees" than can be expected from a distributed real-time operating system.

TA being a ParSync model and TCB resting on a ParSync model—in both cases, time bounds are assumed for ("high" level) interprocess communication delays—the conclusions arrived at in this section apply fully to TA and to TCB. Drawbacks that result from "early binding" to a ParSync model can be illustrated with some of the difficulties that arise with TA or TCB.

Given that timings are "guessed,"[6] it is impossible to bound the density of aborts provoked by measure-compare-and-kill algorithms. This is acknowledged in [2] and [24]. Consequently, algorithms that make use of variables that depend on postulated bounds for failures (e.g., bound $t$ for consensus or atomic broadcast) cannot be considered, the coverage of any a priori valuation of such variables being unknown. Moreover, systems may turn mute arbitrarily often, for arbitrarily long intervals, which is usually considered unacceptable—recall $\mathrm{cov}(\langle p.X \rangle)$.

Finally, the runtime penalty incurred with executing these measure-compare-and-kill algorithms more or less continuously cannot be ignored. Hence, FCs established with TA or TCB can only be worse than those established under the "late binding" approach, given that there is no need to supplement asynchronous solutions with measure-compare-and-kill algorithms.

Compounding this with conclusion $\mathcal{TCC}$, it follows trivially that TA or TCB lead to inefficient working solutions.

A detailed examination of these issues can be found in [18].

## 3 Architectural Model

We illustrate our approach with the Real-Time UC problem, denoted *RTUC*. Our $\mathcal{M}(\Delta)$ is the pure Async model augmented with Strong FDs [1], denoted S-Async. For the construction of Fast FDs, we also consider the pure Async model augmented with Perfect FDs, denoted P-Async. In such models, any solution to UC comprises a Distributed Uniform Consensus algorithm, denoted DC, and an FD construct. We derive our fast solutions to *RTUC* by augmenting Fast FD constructs with some algorithm, denoted $\Phi$. Pair $\{\mathrm{FD}, \Phi\}$ is referred to as a Failure Management construct, denoted FM.

We consider a finite set $\Pi$ of processors, interconnected by a network, referred to as Net. The nominal size of $\Pi$ is

5. Excerpt from [21]: "It is impossible or inefficient to implement the synchronous model in many types of distributed systems."

6. With real-time systems, the idea of correcting erroneous guesses after $S$ has been turned on does not make sense.

$n > 1$. The model of a processor is given in Fig. 1. The software/hardware architecture is modeled after a number of levels, such as, e.g., the application software level, the middleware level, the executive/operating system level, various communication protocol levels, the input/output (I/O) level. With existing systems, algorithms such as DC belong to the middleware level, exceptionally to a lower level. Messages handled by DC (resp., by FDs) are denoted DC-messages (resp., FD-messages). Messages other than FD-messages are referred to as ordinary messages. Let COM denote the level of communication protocols where one finds FD/FM constructs.

Let us model those waiting queues visited by outgoing messages, from the application level down to the COM level, as a single queue, denoted *outQ*, and those waiting queues visited by outgoing messages, from the COM level down to the I/O level, as a single queue, denoted *outq*. We define *inq* and *inQ* similarly. Let $\Gamma$ (resp., $\gamma$) stand for an upper bound on end-to-end delays for a DC-message (resp., an FD-message), measured at the COM level. Let $D$ stand for an upper bound on end-to-end delays for a DC-message, measured at the DC level. Let $W_{queue}$ denote the maximum sojourn time in *queue* for a DC-message. We have $D = W_{outQ} + \Gamma + W_{inQ}$.

For the sake of clarity, our analyses rest on assuming that a waiting queue is always serviced unless it is empty. Extension of our results to the more realistic case where a processor "does some work" before resuming servicing a nonempty waiting queue is reasonably straightforward.

A correct modeling of reality leads to considering that a processor servicing a message pending in a waiting queue is not preempted. Consequently, we define variables $w_{outQ}$, $w_{outq}$, $w_{inq}$, and $w_{inQ}$ as the blocking factors corresponding to the four waiting queues, respectively, that is the worst-case times for servicing a message pending in each of these waiting queues.[7] Similarly, we define $\delta_m$ as the blocking factor with Net, i.e., $\delta_m$ is the exact time needed for transmitting the longest ordinary message over the physical link between a processor and Net.

Assuming Net is idle and every visited waiting queue empty, a message is transmitted in some finite time, which is the lower bound of its processing and transmission delays. Let $\gamma_0$ be a lower bound of COM level end-to-end delays for an FD-message.

Whenever the transmission of a message first in *outq* is attempted, Net contention may occur. Such contention phenomena are handled via a multiaccess protocol (the distributed medium case) or a scheduling algorithm (internal to a Net node), which determines the ordering of messages successfully inserted in Net by competing I/O handlers. Let $\Psi$ stand for the worst-case time it takes to fully resolve Net contention arising between $n$ processors, each attempting to transmit an ordinary message ranked first in *outq*. $\Psi$ includes delays involved with local transmissions, between a processor and Net.

Whenever messages arrive "fast enough" at some processor, possibly from different input links, waiting queues *inq* and *inQ* build up. Consider some ordinary message *msg* waiting in *inq*. Let $k$ be the maximum

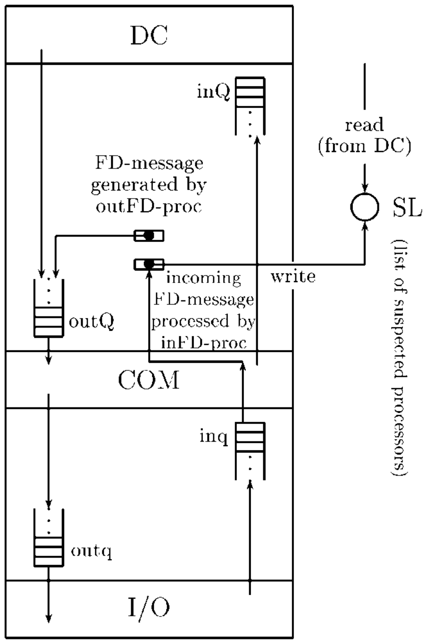

Fig. 1. Architectural model of a processor.

number of ordinary messages, generated by some processor, that may be serviced prior to *msg*. Hence, the worst-case rank of *msg* in *inq* is $m = kn$ and the worst-case time it takes for servicing *msg* is equal to $m\, w_{inq}$. Given that many processes (e.g., application-level, middleware-level) may generate messages having priorities higher or deadlines shorter than those assigned to *msg*, $k$ may be high. Let $\Theta$ stand for the worst-case time needed for transmitting a DC-message across Net, measured at the I/O level ($\Theta$ includes time of delivery into *inq*). For an ordinary message ranked $r$th in *outq* and $m$th in *inq*, let us write $\Gamma'(r, m) = r\,\Psi + \Theta + W_{inq}(m)$, with $W_{inq}(m) = m\, w_{inq}$.

Level COM is in charge of translating the destination field of every outgoing message into one or many physical addresses of processors. DC-messages and FD-messages carry $\mathcal{G}$ in their destination field. These messages are broadcast within some group of processors, denoted G, logical address $\mathcal{G}$. Throughout this paper, we consider that G is set $\Pi$.

Whenever appropriate, levels communicate through shared data structures and signaling. In accordance with [1], this is the case for DC and local FD, in which FD updates a list denoted $SL$, to be read by DC. At any time, $SL(p)$ contains names of those processors that $p$'s FD suspects (rightly, erroneously) of having failed. An FD module consists of two processes, one denoted inFD-proc, which maintains local list $SL$, another one denoted outFD-proc, which broadcasts FD-messages at regular intervals so as to "prove" that its processor has not failed. Whenever outFD-proc is run, it deposits an FD-message in *outQ*. Whenever an FD-message in *inq* is serviced at level COM, inFD-proc is signaled.

The body of an FD-message is empty. With algorithm *FastUC*, an FM-message carries some short data item in its body field, which data is invariant for a given processor and

7. Blocking factor $w_{queue}$ is included in $W_{queue}$.

for a given run of DC (see Section 7). In order to implement the reliable Net assumption w.r.t. FD-messages and FM-messages, retransmission of a previously transmitted but unacknowledged message—which is what TCP provides—would be a useless and costly luxury. Hence, UDP or any low level datagram protocol is appropriate for transmitting FD-messages and FM-messages.[8]

## 4 The Real-Time Uniform Consensus Problem

Every application-level service—denoted $Sv$—that needs UC is assigned some system-wide unique identifier, denoted $id_{Sv}$. Each time a process invokes UC on behalf of $Sv$, it provides a system-wide unique identifier for that invocation, denoted ID, such as, e.g., $id_{Sv}$, concatenated with some local sequencer value or with that sequencer value assigned to the last invocation of UC completed on behalf of $Sv$. An invocation of UC triggers the execution of algorithm DC.

Due to space limitations, we omit a detailed presentation of $\langle m.RTUC\rangle$. For our purposes, it suffices to know that, in addition to the architectural model given in Section 3, we consider those assumptions originally set for defining UC, i.e., 1) each time a process participates in a run of DC, it proposes some initial value, 2) processors may fail by stopping (correct behavior or permanent silence), 3) Net is reliable (Net does not partition, messages are not lost). Variable $t$ stands for an upper bound on the number of processor failures that may be experienced during any run of DC ($0 < t < n$). A process mapped onto a processor that fails is called an incorrect process (correct otherwise).

$\langle p.RTUC\rangle$: (For every DC run, from time 0)

- SafeP:
  - Uniform integrity: Every process (correct or incorrect) decides at most once.
  - Uniform agreement: No two processes (correct or incorrect) decide differently.
  - Uniform validity: If a process decides value $v$, then $v$ was proposed initially by some process (correct or incorrect).
- TimeP: If a process decides, it does so within at most $L$ time units after having invoked UC.[9]

In $\langle RTUC\rangle$, TimeP replaces LiveP—every correct process eventually decides some value—that appears in classical $\langle UC\rangle$ [1].

S-Async is our design model. Note that Eventually Strong FDs ($\diamond\mathcal{S}$) cannot be considered, given that TimeP should hold from time 0.

Following our approach, first, we ensure SafeP by selecting some convenient asynchronous DC algorithm, such as, e.g., the Rotating Coordinator algorithm [1] or a sequential algorithm originally introduced under the name of *Simple S* in [13], generalized in [22]. In the sequel, this algorithm is referred to as $Seq$ (see $MiniSeq$ in Section 7). Second, we show how to implement Strong FDs. Third, we prove TimeP. For our purposes, it is not necessary to address those schedulability issues raised with expressing $L$. In addition to requiring a complete presentation of $\langle m.RTUC\rangle$, this would be out of the scope of this paper. We simply need to show how to minimize $Z(DC)$, the worst-case termination time of selected algorithm DC so as to obtain the best achievable FCs—those closest to necessary and sufficient FCs. Fast FDs are necessary to attain that goal.

## 5 How to Construct Fast Failure Detectors

TrustFD is the problem of how to correctly instantiate the semantics of Perfect or Strong FDs in a (weak) ParSync model. Only some low-level modules, sitting at the COM level and below, are concerned with the construction of FDs.

An FD is Strong (resp., Perfect) if it satisfies strong completeness and weak (resp., strong) accuracy. Weak accuracy states that some correct processor is never suspected. Let $s$ ($s > 0$) be the smallest number of "stable" processors, a "stable" processor never being suspected unless it fails. Strong accuracy states that no processor is suspected before it fails. Strong completeness states that, eventually, every processor that fails is permanently suspected by every correct processor [1]. In fact, we are interested in strengthening strong completeness, as follows: Every processor that fails is permanently suspected by every correct processor *in some bounded finite time*. Let $d$ stand for the worst-case time needed to detect a processor failure, counted from the time of actual failure occurrence. We want such semantics to hold true from time 0. It turns out that this problem can be specified as the following COM level generic distributed message scheduling problem.

### 5.1 The TrustFD Problem

$\langle m.TrustFD\rangle$:

- Set $\Pi$ of $n$ processors, acting as sources of messages, exchanged via some network Net. There are two types of messages: FD-messages and ordinary messages (e.g., application, system, DC-messages).
- An ordinary message is initially deposited (by DC) in $outQ$, moved to $outq$ after being serviced in $outQ$, transmitted across Net after being serviced in $outq$, deposited in $inq$, then moved to $inQ$ after being serviced in $inq$, and delivered to DC after being serviced in $inQ$.
- An FD-message is initially deposited (by outFD-proc) in $outQ$, moved to $outq$ after being serviced in $outQ$, transmitted across Net after being serviced in $outq$, deposited in $inq$, then delivered to inFD-proc after being serviced in $inq$.
- Message arrival models: periodic generation of FD-messages (by every outFD-proc), period $\tau$; arbitrary arrivals for ordinary messages.
- Every processor is equipped with a "good" clock (see further). Every inFD-proc is equipped with a number of timers. Processor $j$'s inFD-proc uses a

8. In systems built out of standard protocol stacks, COM can be the UDP or TCP level. In systems built out of customized hardware/firmware, COM can be the data link level, the case with, e.g., spaceborne systems or defense systems.

9. $L$ includes those latencies due to the fact that a processor schedules and runs processes other than those which invoke UC, as well as services other than DC.

timer, denoted $T_{i,j}$, to monitor processor $i$. More or less periodically, every timer is reassigned some finite value. If $T_{i,j}$ awakes, inFD-proc($j$) declares "$i$ suspected" by adding $i$ to $SL(j)$. Let $V_{i,j}$ be the infinite set of consecutive values assigned to $T_{i,j}$.

- Processors may fail by stopping (up to $t$), Net is reliable.

$\langle p.TrustFD \rangle$:

- TimeP: $\forall i$, $\forall j$, time values in set $V_{i,j}$ are such that:
  - strong completeness and weak accuracy hold from time 0 (Strong FDs),
  - strong completeness and strong accuracy hold from time 0 (Perfect FDs).

Given TimeP, there must be an upper bound on the time interval that may elapse between any two consecutive generations of an FD-message by any given processor. This is encapsulated in the assumption that FD-messages are generated periodically.[10]

"Good" clocks are clocks that have a drift which is negligible relative to some physical time referential. Any two "good" clocks measure the actual duration of any given "short" time interval identically. We do not need synchronized clocks. Note that timers cannot be set to values that "look good" (offline), or that are "tuned appropriately" while a system is running. One must demonstrate, prior to fielding a system, that timers will be set to online computed values such that TimeP holds.

Some processor and Net capacity is used up by the processing and the transmission of FD-messages. Let $\rho$ be the worst-case overhead induced by FD-messages.

TimeP is proven if and only if one establishes analytical (computable) expressions of $d$ and $\gamma$, which amounts to solving a distributed message scheduling problem.

## 5.2 The FastFD Problem and How to Solve It

The *FastFD* problem is *TrustFD*, with $\langle p.TrustFD \rangle$ strengthened as follows:

$\forall i$, $\forall j$, values in set $V_{i,j}$ are lower bounds, for any given $\rho$.

*FastFD* is solved by showing how to establish a *tight* upper bound $d$ for some given $\rho$, which implies first showing how to establish a *tight* upper bound $\gamma$. Note that previous results related to FD implementation issues (e.g., [1], [15]) are not directed at solving *FastFD*.

### *5.2.1 How to Establish Tight $\gamma$*

Under worst-case processor and Net "loads," waiting queues build up and Net contention arises for transmitting concurrent FD-messages and ordinary messages on the one hand, concurrent FD-messages on the other hand. Fast failure detection is achievable only if upper bounds for FD-messages' sojourn times in waiting queues and Net nodes are optimal, the case whenever FD-messages are serviced prior to ordinary messages. This can be enforced by resorting to classical priority-driven, or deadline-driven, scheduling algorithms that implement the well-known head-of-the-line policy. Consequently, we retain the following $\mathcal{SW}$ algorithm:

$\mathcal{SW}$: In every visited waiting queue, an FD-message is always deposited ahead of ordinary messages and serviced prior to any ordinary message.

In order to resolve interprocessor competition optimally, processors may be assigned priorities or messages may be assigned different relative deadlines. Priorities or deadlines being fixed, they define a total order over any set of FD-messages whenever contention develops. Any such assignment is equivalent to assigning indices $1, \ldots, n$ over set $\Pi$, one index per processor. Moreover, whenever possible, preemption (of a broadcast medium, of a Net node) should be exercised, to the benefit of FD-messages, for it is known that preemption may be needed to achieve optimality. Therefore, we retain the following $\mathcal{SN}$ algorithm:

$\mathcal{SN}$: Net resources are allocated to FD-messages, prior to ordinary messages; in case of interprocessor competition for transmitting FD-messages, FD-messages are serviced in increasing index order.[11]

Let $\psi(x)$ stand for the worst-case time it takes for $x$ processors to preempt Net locally and to fully resolve Net contention involving $x$ FD-messages and $\psi'(x')$ stand for the worst-case time it takes for a processor to fully service a set of $x$ incoming FD-messages, both measured at the COM level ($x'$ is a function of $x$). Let $\nu$ be the smallest FD-message interarrival delay. Bound $x'$ is the maximum number of FD-messages (out of $x$) that are not serviced at the time the last incoming FD-message is deposited into $inq$. Given $\mathcal{SW}$, $x' = 1$ if $\nu \geq w_{inq}$, $x' = \lceil x\,(1 - \nu w_{inq}) \rceil$ if $\nu < w_{inq}$, and $\psi'(x') = x'\,w_{inq}$.

Bound $\psi(x)$ is determined by algorithms $\mathcal{SW}$ and $\mathcal{SN}$ and bound $\psi'(x')$ is determined by algorithm $\mathcal{SW}$. Hence, $\psi(x)$ and $\psi'(x')$ are tight [14], [20], [23]. Let $\theta$ stand for the worst-case time needed for transmitting an FD-message across Net, measured at the I/O level, including the time needed for delivery into $inq$. A tight bound $\theta$ can be computed, considering that optimal schedulers (proper to Net) service FD-messages prior to DC-messages [5], [25].

Consequently, for an FD-message generated by that processor assigned index $x$, tight bound $\gamma(x)$ is as follows:

$$\gamma(x) = w_{outQ} + w_{outq} + \delta_m + \psi(x) + \theta + x'\,w_{inq}.$$

By choosing $x := s$, processors assigned indices $1, \ldots, x$ are stable processors. Hence, one solves $FastFD$ in the S-Async (resp., P-Async) model by establishing an analytical expression of $\gamma(s)$ (resp., $\gamma(n)$). Worst-case FD overhead is $\rho = (\psi(x) + x\,w_{inq})/\tau$. Given that $\rho$ is set to some imposed (acceptable) value a priori, a lower bound for $\tau$ is $\tau = \psi(*)/\rho$, where $\psi(*) = \psi(x) + x\,w_{inq}$. Typically, $\rho \ll 1$. Note that $\rho$ may be smaller in fact in the case of real-time systems that make use of synchronized clocks. Receiving a clock synchronization message issued by some processor $p$ is equivalent to receiving an FD-message from $p$.

Given that $\psi(x)$ is tight, the lower bound of $\tau$ is tight as well for any given $\rho$. In order to construct Fast FDs, $\tau$ should be set equal to its lower bound.

10. It is likely that assuming no such bound as $\tau$, directly or indirectly, would lead to an impossibility. Stricto sensu, $\tau$ is a partial solution to a time-free specification of *TrustFD* (no timers, no "good" clocks assumption, no $\tau$), not presented in this paper.

11. A nonoptimal rule consists of using process names as indices.

#### 5.2.2 How to Establish Tight $d$

This issue is solved by showing that values used by FD timers are tight. Our algorithmic solution—denoted *RTV*, for Reset Timer Value—performs online calculations of those consecutive, time-dependent, values assigned to timers, in chronological order.

Every processor in $\Pi$ runs algorithm *RTV*. Let $v_{i,j}(k)$ be the value assigned to $T_{i,j}$ at the $k$th iteration of *RTV*. For the sake of simplicity, let us assume $\tau > \gamma(x) - \gamma_0$. If $\tau \leq \gamma(x) - \gamma_0$, then every FD-message must carry a sequence number, incremented at every round of FD-message broadcast by the sending out-FD proc, and sequence numbers must be part of algorithm *RTV*.

Consider any two processors $p$ and $q$. When $q$'s inFD-proc receives the first FD-message sent by $p$, $T_{p,q}$ is set to value $v_{p,q}(1) = \tau + \gamma - \gamma_0$. Implicitly, this is equivalent to assuming that this first message has traveled in $\gamma_0$ time units exactly. This may lead to computing too high a value for $v_{p,q}(1)$, but any other assumption may be invalid. Upon the arrival of the $k$th FD-message generated by $p$, $q$'s inFD-proc computes $v_{p,q}(k)$ as follows:

- Nonlearning *RTV*:

$$v_{p,q}(k) = \text{current value of } T_{p,q} + \tau.$$

- Learning *RTV*:

$$v_{p,q}(k) = \min\{\text{current value of } T_{p,q}, \gamma - \gamma_0\} + \tau.$$

Optimal timer values are obtained with the latter scheme.[12] The chronologically ordered $v_{p,q}(.)$ are those consecutive values that constitute set $V_{p,q}$. Rather than traveling in $\gamma_0$ time units, a first FD-message may travel in $\gamma$ time units. Therefore: $d = \tau + 2\gamma - \gamma_0$.

The "good clocks" hypothesis amounts to assuming that the physical drift of a correct clock is negligible over intervals at most equal to $d$. For any given $\rho$, $\tau$ and $d$ in the S-Async model are smaller than in the P-Async model. Given that $\tau$ and $\gamma$ are tight, $d$ is tight as well.

Furthermore, C($\gamma$) and C($d$) should be high, given that our construction of Fast FDs rests on proving "low" level time bounds, out of "low" level timing assumptions. The lower the level of COM is, the weaker the ParSync model considered for constructing Fast FDs is and the higher C($\gamma$) and C($d$) are (see Section 2).

One may consider addressing the $TrustFD$ problem or the $FastFD$ problem by providing oneself with two separate networks, one being exclusively reserved for "selected" processors and/or FD-messages. At first sight, this approach looks simpler, albeit more costly. However, one must still address those scheduling issues examined in this paper, namely, how to service $x$ competing sources that transmit FD-messages (as well as DC-messages and other messages, possibly) so as to establish (tight) bounds $\gamma$, $\tau$, and $d$.

12. Performance analyses presented in the sequel are conducted for worst cases, i.e., assuming a nonlearning *RTV*.

### 5.3 Conclusions

#### 5.3.1 $D$ versus $d$

Trivially, given algorithms $\mathcal{SW}$, $\mathcal{SN}$, and *RTV*, $\gamma \ll \Gamma'(r,m)$, hence $d < D$. In real systems, under worst-case conditions, i.e., when waiting queues build up at various levels of abstraction or implementation, one may even have $d \ll D$.

The formula given for $\Gamma'(r,m)$ in Section 3 does not include delays due to FD-messages. Consider levels up to level COM. Every $\tau$, up to $\psi(*)$ time units are used up (end-to-end) by FD-messages. Accounting for FDs, the actual bound is $\Gamma(r,m) = \Gamma'(r,m) + \lceil\frac{\Gamma'(r,m)}{\tau-\psi(*)}\rceil\psi(*)$. Knowing that

$$\Gamma'(r,m)(1 + \frac{\psi(*)}{\tau - \psi(*)}) \leq \Gamma(r,m)$$
$$< \Gamma'(r,m)(1 + \frac{\psi(*)}{\tau - \psi(*)}) + \psi(*)$$

and recalling that $\psi(*) \ll \Gamma'(r,m)$, we can write:

$$\Gamma(r,m) = \Gamma'(r,m)/(1-\rho) \text{ and } D = W_{outQ} + \Gamma(r,m) + W_{inQ}.$$

Recall that our analyses rest on assuming that a waiting queue is always serviced unless it is empty, which favors $D$ to a greater extent than $d$. Numerical illustrations are given in Section 6.

#### 5.3.2 How to Match the $t+1$ Rounds Lower Bound in the S-Async Model

Ranking of processors according to their indices is decided upon a priori. Under worst-case conditions, FD-messages generated by stable processors are always transmitted prior to FD-messages generated by processors assigned indices $s+1,\ldots,n$, referred to as "unstable." Given algorithm *RTV*, a correct unstable processor may be erroneously suspected under worst-case conditions. Therefore, it is useless to let an unstable processor send out FD-messages (only its inFD-proc should be active), hence the idea of splitting set $\Pi$ into two groups, one that comprises stable processors only, which group is called the $active$ group, another one comprising unstable processors only, called the $silent$ group.

DC-messages and FD-messages are broadcast within group G (defined in Section 3) by stable processors only. Processors in the silent group receive DC-messages generated by the active group, but they do not broadcast any. The silent group implements pure asynchronous semantics. Given that at most $t$ processors may fail, it suffices to set $s$ to value $t+1$ for implementing the semantics of the S-Async model in set $\Pi$. Actually, an active group implements the semantics of Perfect FDs.

The worst-case lower bound for round-based algorithms that solve UC in the S-Async model is $n-s+1$ rounds. This bound may be significantly greater than $t+1$. With our construction of Strong FDs, there is at least one stable and correct processor in the active group. Hence, if we run algorithm $Seq$ in the active group, it follows that $Seq$ terminates in exactly $t+1$ rounds.

This suffices to demonstrate that state-of-the-art asynchronous solutions can be as efficient as state-of-the-art synchronous solutions regarding (logical) time complexity.

However, with real-time systems, physical time bounds matter and logical time complexity should not be equated to physical time bounds. An algorithm that terminates in $R$ rounds may be slower than another algorithm that terminates in $R'$ rounds, $R' > R$. This is demonstrated in [18], where synchronous consensus algorithms—denoted $\mathcal{F}$—that terminate in $t+1$ rounds, are compared to *FairSeq*, a "fair" extension of *Seq*, that terminates in $t+2$ rounds. Indeed, whenever $n > t+1$, processor(s) in the silent group cannot propose some initial value. In order to alleviate unfairness, one extra round is added to *Seq*, during which every processor in the silent group broadcasts its initial value within the active group. We establish under which condition *FairSeq*'s worst-case termination time is smaller than $\mathcal{F}$'s worst-case termination time. We show that this condition is met most often with real systems.

This result demonstrates that state-of-the-art asynchronous solutions may exhibit timeliness properties and FCs that are better than those achieved with state-of-the-art synchronous solutions.

Some synchronous version of *FairSeq* can be imagined. Our demonstration still holds, for the reasons given in conclusion *TCC* (Section 2).

## 6 Illustration with Ethernets

Let us illustrate these generic results with Ethernet-like networks. In this section, COM is the ISO/OSI data link level. With Ethernets, $\theta = 0$ ($\psi(x)$ includes local physical transmission delays and there is no additional transmission delay). Hence, $\gamma(x) = w_{outQ} + w_{outq} + \delta_m + \psi(x) + x'\, w_{inq}$.

Being concerned with real-time systems, we must consider a deterministic variant of the original Ethernet CSMA-CD protocol. This variant, called CSMA/DCR (Carrier Sense Multi Access/Deterministic Collision Resolution), which is based on deterministic balanced *m*-ary tree searches [19], has been implemented in COTS products.[13]

### 6.1 CSMA/DCR

Broadcast media are physically characterized by a channel slot time, denoted $\sigma$. Sources of messages are processors. Channel sharing between sources works à la CSMA-CD whenever there is no unresolved collision pending. When a collision is detected (and there is no previous collision pending), sources initiate a deterministic balanced *m*-ary tree search collectively. To this end, every source is assigned some unique index. For this illustration, it suffices to consider exactly one index per source. A tree search proceeds from left to right, searching for subtrees that either are empty or contain exactly one active leaf. A leaf is active if its index is that of a source which has a message pending. Obviously, during a tree search, a message submitted by a source assigned index $i$ is transmitted prior to messages submitted by sources assigned indices greater than $i$. A tree search is time bounded, which permits computing $\psi(x)$.

Consider $x$ sources, each attempting to transmit a pending message (rank 1 in *outq*). Let $\Sigma(x)$ be the time needed to physically transmit these $x$ messages locally, in the absence of contention. Consider now that these $x$ sources "collide." Let $\xi_x^l$ be the maximum number of steps needed to search $x$ leaves in an $m$-ary tree of $l$ leaves. In [10] and [11], one shows:

$$\xi_x^l = \frac{m^{\left\lceil log_m\left(m\left\lfloor \frac{x}{2}\right\rfloor\right)\right\rceil} - 1}{m-1} + m\left\lfloor\frac{x}{2}\right\rfloor \left\lfloor log_m\left(\frac{l}{m\left\lfloor\frac{x}{2}\right\rfloor}\right)\right\rfloor - \left(x - m\left\lfloor\frac{x}{2}\right\rfloor\right),\ x \in \{2,\ldots,l\}.$$

This formula applies for any assignment of $x$ indices over $l$ sources (general assignment). A tighter bound—denoted $\zeta_x^l$—holds when those $x$ sources that collide own indices $1, 2, \ldots x$ (optimal assignment). The closed-form expression of $\zeta$ is as follows:

$$\zeta_x^l = 1 + m\left(log_m(l) + \sum_{i=1}^{log_m(l)} \left\lfloor\frac{x-2}{m^i}\right\rfloor\right) - x,\ x \in \{2,\ldots,l\}.$$

Therefore, depending whether index assignment is general or optimal, the worst-case delay involved with resolving a collision fully is $\Sigma(x) + \xi_x^l\,\sigma$ or $\Sigma(x) + \zeta_x^l\,\sigma$, respectively.

CSMA/DCR has been designed to be fault-tolerant. This protocol may be defeated whenever sources get out of synchrony, which is revealed by detecting a collision on some tree leaf. Whenever this occurs, a channel jamming sequence of duration at least equal to $log_m(l)\,\sigma$ is generated by the sources. Message transmissions are resumed when the channel returns to idle. We now show how these CSMA/DCR features can be applied to our generic solution.

### 6.2 Bounds $\gamma$ and $d$

Under worst-case conditions, the channel is jammed during $log_m(l)\,\sigma$, to indicate that an on-going tree search performed for ordinary messages must be stopped, in order to transmit FD-messages. The channel is preempted without aborting any ordinary message (the blocking factor is $\delta_m$). After such a sequence has been generated, only FD-messages are transmitted, if so desired (our choice, for this illustration). Processors get synchronized via the jamming sequence and FD-messages are transmitted during the same tree search. Tree search for ordinary messages is resumed from its preemption state when the channel returns to idle. Given that an FD-message is empty, its physical transmission delay is that of a message of minimum duration, i.e., slot time $\sigma$. Therefore, $\Sigma(x) = x\,\sigma$. Depending on whether index assignment is general or optimal, $\psi(x)$ writes $\psi^+(x)$ or $\psi^-(x)$, respectively, as follows:

$$\psi^+(x) = \left(log_m(l) + x + \xi_x^l\right)\sigma \quad \text{and}$$
$$\psi^-(x) = \left(log_m(l) + x + \zeta_x^l\right)\sigma.$$

The smallest FD-message interarrival delay $\nu$ (Section 5) is $\sigma$. Hence, $x' = \lceil x(1 - \sigma w_{inq})\rceil$ if $\sigma < w_{inq}$, $x' = 1$ if $\sigma \geq w_{inq}$. Tight bound $\gamma$ for the $x$th FD-message is:

$$\gamma^+(x) = w_{outQ} + w_{outq} + \delta_m + \psi^+(x) + x'\, w_{inq} \quad \text{or}$$
$$\gamma^-(x) = w_{outQ} + w_{outq} + \delta_m + \psi^-(x) + x'\, w_{inq}.$$

13. CSMA/DCR networks are used in, e.g., the launchpad of Ariane, the European satellite launcher, defense shipborne systems, and factories.

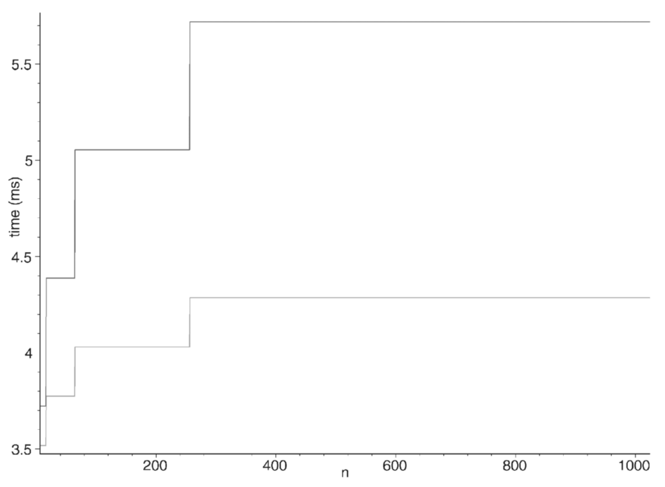

Fig. 2. $\gamma_A^+$ (upper) and $\gamma_A^-$ (lower), with *l*-leaf balanced quaternary trees, $16 \le l \le 1{,}024$, $t+1 \le l \le 1{,}024$, $t = 5$.

The shortest FD-message duration is $\gamma_0 = w_{outQ} + w_{outq} + \sigma + w_{inq}$. Recall that tight bound $d$ is $d = \tau + 2\,\gamma - \gamma_0$.

## 6.3 Numerical Examples

Let us consider 10 MBit/s Ethernets.[14] Let us consider that the size of the longest ordinary message (I/O level framing) is 10,000 bits, i.e., $\delta_m = 1\,ms$. Let us pick up $250\,\mu s$ for each of the blocking factors $w_{outQ}$, $w_{outq}$, and $w_{inq}$. Hence, $\gamma_0 = 801.2\,\mu s$ and $\gamma(x) = 1.5 + \psi(x) + 0.25\,x'$ (in $ms$), with $x' = \lceil 0.7952\,x \rceil$. Results shown below are rounded up to a precision of 10 $\mu s$. Let us consider quaternary trees ($m = 4$) and choose $t = 5$.

Let FFDs stand for Fast FDs. With Strong FFDs, $x = s = 6$, while $x = n$ with Perfect FFDs. Subscript A (resp., PA) stands for the S-Async (resp., P-Async) model. We get the following results (see also Fig. 2).

- Case #1: $n = l = 16$
  - Perfect FFDs: $x' = 13$, $\psi_{PA} = 1.18\,\mu s$, hence $\gamma_{PA} = 5.93\,ms$.
  - Strong FFDs: $x' = 5$, $\psi_A^+ = 0.97\,ms$, and $\psi_A^- = 0.77\,ms$. Hence, $\gamma_A^+ = 3.72\,ms$ and $\gamma_A^- = 3.52\,ms$.
- Case #2: $n = l = 1{,}024$
  - Perfect FFDs: $x' = 815$, $\psi_{PA} = 70.14\,ms$, hence $\gamma_{PA} = 275.39\,ms$.
  - Strong FFDs: $x' = 5$, $\psi_A^+ = 2.97\,ms$, and $\psi_A^- = 1.54\,ms$. Hence, $\gamma_A^+ = 5.72\,ms$ and $\gamma_A^- = 4.29\,ms$.

For a fair comparison between Perfect and Strong FFDs, we should consider the same worst-case overhead $\rho$ in both cases. Let us pick up $\rho = \frac{\psi(x) + x\,w_{inq}}{\tau} = 5\%$, that is $\tau = 20\,(\psi(x) + 0.25\,x)$ (in $ms$).

- Case #1: $n = 16$
  - Perfect FFDs: $\tau_{PA} = 103.55\,ms$ and $d_{PA} = 114.61\,ms$.
  - Strong FFDs: $\tau_A^+ = 49.46\,ms$ and $\tau_A^- = 45.36\,ms$. Hence, $d_A^+ = 56.10\,ms$ and $d_A^- = 51.59\,ms$.
- Case #2: $n = 1{,}024$
  - Perfect FFDs: $\tau_{PA} = 6.52288\,s$ and $d_{PA} = 7.07287\,s$.
  - Strong FFDs: $\tau_A^+ = 89.39\,ms$ and $\tau_A^- = 60.72\,ms$. Hence, $d_A^+ = 100.03\,ms$ and $d_A^- = 68.49\,ms$.

S-Async being our design model, let us focus on performance figures achieved with Strong FFDs. Consider $\Gamma(r, m)$, pick up $k = 5$ (i.e., $m = 5\,n$), $r = 5$, and the same numerical values as above. With Ethernets, $\Theta = 0$ (same reasons as for $\theta = 0$). With CSMA-DCR and quaternary tree search, $\Psi(16) = 5 \times 0.0512 + 16 \times 1\,ms = 16.26\,ms$ and $\Psi(1024) = 341 \times 0.0512 + 1024 \times 1\,ms = 1.04146\,s$. Hence, $\Gamma = 106.61\,ms$ for $n = 16$ and $\Gamma = 6.82873\,s$ for $n = 1{,}024$. As $n$ ranges from 16 to 1,024, ratio $\Gamma/\gamma_A^+$ ranges approximately from 29 to 1,194 and ratio $\Gamma/\gamma_A^-$ ranges approximately from 30 to 1,593.

Let us pick up $W_{outQ} = W_{inQ} = 150\,ms$ (worst-case bounds). For $n = 6$, one finds $D = 406.61\,ms$. For $n = 1{,}024$, one finds $D = 7.12873\,s$. As $n$ ranges from 16 to 1,024, ratio $D/d_A^+$ ranges approximately from 7 to 71 and ratio $D/d_A^-$ ranges approximately from 8 to 104.

We observe that a small $d_A$ is achievable at the expense of some marginal processing and communication overhead. Ratio $D/d$ illustrates the merits of Strong Fast FDs.

Recall that our analyses rest on assuming that a waiting queue is always serviced unless it is empty. Consequently, numerical values computed for $d$ are more realistic than those computed for $\Gamma$ and $D$, which are optimistic values. Such results lead to the possibility of devising Fast UC solutions (see Section 7).

Similar results can be established, considering other types of networks (e.g., store-and-forward, irregular meshed networks) and other deterministic multiaccess or/and communication protocols, as well as faster links.

## 7 A Novel Solution for Fast Uniform Consensus

Let us now examine TimeP (see $\langle RTUC \rangle$). Our objective is to minimize $Z(DC)$, the worst-case time for achieving UC, DC being "immersed" in some ParSync model. Let us consider that delays involved with computations in a DC round are negligible compared to $D$. Hence, $D$ is the worst-case termination time of a round. A round may in fact terminate as soon as some processor failure is detected. Such a round is referred to as a mini-round. De facto, variable $d$ is a tight upper bound on mini-round durations. As for rounds, we consider that delays induced by computation steps in a mini-round are negligible, compared to $d$.

Fast Uniform Consensus derives from the idea of replacing rounds with mini-rounds, whenever feasible. Consider *Seq*. Under worst-case failure scenarios, $Z(Seq) = D + t\,d + \gamma$. However, worst-case values of $Z(Seq)$ are obtained with "lucky" runs (no failures), in which case $Z(Seq) = (t+1)\,D$.

14. According to the ISO/OSI standard, greatest values are 1,024 for $n$, $51.2\,\mu s$ for $\sigma$.

```
SRC
cobegin
    {if $k \leq s$ then /* $p_k$ is an active processor */
        DC-message($k$) $\longleftarrow$ $\{k, prop_k\}$
        Deposit DC-message($k$) into $outQ$ /* message eventually broadcast, unless $p_k$ fails */
    end if}
    {Record every delivered DC-message($\cdot$)
    Wait until (TC1) and (TC2) are met}
coend
    Decision($p_k$) $\longleftarrow$ $prop_{win}$

MiniSeq (for some given ID)
    $est_k \leftarrow \{k\}$
    for $r$ from 1 to $s$ do
        if $r = k$ then /* $p_k$ is the coordinator of round $r$ and $p_k$ is an active processor */
            FM-message($k$) $\longleftarrow$ $\{est_k\}$
            Deposit FM-message($k$) into $outQ$ /* FM-message($k$) is eventually broadcast within G,
                                                   unless $p_k$ fails */
        else /* $p_k$ is not the coordinator of round $r$ or $p_k$ is not an active processor */
            wait until (($p_r \in SL(k)$) $\vee$ ($est\_from_{p_r}$ is received from $p_r$))
            /* $est\_from_{p_r}$ is the content of delivered FM-message($r$) */
            if ($est\_from_{p_r}$ is received from $p_r$) then
                $est_k \leftarrow est\_from_{p_r}$
            end if
        end if
    end for
    $win(p_k) \leftarrow est_k$
```

Fig. 3. Algorithms SRC and MiniSeq for processor $p_k$, assigned index $k$, $k \in \{1, \ldots, n\}$.

With Fast FDs, it is possible to speed up the execution of DC algorithms, irrespective of failure occurrence patterns.[15] The solution presented in this paper rests on 1) "augmenting" a Fast FD construct with a failure management (FM) construct, 2) taking advantage of the inherent parallelism between DC and FD/FM. Actually, this approach is similar to that followed for constructing synchronized clocks, where "high" level processes are provided with a global time service (FM in our case), implemented at some "low" level.

## 7.1 Fast Uniform Consensus

Given that, with *RTUC*, worst-cases matter, let us present a non-early-deciding/stopping solution, referred to as *FastUC* (see Fig. 3). *FastUC* comprises 1) a DC algorithm, called SRC (Single Round Consensus), 2) an FM construct {Fast FD, *MiniSeq*}. Let us present *FastUC* informally.

### 7.1.1 SRC

SRC consists of having every active processor broadcast its DC-message, carrying some proposed value, denoted *prop*, tagged with some unique identifier ID (see Section 4), with $\mathcal{G}$ in its destination field. Whenever a DC-message is processed at the COM level, address $\mathcal{G}$ is decoded, in which case the local FM module is signaled, with ID as a parameter. Upon receiving that signal, the FM module starts running *MiniSeq*(ID), which results in having an FM-message deposited in *outQ*, behind the ID-matching DC-message.

Algorithms $\mathcal{SW}$ and $\mathcal{SN}$ (Section 5) are used to process FM-messages. However, the scheduling of FM-messages should obey the following ordering constraint:

Within *outq*, the scheduling of a DC-message and an FM-message carrying the same ID should preserve their initial relative ordering.

Therefore, the following local precedence ($\prec$) property holds for any run of *FastUC*:

($\mathcal{P}$): completion of a DC-message broadcast $\prec$ start of an FM-message broadcast.

Incoming DC-messages are recorded, until conditions (TC1) and (TC2) are met—see below. The message complexity of SRC is $(t+1)\,n$.

### 7.1.2 MiniSeq

*MiniSeq* is derived from *Seq*. With *Seq*, *value* is any type of proposition. With *MiniSeq*, *value* is pair $\{\text{k, ID}\}$, where k is a processor's index and ID is a unique identifier of some

15. Ongoing work shows how to achieve such bounds as $Z = D + t\,d + \gamma$, in synchronous models, even when no failures occur, using Fast FDs.

SRC run. In other words, for any given ID, consensus $values$ are processor indices (variable $est$).

$MiniSeq$ consists of having active processors take turns according to their respective indices. An active processor broadcasts its FM-message only once. FD-messages keep being generated while $MiniSeq$ is running. Processor $p_k$, assigned index $k$, monitors processors $p_j$, assigned indices $j$, $j \in \{1, \dots, k-1\}$. Processor $p_k$ broadcasts some value $est_k$ if and only if it is not waiting for some processor $p_j$. The wait for processor $p_j$ is over for $p_k$ whenever either $p_j$ appears on $SL(k)$ or $est\_from_{p_j}$ is received by $p_k$. Processors in the silent group perform the "wait until" part of $MiniSeq$. Every processor terminates with $win$ = most "recent" $est\_from_{p_i}$ received, $i$ being the highest index seen.

$MiniSeq$(ID) runs while matching ID-tagged DC-messages travel across Net, sojourn in $inq$, then in $inQ$, before being delivered at the DC level. When $MiniSeq$ terminates, one processor is elected as "the winner," denoted $win$. The content of its DC-message is denoted $prop_{win}$. SRC terminates whenever the following termination conditions (TC1) and (TC2) are met:

$win$ is known (TC1) and $prop_{win}$ has been delivered (TC2).

It may be useful to provide FM constructs with an "overdrive" option. If "overdrive" is off, FD-messages are broadcast every $\tau$. If "overdrive" is on, then, while $MiniSeq$ is run, FD-messages are broadcast at some period smaller than $\tau$. This permits finetuning runs of $MiniSeq$—so as to achieve UC in exactly $D$ (see further)—without having to incur any excessive permanent processing and communication overhead with FD-messages. Due to space limitations, the detailed specification of this option is omitted.

### 7.2 Proof Overview and Performance

Proofs that $Seq$—hence $MiniSeq$—solves UC in the S-Async model can be found in [22]. Let $i$ be the elected index ($win = i$). This implies that $p_i$'s FM module has broadcast, at least partially, an FM-message that carries $i$. Given $\mathcal{P}$, it follows that $prop_i$ has necessarily been fully broadcast by processor $p_i$ when SRC was run. This, in conjunction with the reliable Net assumption, suffices to demonstrate that $prop_i$ is eventually received by every correct processor. Hence, $FastUC$ solves UC.

Let us now examine $Z(FastUC)$, the worst-case termination time of $FastUC$. Bounds $\psi(x)$ and $\psi'(x')$ are valid for $x$ FD-messages. FM-messages are broadcast sequentially, one every $\tau$. Therefore, new bounds of interest are $\psi(x+1)$, $\psi'(x'') = x'' \, w_{inq}$, with $x'' = 1$ if $\nu \geq w_{inq}$, $x'' = \lceil (x+1)\,(1 - \frac{\nu}{w_{inq}}) \rceil$ if $\nu < w_{inq}$, $\rho'' = (\psi(x+1) + (x+1)\, w_{inq})/\tau$. Bounds $\gamma''$ and $d''$ are computed accordingly.

With most common multiaccess protocols or scheduling algorithms, if $\rho$ is acceptable, then so is $\rho''$.

The worst-case time for meeting (TC2) with SRC is $D = W_{outQ} + \Gamma(r,m) + W_{inQ}$ (see Section 5). Let us define $\Lambda$ as $\Theta + W_{inq} + W_{inQ}$ and let us write $\Lambda = \varphi\, D$, $0 < \varphi < 1$. Most often, with real systems and general networks, $\varphi \approx 1/2$. Of course, contrary to FD/FM related bounds (see above), bounds $\Lambda$ and $D$ are those computed ignoring the overhead due to FM-messages.

After relative time $W_{outQ} + W_{outq} = (1-\varphi)\,D$ at the latest, every correct processor runs $MiniSeq$, which then terminates in no more than $(t+1)\,\gamma''$ with "lucky runs" (no failures),[16] in no more than $t\,d'' + \gamma''$ (trivial) under worst-case failure scenarios, which is the worst-case time for meeting (TC1). Therefore:

$$Z(FastUC) = \max\{D; D - \Lambda + t\,d'' + \gamma''\}.$$

*FastUC* appears to be the first UC algorithm that ensures a worst-case termination time sublinear in $t\,D$. Optimality is achieved with *FastUC* under the following condition:

$(1\mathcal{R})$ $FastUC$ terminates in no more than 1 (SRC) round duration $\Longleftrightarrow t\,d'' + \gamma'' \leq \Lambda$.

Equivalently:

$$\tau \leq (\Lambda - \gamma'')/t - (2\,\gamma'' - \gamma_0) \quad \text{or} \quad t \leq \lfloor (\Lambda - \gamma'')/d'' \rfloor.$$

Due to space limitations, a detailed analysis of this constraint is omitted. In order to show that $(1\mathcal{R})$ can be met with real systems and common values of $t$, let us reuse the numerical illustrations given end of Section 6, computed for $n = 16$ and $s = 6$ ($t = 5$).

$D$ was found to be 406.61 $ms$ and $\varphi = 0.431$. Hence, $\Lambda = 175.25\,ms$. Note that we keep considering $\Theta = 0$, which is not representative of general networks and which does not help in meeting $(1\mathcal{R})$. Given that $\gamma''$ is not much higher than $\gamma$, let us pick up, for $\gamma''$, the average of those values computed for the general and the optimal index assignments, that is, $\gamma'' = 3.62\ ms$. Similarly, for $\tau$, we consider the average value, that is, $\tau = 47.41\,ms$. Consequently, $d'' = 53.85\,ms$. We can now conclude:

$$(1\mathcal{R}) \text{ is met if } t \leq 3.$$

Observe that the numerical values considered for $\gamma''$ and $\tau$ have been established for $t = 5$. In other words, these values are "pessimistic." It follows that the result $t \leq 3$ holds a fortiori.

This numerical illustration shows that $(1\mathcal{R})$ can be met with values of $t$ commonly considered. Whenever this is the case, $FastUC$ achieves UC in no more than $D$, which is the absolute lower bound. Hence, $FastUC$ is an optimal solution for $RTUC$.

That UC may be achieved at almost no extra "time cost" in some circumstances should be a result of interest to many designers of real-time distributed systems.

Any synchronous UC algorithm has a worst-case termination time at least equal to $2D$ (unless it would rest on Fast FDs). It follows that any such algorithm is easily outperformed by $FastUC$, yet another example of the particular relevance of FDs regarding real-time distributed systems.

## 8 Conclusions

We have examined the merits and drawbacks that result from adopting a particular computational model, from synchronous to asynchronous ones, for designing solutions

16. As can be seen, an early-deciding version of $MiniSeq$ would achieve impressively small termination times.

intended for real-time or non-real-time distributed fault-tolerant computing problems. The principle of "late binding" asynchronous solutions to some (partially) synchronous model has been presented. We have shown that this principle permits 1) circumventing the apparent contradiction between asynchrony and the need to consider some (partially) synchronous model for conducting schedulability analyses, 2) maximizing the coverage of demonstrated safety, liveness, and timeliness properties.

This general principle has been illustrated with the introduction of Fast Failure Detectors, a refinement of classical Unreliable Failure Detectors, as well as with the introduction of *FastUC*, a worst-case time optimal algorithm that solves the uniform consensus problem. These new results, as well as their extension to more general failure models (e.g., unreliable communications), may pave the way for new generations of real-time distributed systems that would use asynchronous algorithms designed to solve problems in fault-tolerant distributed computing.

Most problems that lie in the intersection of the Distributed Algorithms and the Real-Time Scheduling theories are open. New algorithmic solutions, new optimality results, will be established in the future. Hopefully, this paper is a contribution to advancing the state-of-the-art in this dual area.

## 9 Glossary

$n$: number of processors, $n > 1$.

$t$: maximum number of processor failures that may occur while running an instance of DC, $0 < t < n$.

$s$: smallest number of stable/active processors, $0 < s \leq n$.

$\gamma$: upper bound of COM level end-to-end delays for FD-messages.

$\gamma_0$: lower bound of COM level end-to-end delays for FD-messages.

$\rho$: maximum processing and communication overhead caused by FD-messages.

$\tau$: FD-message broadcasting period.

$d$: upper bound on processor failure detection latencies = upper bound on mini-round durations.

$\Gamma$: upper bound of COM level end-to-end delays for DC-messages.

$D$: upper bound of DC level end-to-end delays for DC-messages = upper bound on round durations.

$Z$: upper bound on execution delays of a DC algorithm.

## References

[1] T.D. Chandra and S. Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems," *J. ACM,* vol. 43, no. 2, pp. 225-267, Mar. 1996. (A preliminary version appeared in *Proc. 10th ACM Symp. Principles of Distributed Computing,* pp. 325-340, 1991 ).

[2] F. Cristian and C. Fetzer, "The Timed Asynchronous Distributed System Model," *IEEE Trans. Parallel and Distributed Systems,* vol. 10, no. 6, pp. 642-657, June 1999.

[3] D. Dolev, C. Dwork, and L. Stockmeyer, "On the Minimal Synchronism Needed for Distributed Consensus," *J. ACM,* vol. 34, no. 1, pp. 77-97, Jan. 1987.

[4] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the Presence of Partial Synchrony," *J. ACM,* vol. 35, no. 2, pp. 288-323, Apr. 1988.

[5] D. Ferrari and D.C. Verma, "A Scheme for Real-Time Channel Establishment in Wide-Area Networks," *IEEE J. Selected Areas in Comm.,* vol. 8, no. 3, pp. 368-379, Apr. 1990.

[6] M.J. Fischer and N.A. Lynch, "A Lower Bound for the Time to Assure Interactive Consistency," *Information Processing Letters,* vol. 14, pp. 183-186, June 1982.

[7] M.J. Fischer, N.A. Lynch, and M.S. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *J. ACM,* vol. 32, no. 2, pp. 374-382, Apr. 1985.

[8] R. Guerraoui, "Indulgent Algorithms," *Proc. 19th ACM Symp. Principles of Distributed Computing,* pp. 289-297, July 2000.

[9] R. Guerraoui and A. Schiper, "Consensus: the Big Misunderstanding," *Proc. Sixth IEEE Workshop Future Trends of Distributed Computing Systems,* pp. 183-188, Oct. 1997.

[10] J.-F. Hermant and G. Le Lann, "A Protocol and Correctness Proofs for Real-Time High-Performance Broadcast Networks," *Proc. IEEE Int'l Conf. Distributed Computing Systems,* pp. 360-369, May 1998.

[11] J.-F. Hermant, "Quelques Problèmes et Solutions en Ordonnancement Temps Réel pour Systèmes Répartis," PhD thesis, Paris-VI-Pierre-et-Marie-Curie Univ., Sept. 1999.

[12] M. Hurfin and M. Raynal, "Asynchronous Protocols to Meet Real-Time Constraints: Is It Really Sensible? How to Proceed?" *Proc. IEEE Int'l Symp. Object-Oriented Real-Time Distributed Computing,* pp. 290-297, Apr. 1998.

[13] Algorithm derived independently in 1997 by P. Jayanti and S. Toueg, and by B. Charron-Bost (S. Toueg, private comm., 1999).

[14] J.F. Kurose, M. Schwartz, and Y. Yemini, "Multiple-Access Protocols and Time-Constrained Communication," *ACM Computing Surveys,* vol. 16, no. 1, pp. 43-70, Mar. 1984.

[15] M. Larrea, S. Arévalo, and A. Fernández, "Efficient Algorithms to Implement Unreliable Failure Detectors in Partially Synchronous Systems," *Proc. 13th Int'l Symp. Distributed Computing,* pp. 34-48, Sept. 1999.

[16] G. Le Lann, "On Real-Time and Non Real-Time Distributed Computing," *Proc. Ninth Int'l Workshop Distributed Algorithms,* invited paper, *Lecture Notes in Computer Science,* vol. 972, pp. 51-70, Springer-Verlag, Sept. 1995.

[17] G. Le Lann, "Proof-Based System Engineering and Embedded Systems," *Proc. European School on Embedded Systems,* invited paper, *Lecture Notes in Computer Science,* vol. 1494, pp. 208-248, Springer-Verlag, Nov. 1996.

[18] G. Le Lann, "Is 'Asynchronous Real-Time' an Oxymoron?" *15th Int'l Symp. Distributed Computing,* invited talk, Oct. 2001, INRIA Research Report, to appear.

[19] G. Le Lann and P. Rolin, "Process and Device for the Transmission of Messages between Different Stations through a Local Distribution Network," US Patent Number 4,847,835, July 1989, French Patent Number 84-16957, Nov. 1984.

[20] C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *J. ACM,* vol. 20, no. 1, pp. 46-61, Jan. 1973.

[21] N.A. Lynch, *Distributed Algorithms.* Morgan Kaufmann, 1996.

[22] A. Mostefaoui and M. Raynal, "Consensus Based on Failure Detectors with a Perpetual Weak Accuracy Property," *Proc. IEEE Int'l Parallel and Distributed Processing Symp.,* pp. 514-519, May 2000.

[23] K. Tindell, A. Burns, and A.J. Wellings, "Analysis of Hard Real-Time Communications," *J. Real-Time Systems,* vol. 9, no. 1, pp. 147-171, Sept. 1995.

[24] P. Verissimo, A. Casimiro, and C. Fetzer, "The Timely Computing Base: Timely Actions in the Presence of Uncertain Timeliness," *Proc. IEEE Int'l Conf. Distributed Systems and Networks,* pp. 533-542, July 2000.

[25] H. Zhang, "Service Disciplines for Guaranteed Performance Service in Packet-Switching Networks," *Proc. IEEE,* vol. 83, no. 10, pp. 1374-1399, Oct. 1995.

**Jean-François Hermant** received the PhD degree in computer science from the University of Paris VI in 1999, a "DEA" (French post-graduate degree) in electronics from the University of Paris XI in 1994, and an engineer degree from ESIGETEL (French graduate school in Computer Science and Telecommunications) in 1994. He joined INRIA (the French National Institute for Research in Computer Science and Control), Rocquencourt, in 1995, as a research associate. He spent one year at the École Polytechnique, Palaiseau, in 1998, as a scientist in the Laboratory of Computer Science. In 2001, he joined the University of Virginia for one year, as a visiting research associate in the Department of Computer Science. His primary areas of interest are real-time, distributed, and fault-tolerant systems, as well as computer networks.

**Gérard Le Lann** holds the MS degree in applied mathematics from the University of Toulouse, the engineer degree in computer science from ENSEEIHT, Toulouse, and the PhD degree in computer science from the University of Rennes. Since 1969, he has successively joined CERN (the European Research Center for Nuclear Physics, Switzerland), the Cyclades Project (French Ministry of Industry), inspired by the US Arpanet Project, and Stanford University, where he was involved in the design of what became known as the TCP/IP protocol. He joined INRIA (the French National Institute for Research in Computer Science and Control), Rocquencourt, in 1978, as a research director. Since 1977, when he published one of the early papers in distributed computing, he has been active in the areas of networking and real-time, distributed, fault-tolerant computing. His current research interests concern theoretical and practical aspects of asynchronous real-time distributed dependable computing and proof-based system engineering. He is a member of the IEEE Computer Society