Implementing Reliable Distributed Real-Time Systems with the Θ-Model

Jean-François Hermant and Josef Widder*

[1] INRIA Rocquencourt, Projet Novaltis, BP 105
F-78153 Le Chesnay Cedex (France)
jean-francois.hermant@inria.fr
[2] Technische Universität Wien, Embedded Computing Systems Group E182/2
Treitlstraße 3, A-1040 Vienna (Austria)
widder@ecs.tuwien.ac.at

Abstract. A widely accepted viewpoint is that designs for distributed real-time systems should be based on synchronous computational models. Safety in such designs, however, requires that the target system behaves as the synchronous model postulates. We believe that this approach is rather risky, as it rests on solving distributed scheduling problems which are known to be NP-hard. We therefore advocate the use of more relaxed system models, namely asynchronous models equipped with unreliable failure detectors.

To this end, we introduce a novel implementation of the perfect failure detector, resting on an abstract model without upper bounds on end-to-end message delays. Then, we demonstrate how this algorithm can be transferred from the abstract model into a real network/system architecture. Finally, we prove that this solution exhibits real-time behavior.

1 Introduction

The research disciplines of distributed computing (DC) and real-time (RT), although often dealing with common problem domains — e.g. fault-tolerant systems/networks — seem not to reflect each others' results adequately. This might stem from apparently contrary research directions: Due to impossibility results, e.g. the impossibility of solving consensus in asynchronous systems [1], a major driver for DC research was to find weak timing models as close as possible to the asynchronous model (e.g. via unreliable failure detectors) that (1) match the highest possible number of real applications/systems and (2) allow fundamental non-trivial problems such as consensus or atomic broadcast to be solved [2, 3, 4]. Of course, with algorithms designed in some non synchronous model, liveness only can be proven to hold, not timeliness (known bounded finite delays).

In contrast, RT research traditionally focuses on queuing and scheduling disciplines in order to prove timeliness properties considering synchronous models.

* Partially supported by the BM:vit FIT-IT project *DCBA* (proj. no. 808198), and the Austrian FWF project *Theta* (proj. no. P17757-N04).

J.H. Anderson, G. Prencipe, and R. Wattenhofer (Eds.): OPODIS 2005, LNCS 3974, pp. 334–350, 2006.
© Springer-Verlag Berlin Heidelberg 2006

However, as is well known, proofs of timeliness properties involve solving combinatorial problems that are NP-hard, usually considering *simplified* models of reality (e.g. no failures, no cache-memory conflicts). We arrive at the mentioned gap: DC seeks solutions that are as far as possible independent of the system's timing behavior while RT considers underlying timing behaviors so as to ensure that, e.g. deadlines are met.

Is it the case that DC results regarding weak (partial) synchrony — assuming weak timing behavior — are of no use for designing RT systems, as the timeliness properties in RT should be predictable? (Exaggeratedly, one could rephrase this question to "Do DC results that guarantee safety and liveness in asynchronous *models* suddenly lose their properties when used in a synchronous *system*?")

Consider, e.g., the classic failure detector (FD) implementation by Chandra and Toueg [5]. The guarantee we get with such FDs is that crashes can eventually be detected *if eventually some unknown bound on message transmission and computing speeds holds*. At first sight, "eventually" and "some unknown bound" seem to be of little use when one wants to guarantee *hard real-time* behavior. This is mistaken. Just run Chandra and Toueg's FD implementation in a system where worst-case response times have been properly established via (distributed) schedulability analysis. In such systems one can derive worst-case response times for failure detection. Note again, that the *distributed algorithm* that implements the FD was designed for a generalized partially synchronous model (close to asynchrony), but now it provides real-time behavior!

But why should one take such a (not really straightforward) design approach? To this question there are several answers. Let us discuss two of them.

Safety. No timeliness failure — i.e., violation of the demonstrated timeliness properties — can ever lead to disagreement with asynchronous consensus algorithms based e.g. on eventually strong FD $\diamond\mathcal{S}$ [5].

This is of great interest as every solution to some RT scheduling problem inevitably rests on some schedulability analysis yielding "worst-cases under a given set of assumptions". As every assumption has a non-zero probability of being violated during operation (as we cannot tell the future) every derived bound on e.g. message delays could be violated. In such a case an algorithm that relies on these bounds will cease to work. That is, one loses safety (consistency) along with timeliness, although it need not be this way.

Performance. When trying to "implement" synchronous models, i.e., provide applications the illusion that they run in synchronous systems, timers (e.g. round durations, periods of scheduler activation) have to be set to worst-case values computed for worst-case load and failure conditions (e.g. re-transmissions of lost messages contributes to delay bounds). Thus, in many synchronous system designs, run-time behavior which is dictated by worst-case timeouts is bound to be worst-case behavior. Synchronous designs are inefficient by construction.

This deficiency increases even more as new applications for real-time systems demand highly dynamic computations. Dimensioning the systems according to rare (on demand) computations is expensive, and often not even required from a safety point of view. This can be avoided by favoring designs based on less restrictive assumptions. From these considerations emerged the concept of *design immersion* or *late binding*, which has been introduced and discussed formally by Le Lann [6, 7, 8].

Contribution. This paper provides an example of how weak synchrony assumptions can be immersed into systems to achieve real-time behavior. In Sect. 3, we present a novel algorithm for implementing the perfect FD $\mathcal{P}$ [5] in the Θ-Model, where one does not assume local clocks, or bounded computational step time or upper bounds on message transmission delay [9, 4, 10]. Rather, the Θ-Model assumes just eventual step/termination and a bounded ratio of end-to-end delays experienced by messages that are in transit simultaneously. We then show how to immerse our algorithm into a system where clocks may exist and bounded computational step times are used to implement and prove real-time behavior. To this end, we revisit the architecture that was employed in [11]. This allows us to compare timeout based FD implementations to our solutions that are message-driven and timer-free.

2 Model

For our presentation of the algorithm and our formal analysis[1] we consider a system of n distributed processes denoted as $p, q, \ldots$, which communicate through a reliable, error free and fully connected point-to-point network. We assume that a non-faulty receiver of a message knows the sender. The communication channels between processes need not provide FIFO transmission, and there is no authentication service.

At most f processes may stop by prematurely halting. A process is considered correct until it stops operation. Since we will immerse our solution into a broadcast network (Deterministic Ethernet [13]) later, we assume that any broadcast message is either received by all correct processes or by none (in the case the sender crashes before finishing its broadcast). This leads to simple fault semantics, i.e., clean crashes.

We now give two different models. The dynamic model of Sect. 2.1 will just be considered for coverage analysis. In this model, upper bounds on message end-to-end delays do not exist. There is just a relation of long and short transmission times of those messages which are simultaneously in transit. In order to keep the analysis simple we give the static model in Sect. 2.2 which is logically equivalent, i.e., any problem solvable in one of the models has a solution in the other model (see Theorem 1). However, both models assume that processes have no access to local clocks and can take a computational step only as a reaction to received messages, thus they are message-driven.

---

[1] Due to space restrictions we had to omit the full analysis of our algorithm. It can be found in the full version of the paper [12].

2.1 Dynamic Timing Model

In our dynamic model, processes communicate by message passing. The time interval a message m is in transit consists of three parts: Local message preparation and queuing at the sender, transmission over the link, and local receive computation and queuing at the receiver. We denote as t_s^m the instant the preparation of message m starts. The instant the receive computation is finished we denote as t_r^m. We assume that all communication is done by broadcasting.

In the dynamic model we say that message m is *in transit* during time interval $(t_s^m, t_r^m]$. We denote $\eta^m = t_r^m - t_s^m$ the finite end-to-end computational + queuing + transmission delay of message m sent from one correct process to another. Let $\mathcal{M}(t)$ be the set of all messages which are in transit at time t. Let $\delta(t)$ be a lower envelope function on transmission delays of all messages that are in transit at time t, such that for any time t it holds that $\delta(t) \leq min(\eta^m)$ for all $m \in \mathcal{M}(t)$ if $|\mathcal{M}(t)| > 0$ and $\delta(t) = 1$ otherwise. We define for a fixed $\Theta \in \mathbb{R}, \Theta \geq 1$ the upper envelope function $\Delta(t) = \Theta\delta(t)$. At any time t it must hold that $\Delta(t) \geq max(\eta^m)$ for all $m \in \mathcal{M}(t)$ if $|\mathcal{M}(t)| > 0$.

2.2 Static Timing Model

In contrast to the dynamic model, the static one stipulates an upper bound Δ on end-to-end delays as well as a lower bound δ such that $0 < \delta \leq \eta^m \leq \Delta < \infty$, where δ and Δ are not known in advance. Since $\Delta < \infty$, every message sent from a correct process to another one is eventually received. The *transmission delay ratio* is $\Theta = \Delta/\delta$. For our formal treatment we assume that processes have a priori knowledge of some integer Ξ (a function of Θ; cf. Theorem 5) but no knowledge on time bounds. Moreover, there is no access to an external time base, hardware clocks or similar devices that allow to get a notion of elapsed time; in other words, executions are message-driven. It follows that time passing information has to be obtained solely out of the message pattern.

In [10] we have shown the following theorem by proving that the runs of the dynamic and the static model cannot be distinguished by the processes. It follows that an algorithm that was proven correct in the static model is correct in the dynamic model as well.

Theorem 1 (Equivalence [10, Theorem 1]). *The dynamic model and the static model have the same expressive power.*

Uncertainty. Processes only communicate by broadcasting. In literature (in particular in work on clock synchronization [14]), usually, the delay uncertainty $E = \Delta - \delta$ is employed to discuss the effect of timing uncertainty (jitter). Our analysis reveals, however, that for our broadcast based algorithm we can employ a finer grained measure. Assume that a message m is broadcast at time t. It will be received by correct processes in the interval $[t + r_m, t + R_m]$ with $r_m \geq \delta$ and $R_m \leq \Delta$. We define the broadcast uncertainty $\varepsilon \geq R_m - r_m$ for all messages m. Obviously we have $\varepsilon \leq E < \Delta$. We will see in Sect. 5.2 that distinguishing

between ε and E makes a big difference. It will turn out that in our targeted architecture the value of ε is close to 0 while E remains close to Δ.

Significant Timing Values. In this paper, we consider a round based algorithm that is executed in asynchronous rounds, i.e., every correct process sends a message in every round k. The transition to round $k+1$ occurs when $n-f$ messages for the current round are received. It will turn out that the uncertainty we have to deal with does not stem from the ratio of message delays directly but rather from the ratio of the longest message delay and the shortest round switching intervals. The shortest round-switching interval δ^r, however, is not determined by only one single correct message. Rather it is determined by the sending time of the first message and the receive time of the $(n-f)^{\text{th}}$ message. This might seem irrelevant since any message is bounded by δ, and all could be sent simultaneously. From a practical point of view — as is confirmed by our analysis in Sect. 4 — this is very important, however. If one tries to establish an analytical expression for δ one would examine an idle system and the sending of a single message in this system — which could be a self reception as well. Obviously the receiver just has to deliver one message here. However, assuming that a receiver can process, say $n-f$ messages as fast as a single one is typically not valid in real systems — this would amount to assuming infinite computational power. Choosing $\delta^r = \delta$ hence would be overly conservative since round switching requires $n-f$ messages, i.e. is determined by the $(n-f)^{\text{th}}$ fastest message. Moreover, in broadcast bus networks one cannot transmit two messages simultaneously over the bus, i.e. the $n-f$ fastest messages must be transmitted one after the other. The time for $n-f$ messages to be transmitted in such networks is hence always larger than the best-case time of sending a single message in an idle system. Using δ in the analysis would lead to over-valuation of its significance.

In our analysis, we will hence set δ^r equal to the transmission time of the $(n-f)^{\text{th}}$ fastest message. That is, we will use δ^r as expression for the shortest time it may take to send $n-f$ messages from distinct processes to a single receiver (end-to-end). Let us formalize this:

Definition 1 (Incoming Messages). *For any correct process q, δ_q is the $n-f$ smallest η^m for all messages m sent by distinct correct processes that enable an event at q. δ^r is defined as the the smallest δ_q of all correct processes q.*

Lemma 1 (Sending Time). *If a correct process receives messages from at least $n-f$ distinct correct processes by time t, then at least one message was sent by time $t-\delta^r$.*

2.3 Event Generation

In previous work [15, 4] we considered purely message-driven algorithms. These algorithms are started by an external event, which triggers the first computational step. All steps after the first one are direct responses to received messages. As already shown in [4], such protocols can be employed efficiently in systems with large delay×bandwidth product; e.g. satellite broadcast communication

link. In the architecture of Sect. 4, we show how to fine tune the overhead by introducing mute periods.[2] To this end we add local events to the model. Previous work that investigated the intersection of message-driven and time-driven semantics can be found in [16] where one has shown that message-driven semantics are weaker than time-driven ones by proving that the problem of self-stabilizing failure detection cannot have a message-driven solution [16] while time-driven solutions are known [17]. In fact, in [16] we proved even more. We showed that the impossibility result even holds if there are locally generated (deadlock prevention) events where no assumption is made on the occurrence of their arrival such that their semantics are too weak to employ them as clocks.

Following this result we add local events without assumptions on arrival laws to our model as well. We use local event generation in order to start instances of the basic round synchronization algorithm. In our theoretical analysis we show that the correctness of the algorithm does not depend on the actual intervals between these events. In our implementation example we then use these events in order to control the overhead such that we compare our performance with previous failure detector implementations [11].

3 General Implementation and Analysis of Perfect Failure Detector $\mathcal{P}$

The need for a definition of (unreliable) FDs emanated from the impossibility [1] of deterministic fault-tolerant consensus in asynchronous distributed systems. In their seminal paper, Chandra and Toueg [5] characterized FDs by the two properties completeness and accuracy. In an asynchronous distributed system equipped with FDs consensus is solvable. In this paper we will give an implementation of the perfect failure detector $\mathcal{P}$ that ensures the following two properties.

Strong Completeness. Eventually, every correct process permanently suspects every crashed process.

Strong Accuracy. No process is suspected before it crashes.

Our implementation of $\mathcal{P}$ follows an idea originally proposed in [9]; its pseudocode is given in Fig. 1. It is executed in consecutive instantiations which are numbered using variable i (see `line 1`). We will see that each instantiation can be regarded independently and that a process that crashes before instantiation j is started will be suspected at the end of j. The algorithm is started independently at each process by sending $(round, 0, 0)$ in `line 4` — *round* being just a message identifier. Note that round $k = 0$ is the only round where not all correct processes must send messages. The code inside the statement starting at `line 6` is a simple round synchronization algorithm, i.e., the round number is a counter of the terminated rounds of computation. If a process has received $n - f$ messages from distinct processes for the current round k it increases k and sends its

[2] This is done in all FD implementations where e.g. heartbeats are sent every x (physical) time units while the FD remains mute in between.

```
0:  VAR k : integer := 0;                              /* round number               */
1:  VAR i : integer := 0;                              /* instantiation number       */
2:  VAR SL[n] : boolean := false;                      /* list of suspected processes */
3:  VAR saw_max[n][∞] : integer := 0;

4:  broadcast (round, 0, 0) [once];                    /* Initialization             */

5:  /* Round Synchronization */
6:  if received (round, i, k) from at least n − f distinct processes
7:        →  k := k + 1;
8:           if k > Ξ
9:                →  ∀q: if saw_max[q][i] = 0 → SL[q]:= true;
10:                   k := 0;
11:                   i := i + 1;
12:                   control_overhead;
13:          broadcast (round, i, k) [once];           /* start next round           */

14: if received (round, j, ℓ) from q                   /* Store received messages    */
15:       → saw_max[q][j] := max(ℓ, saw_max[q][j]);
```

Fig. 1. Perfect Failure Detector Implementation

message for the new round. Due to a priori knowledge of Ξ, a process can determine upon updating its counter whether messages from other processes for past rounds are missing. It does so in `line 8` where it checks whether all processes succeeded in sending at least $(round, i, 1)$. Processes which did not, must have crashed and are therefore suspected, i.e., added to the list of suspects SL which is the interface to upper layer applications (see Sect. 4). After that, k and i are updated and the next instantiation is started after the *control_overhead* command (see `line 12`), which will be used to create silent intervals between two instantiations, when the algorithm is immersed into our targeted architecture. It does so by, e.g., scheduling the transmission of the given message for some later point in time. Another way to implement timer-free local waiting are hardware instructions that exist in certain computers that allow to trigger some event when x instructions have been executed.

The chosen value of this timeout — we will introduce τ in Definition 3 as upper bound on it — neither has an influence on the correctness of the FD implementation (it could as well be set to 0) nor must be the same at every process. Moreover if there is no local timer, diverse local information may be used to get some rough estimate of elapsed time (counting interrupts or updates of the program counter etc.). In our example (Deterministic Ethernet in Sect. 5), we derive a timeout value which leads to a worst-case overhead of 5% for FD messages.

Contrary to most other FD implementations, we do not use hardware clocks to achieve FD properties, neither do we require upper bounds on message delays. Other FD implementations use local clocks or timers to timeout (1) τ and (2) the upper bound on end-to-end delays. Our solution neither relies on local information about τ nor on the existence of some upper bounds on end-to-end delays in

order to detect a crashed process. It follows that our timeout mechanism remains message-driven.

To complete the description of our algorithm: The code of the statement in `line 14` stores which processes have sent messages for which round. Only the message for the largest round number has to be stored here. (Note that the declaration of saw_max in `line 3` includes an infinity of rows in the matrix just for conciseness of presentation. In real implementations, information from past instantiations need not be stored such that just bounded memory is needed in order to maintain the required information.)

In order to show that our algorithm does implement $\mathcal{P}$ we first analyze some properties of its included round synchronization algorithm (`line 6`) where k is the local round number. Since messages for different instantiations do not interfere logically with each other, we just examine the rounds for one instantiation here. For conciseness we therefore suppress i in the following. We follow the analysis of [15] where a logical clock synchronization algorithm in the presence of Byzantine faults was considered. We just focus on crashes and therefore have a simpler algorithm. After that we show the FD properties based upon these round synchronization properties. We start with some preliminary definitions.

Theorem 2 (Properties). *In the presence of $f < n$ faults, the algorithm given in Fig. 1 satisfies the following properties:*

(P) Uniform Progress. *If all correct processes set their round numbers to k by time t, then every process sets its round number at least to $k+1$ by time $t+\Delta$.*
(U) Uniform Unforgeability. *If no process sets its round number to k by time t, then no process sets its round number to $k+1$ by time $t+\delta^r$ or earlier.*
(S) Uniform Simultaneity. *If some process sets its round number to k at time t, then every process sets its round number at least to k by time $t+\varepsilon$.*

Lemma 2 (Fastest Progress). *Let the first correct process set its round number to k at time t. Then no correct process can reach a larger round number $k' > k$ before $t + \delta^r(k'-k)$.*

After discussing the fundamental round synchronization properties we now turn our attention to the problem of failure detection. We start with the behavior of instantiations.

Definition 2 (Instantiation). *The start of instantiation i is defined to be the earliest time b_i by which $n-f$ processes have sent $(round, i, 0)$. Further, the end of instantiation i is defined to be the time e_i the last process sets its round number to $k > \Xi$.*

The following corollary follows directly from (P) and Definition 2, which introduces D as the worst-case time between start and end of an instantiation.

Corollary 1 (Instantiation Termination Time). *For all instantiations i it holds that $e_i - b_i \leq D$, where $D = (\Xi + 1)\Delta$.*

As discussed above, we would like to fine tune the overhead of the FD algorithm to application requirements by inserting silent periods in our algorithm. In order to give a bound on detection latency we have to assume an upper bound on the duration of these periods. The value τ that we introduce in the following definition can in fact be arbitrary (depending on the required overhead) such that it cannot be used as a weak clock [3] that could be used to timeout processes.

Definition 3 (Intermission). *$\tau \geq 0$ is the upper bound on the timeout of the* control_overhead *call.*

Corollary 2 (Intermission Period). *For all instantiations i and $i+1$ it holds that $b_{i+1} - e_i \leq \tau$.*

With the results regarding timing we now turn our attention to FD semantics.

Theorem 3 (Strong Completeness). *Let Ξ be some positive integer. In a system with $n > f$ processes, the algorithm given in Fig. 1 ensures that each process p that crashes by time b_i is suspected by all correct processes by time e_i.*

We now give two theorems for strong accuracy. Theorem 5 is typical for results in the Θ-Model as no time unit parameters show up but just the integer $\Xi \geq \Theta$. From Theorem 4, however, one sees the parameters that have to be evaluated during immersion more explicitly, i.e., Δ, ε, and δ^r.

Theorem 4 (Strong Accuracy). *Let $\Xi \geq \left\lfloor \frac{\Delta}{\delta^r} + \frac{\varepsilon}{\delta^r} \right\rfloor + 1$. In a system with $n > f$ processes, the algorithm given in Fig. 1 ensures that no process is suspected before it crashes.*

Theorem 5 (Strong Accuracy in the Θ-Model). *Let $\Xi \geq \lfloor 2\Theta \rfloor$. In a system with $n > f$ processes, the algorithm given in Fig. 1 ensures that no process is suspected before it crashes.*

Assume that we have a system with $\varepsilon = 0$ and $\left\lfloor \frac{\Delta}{\delta^r} \right\rfloor = 1$ which according to Theorem 4 requires $\Xi \geq 2$ for failure detection with our algorithm. Following Theorem 5, we say that the system *behaves as* a system obeying the Θ-Model with $\Theta = 1$. Similarly we show in Sect. 5 that our architecture built upon Deterministic Ethernet requires just $\Xi \geq 2$ as well.

To achieve real-time behavior we require an upper bound on detection time. We give such a bound only for crashes that happen after time b_0, since obviously we cannot bound detection latencies for crashes that occurred before the algorithm was running. Such crashes, however, will be detected by all correct processes by time e_0 by Theorem 3.

Theorem 6 (Detection Latency). *Let Ξ be according to Theorem 4 resp. Theorem 5. In a system with $n > f$ processes, the algorithm given in Fig. 1 ensures that a crash occurring at time $t \geq b_0$ is detected by time $t + L$, with detection latency $L = \tau + 2D$.*

Theorems 3 and 5 show that completeness and accuracy just depend on Θ while only *timeliness* depends on Θ and Δ (i.e., the assumed time bound). If Δ is violated but Θ still holds (which is possible in many real systems/networks [9]) we still guarantee completeness and accuracy while just timely detection is lost. This is the best one can hope for, given that Δ is violated here!

Despite Theorem 6 we have no solution to a real-time problem yet, since we did not show how to implement both the system model and the algorithm in a real network. To this end we have to show that the assumed timing behavior in the abstract model can be matched by demonstrated timeliness properties in real systems. We do so in the following section in order to give a complete solution to the RT FD problem for a network architecture based on Deterministic Ethernet and suitable scheduling algorithms.

4 Architectural Model

In this section, we sketch out a generic architectural model of the systems under consideration. See [11] for a detailed presentation of that model.

We consider a finite set Π of processors, interconnected by a network, referred to as Net. The nominal size of Π is $n > 1$. The model of a processor is given in Fig. 2. The software/hardware architecture is modeled after a number of levels, such as the application software level, the middleware level, the executive/operating system level, various communication protocol levels, the input/output (I/O) level. Processors act as sources of messages. There are two

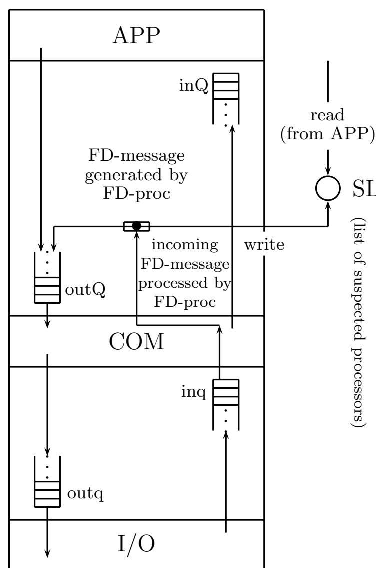

Fig. 2. Architectural model of a processor

types of messages. Messages handled by FDs are denoted FD-messages. Messages other than FD-messages are referred to as ordinary messages (e.g. application, middleware, system). Let COM denote the level of communication protocols where one finds FD modules. An FD module consists of a process, denoted FD-proc, which maintains a local list of suspects, denoted SL. At any time, $SL(p)$ contains the names of those processors that p's FD suspects (rightly or erroneously) of having failed. In addition, FD-proc receives and broadcasts FD-messages so as to "prove" that its processor has not failed. Let APP denote the application level.

Let us model those waiting queues visited by outgoing messages, from the APP 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. An FD-message is initially deposited by FD-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 FD-proc after being serviced in *inq*. An ordinary message is initially deposited by an algorithm A 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 A after being serviced in *inQ*.

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 service times corresponding to the four waiting queues, respectively, that is the worst-case times for servicing a message pending in each of these queues. We define d_m as the blocking factor with Net, i.e., d_m is the exact time needed for transmitting the longest ordinary message over the physical link between a processor and Net.

Let γ stand for an upper bound on end-to-end delays for an FD-message, measured at the COM level.

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 policies 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 behind possibly pending FD-messages. It is serviced prior to any ordinary message.

In order to resolve interprocessor competition for network and processor resources 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 Π, 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 $\mathcal{S}N$ algorithm:

$\mathcal{S}N$. 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.

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 ν be the smallest FD-message inter-arrival 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 $\mathcal{S}W$, $x' = 1$ 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 policies $\mathcal{S}W$ and $\mathcal{S}N$ and bound $\psi'(x')$ is determined by algorithm $\mathcal{S}W$. Hence, $\psi(x)$ and $\psi'(x')$ are tight. Let θ 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 θ can be computed, considering that optimal schedulers (proper to Net) service FD-messages prior to ordinary messages. Consequently, for an FD-message generated by that processor assigned index x, tight bound $\gamma(x)$ is as follows:

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

Note the importance of differentiating between end-to-end delays proper to every process in a system. If no such differentiation is made, one faces a circular dependency. A distributed system consists of some architecture (processors, links) and all the system processes and application processes that run on the architecture. It makes no sense to talk about upper bounds on message delays in the "system" if one part of the system, i.e., the system processes and application processes are not known since they may produce unknown loads on the processors and/or links. Conversely, if one specifies the exact scheduling policy used for every class of processes, the circular dependency vanishes. For a given process p, only processes scheduled prior to p may influence upper bounds on response times. When choosing a head-of-the-line policy for a process (our case, for the FD process), then it is possible to conduct a worst-case schedulability analysis for that process ignoring all other processes (to the exception of simple blocking factors in case of non preemption). This is what we have shown how to do.

In Sect. 5, we consider Deterministic Ethernets and establish the analytical expression of γ for such networks. Then, we derive ρ, the worst-case overhead induced by FD-messages, and finally L, a tight upper bound on processor failure detection latencies. Before that we shortly discuss the network traffic which is induced by our FD implementation.

5 Illustration with Ethernets

We illustrate our generic results with Ethernet-like networks. Let COM be the ISO/OSI data link level. With Ethernets, $\theta = 0$ as $\psi(x)$ includes local physical transmission delays and there is no additional transmission delay. Hence, $\gamma(x) = w_{outQ} + w_{outq} + d_m + \psi(x) + x' \, w_{inq}$. Being concerned with real-time systems, we must consider a deterministic variant of the original Ethernet CSMA/CD protocol. This variant has been implemented in COTS products and is called CSMA/DCR (Carrier Sense Multi Access / Deterministic Collision Resolution), which is based on distributed deterministic balanced m-ary tree searches [13].

5.1 CSMA/DCR

Broadcast media are physically characterized by a channel slot time, denoted σ. Sources of messages are processors. Channel sharing between sources works like 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 indexes 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 ξ_x^l be the maximum number of steps needed to search x leaves in a m-ary tree of l leaves. In [18] and [19], one shows for $x \in \{2, \ldots, l\}$:

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

This formula applies for any assignment of x indexes over l sources. The worst-case delay involved with resolving a collision fully is $\Sigma(x) + \xi_x^l \, \sigma$.

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 $\mathcal{JS}$ 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.

5.2 Behavior of the FD in Our Architecture

Evaluating ε. Our FD algorithm requires a priori knowledge of Ξ. Therefore, we examine the timing of our architecture in order to derive values for the

timing parameters used in Theorem 4. Due to the physical properties of Net, all messages are received by all processes at I/O level within σ. Due to our queuing algorithm $\mathcal{SW}$, all FD-messages are taken out of *inq* by the FD-processes within σ as well. That is, $\varepsilon = \sigma$.

Bound on γ. We now derive a term for calculating the worst-case end-to-end delay of a message that participates in a collision on Net with x messages.

We use the $\mathcal{JS}$ mechanism 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, hence the blocking factor is d_m. After $\mathcal{JS}$ has been generated, only FD-messages are transmitted during the same tree search. Transmission of ordinary messages is resumed from its preemption state when the channel returns to idle. Given that the body of an FD-message requires only to contain 2 bits[3] (which can easily be stored within the smallest possible Ethernet message), its physical transmission delay is that of a message of minimum duration, i.e., slot time σ. Therefore, $\Sigma(x) = x\,\sigma$. Tight bound $\psi(x)$ is as follows:

$$\psi(x) = \left(log_m(l) + x + \xi_x^l\right)\,\sigma.$$

The smallest FD-message inter-arrival delay is σ. Hence, $x' = \lceil x\;(1 - \sigma/w_{inq})\rceil$ if $\sigma < w_{inq}$, $x' = 1$ if $\sigma \geq w_{inq}$. Tight bound γ for the xth FD-message is:

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

Evaluating δ^r. We already partially discussed the expression of Theorem 4 (i.e., ε), which can be used to determine a numerical value for Ξ. What remains to do is to give an analytical expression of $\frac{\Delta}{\delta^r}$. In the worst-case, $\Delta = \gamma(n)$. It remains to derive an expression for δ^r. Recall that δ^r is defined as a *lower bound* on the time between when the first of $n-f$ messages is sent to some process p and when p receives the last of the $n-f$ messages. Let us assume pessimistically that on the bus the $n-f$ messages are sent back to back, i.e., on the bus they require $(n-f)\cdot\sigma$ time units. (Note that this lower bound is by no means tight as our solution always requires $\mathcal{JS}$ which leads to a distributed tree search.) At *inq* the $(n-f)$ messages, however, queue up such that from the time the last message arrives additional $(n-f)'\cdot w_{inq}$ time units are required. At the respective outgoing queues at the first sender we have to add w_{outQ} and w_{outq} while we assume that there is no congestion at Net. Summing up, we see that:

$$\delta^r = w_{outQ} + w_{outq} + (n-f)\cdot\sigma + (n-f)'\cdot w_{inq}.$$

5.3 Numerical Examples and Discussion

We use the same numerical values as in [11] to have a meaningful comparison. Let us consider 10 MBit/s Ethernets. According to the ISO/OSI standard, $n \leq 1,024$

[3] We will justify this in Sect. 5.3 by showing that $\Xi = 2$ suffices for failure detection.

Case 1: $n = l = 16$

results of [11]	our results
$D = 5.93\,ms$	$D = 17.78\,ms$
$\tau = 103.55\,ms$	$\tau = 292.87\,ms$
$L = 114.61\,ms$	$L = 328.44\,ms$

Case 2: $n = l = 1,024$

results of [11]	our results
$D = 275.39\,ms$	$D = 826.18\,ms$
$\tau = 6.52288\,s$	$\tau = 18.74246\,s$
$L = 7.07287\,s$	$L = 20.39482\,s$

Fig. 3. Comparison to [11]

and $\sigma = 51.2\ \mu$s (microseconds). We assume that the size of the longest ordinary message (I/O level framing) is 10,000 bits, i.e., $d_m = 1$ ms (millisecond). Let us pick up 250 μs for each of the service times w_{outQ}, w_{outq}, and w_{inq}. Hence $\gamma(x) = 1.5 + \psi(x) + 0.25\,x'$ (in ms), with $x' = \lceil 0.7952\,x \rceil$. Results shown below are rounded up to a precision of 10 μs. We consider quaternary trees ($m = 4$).

With Perfect FDs, $x = n$. As in [11] we pick up $f = 5$ as the upper bound on the number of processes that can crash during the execution of the algorithm (a very high number given practical fault probabilities). Thus, we get:

Case 1: $\boldsymbol{n = l = 16, f = 5}$. $x' = 13$, $\psi(16) = \psi_1 = 1.18$ ms, hence $\gamma(16) = \gamma_1 = 5.93$ ms.

$(n-f)' = 9$, $\delta_1^r = 3.31$ ms, hence $\frac{\Delta}{\delta_1^r} = \frac{\gamma_1}{\delta_1^r} = \frac{5.93 \text{ ms}}{3.31 \text{ ms}} = 1.79$. Adding $\frac{\varepsilon}{\delta_1^r} = \frac{51.2\ \mu\text{s}}{3.31 \text{ ms}} = 0.02$ to this and applying the floor function (cf. Theorem 4) we get $\Xi = 2$. By Corollary 1 the termination time of an instantiation $D = (\Xi + 1)\Delta$ such that we get $D_1 = 3\gamma_1 = 17.78$ ms

Case 2: $\boldsymbol{n = l = 1024, f = 5}$. $x' = 815$, $\psi(1,024) = \psi_2 = 70.14$ ms, hence $\gamma(1,024) = \gamma_2 = 275.39$ ms.

$(n-f)' = 811$, $\delta_2^r = 255.42$ ms, hence $\frac{\Delta}{\delta_2^r} = \frac{\gamma_2}{\delta_2^r} = \frac{275.39 \text{ ms}}{255.42 \text{ ms}} = 1.08$. Additionally, $\frac{\varepsilon}{\delta_2^r} = \frac{51.2\ \mu\text{s}}{255.42 \text{ ms}} = 0.0002$ such that again $\Xi = 2$. $D_2 = 3\gamma_2 = 826.18$ ms.

The worst-case FD-message overhead is $\rho = 3(\psi(x) + x\,w_{inq})/(D + \tau)$. Let us pick up $\rho = 5\%$. We get $\tau = 60\,(\psi(x) + 0.25\,x) - D$ (in ms). Thus we derive the following values for the mute periods τ and the tight upper bound on the failure detection latency L (cf. Theorem 6).

Case 1: $\tau_1 = 292.87$ ms, $L_1 = 328.44$ ms.

Case 2: $\tau_2 = 18.74246$ s, $L_2 = 20.39482$ s.

Let us compare our results with those presented in [11]. In Fig. 3, we see that the "worst-case price" for our message-driven implementation is a factor 3. This means that we have a worst-case detection latency that is 3 times larger with the same overhead or that one can achieve the same detection latency by tripling the overhead.

At first sight, these results seem to contradict one of the arguments (performance) at the core of the design immersion or late binding principle (cf. Sect. 1). This is not the case. Firstly, a complete comparison should take coverage into consideration. In other words, one should compare numerical results considering violations of the assumed timing behavior postulated in a synchronous model, i.e. timing bounds that are not tight, these bounds being such that the probability of having a bound violated or Θ violated is the same. Secondly, the run-time

behavior induced by the synchronous design of [11] is always the actual worst-case behavior, even if actual delays are smaller than their postulated bounds most of the time (e.g. the blocking factor d_m). Conversely, the run-time behavior induced by the asynchronous design given in this paper simply matches the best-case or average-case message delay scenarios, yielding actual failure detection latencies (much) smaller than values computed for L.

Finally, experimental results [20] confirm the analytical ones, showing that broadcast bus based systems are particularly well suited for the Θ-Model. In our architecture we have seen that the value of Θ (derived from Ξ) is much smaller than the ratio of absolute bounds on worst-case and best-case message end-to-end delays.

6 Conclusions

In this paper, we have illustrated a number of concepts related to real-time computing and asynchronous models of computation. From a theoretical viewpoint, an interesting result is that we showed that local physical clocks are not required to detect crashed processes in bounded finite time. To this end, we have presented an implementation of the perfect FD $\mathcal{P}$ in the Θ-Model. One major merit of this implementation is its very high coverage, since the FD $\mathcal{P}$ semantics are not violated when postulated end-to-end upper bounds are violated, provided that Θ is not violated, which is not the case with implementations in partially synchronous models.

Acknowledgments. We are grateful to Gérard Le Lann and Ulrich Schmid for many valuable discussions on the Θ-Model.

References

1. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. Journal of the ACM **32**(2) (1985) 374–382
2. Aguilera, M.K., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: On implementing Omega with weak reliability and synchrony assumptions. In: Proceeding of the 22nd Annual ACM Symposium on Principles of Distributed Computing (PODC'03). (2003)
3. Fetzer, C., Schmid, U., Süßkraut, M.: On the possibility of consensus in asynchronous systems with finite average response times. In: Proceedings of the 25th International Conference on Distributed Computing Systems (ICDCS'05), Columbus, Ohio, USA (2005)
4. Widder, J., Le Lann, G., Schmid, U.: Failure detection with booting in partially synchronous systems. In: Proceedings of the 5th European Dependable Computing Conference (EDCC-5). Volume 3463 of LNCS., Budapest, Hungary, Springer Verlag (2005) 20–37
5. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM **43**(2) (1996) 225–267

6. Le Lann, G.: On real-time and non real-time distributed computing. In: Proc. 9th Int'l Workshop on Distributed Algorithms. Volume 972 of LNCS., Springer-Verlag (1995) 51–70 invited paper.
7. Le Lann, G.: Proof-based system engineering and embedded systems. In: Proc. European School on Embedded Systems. Volume 1494 of LNCS., Springer Verlag (1996) 208–248 invited paper.
8. Le Lann, G.: Asynchrony and real-time dependable computing. In: 8th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2003), Guadalajara, Mexico. (2003) 18–25
9. Le Lann, G., Schmid, U.: How to implement a timer-free perfect failure detector in partially synchronous systems. Technical Report 183/1-127, Department of Automation, Technische Universität Wien (2003)
10. Widder, J.: Distributed Computing in the Presence of Bounded Asynchrony. PhD thesis, Vienna University of Technology, Fakultät für Informatik (2004)
11. Hermant, J.F., Le Lann, G.: Fast asynchronous uniform consensus in real-time distributed systems. IEEE Transactions on Computers **51**(8) (2002) 931–944
12. Hermant, J.F., Widder, J.: Implementing time free designs for distributed real-time systems (a case study). Research Report 23/2004, Technische Universität Wien, Institut für Technische Informatik (2004) Joint Research Report with INRIA Rocquencourt.
13. Le Lann, G., Rolin, P.: 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, November 1984 (1984)
14. Lundelius-Welch, J., Lynch, N.A.: An upper and lower bound for clock synchronization. Information and Control **62** (1984) 190–204
15. Widder, J.: Booting clock synchronization in partially synchronous systems. In: Proceedings of the 17th International Symposium on Distributed Computing (DISC'03). Volume 2848 of LNCS., Sorrento, Italy, Springer Verlag (2003) 121–135
16. Hutle, M., Widder, J.: On the possibility and the impossibility of message-driven self-stabilizing failure detection. In: Proceedings of the 7th International Symposium on Self Stabilizing Systems (SSS 2005). Volume 3764 of LNCS., Barcelona, Spain, Springer Verlag (2005) 153–170 — Appeared also as brief announcement in *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing (PODC'05).*
17. Beauquier, J., Kekkonen-Moneta, S.: Fault-tolerance and self-stabilization: Impossibility results and solutions using self-stabilizing failure detectors. International Journal of Systems Science **28**(11) (1997) 1177–1187
18. Hermant, J.F., Le Lann, G.: A protocol and correctness proofs for real-time high-performance broadcast networks. Proc. IEEE Int'l Conf. Distributed Computing Systems (1998) 360–369
19. Hermant, J.F.: Quelques problèmes et solutions en ordonnancement temps réel pour systèmes répartis. PhD thesis, Paris-VI-Pierre-et-Marie-Curie Univ. (1999)
20. Albeseder, D.: Evaluation of message delay correlation in distributed systems. In: Proceedings of the Third Workshop on Intelligent Solutions for Embedded Systems, Hamburg, Germany (2005)