Implementing Time Free Designs for Distributed Real-Time Systems (A Case Study)

Jean-François Hermant
INRIA Rocquencourt, BP 105
F-78153 Le Chesnay Cedex (France)
jean-francois.hermant@inria.fr

Josef Widder[†]
TU Wien / Embedded Computing Systems
Treitlstraße 3, A-1040 Vienna (Austria)
widder@ecs.tuwien.ac.at

May 14, 2004

Abstract

Design Immersion refers to a design principle for distributed real-time systems which reconciles advantages of asynchronous solutions regarding assumption coverage and real-time requirements. By case study we show how a totally time free algorithm can be implemented in a real network in order to achieve required timeliness properties.

As first step we introduce a novel implementation of the perfect failure detector resting on an abstract system model without upper bounds on end-to-end message delays. In the second step we demonstrate how this algorithm can be transferred from the abstract model into a real network architecture. Finally we prove that this solution exhibits real-time behavior, like a synchronous solution would do. But additionally our algorithm ensures safety and liveness even with violated time bounds.

1 Introduction

The research disciplines of distributed computing theory (DC) and real-time (RT), although often dealing with common problem domains e.g. fault-tolerant networks, seem to us as not reflecting each others recent results in adequate amount. This might stem from apparently contrary research directions: While much effort in DC was taken recently to find system models with possibly weak synchrony (= timing) as-

[*]Joint Research Report with INRIA Rocquencourt

[†]Supported by the Austrian bmvit FIT–IT project DCBA, project no. 808198.

sumptions [1] which still allow solutions to non-trivial problems (as consensus), RT research traditionally focuses on queuing and scheduling schemes in order to ensure possibly strong timing behavior.

This paper makes an effort in reconciling these two approaches. Our work follows the design immersion principle introduced by Le Lann [7, 8]: First, a design should be made based on possibly weak synchrony assumptions and then formally proved that safety requirements (nothing bad ever happens) and liveness requirements (eventually something good happens) are satisfied by this design. Only after this, the solution is immersed into a real target system and e.g. schedulability analysis is used to prove that the solution satisfies timeliness requirements in this target system as well.

The advantage of this approach is increased coverage, i.e. it is more likely that safety and liveness are ensured than with designs resting directly on the assumed timing properties the target system: Improved coverage means that there are system states where the immersed solution still delivers some service while the direct solution does not. In order to investigate this we have to consider exceptional cases: Consider periods where the target system's behavior deviates from the expected one, which is possible as 100% coverage is impossible to reach, especially when missions with long operation times in not perfectly predictable environments have to be considered. In such periods a design resting on weaker assumptions may still deliver safety and liveness, while just timeliness is lost. A design directly resting on the target system loses safety and liveness along with timeliness. Note that in such periods it is impossible for any solution to ensure timeliness, such that the best one can hope for is safety and liveness (properties which can be ensured even in purely asynchronous systems according to DC theory).

These properties were already discussed formally [7, 8]. We want to strengthen and clarify these arguments by case study. We consider a time free implementation of perfect failure detector $\mathcal{P}$ [2]. Such failure detector (FD) modules provide processes with information about processes that (might) have crashed. It is however impossible to implement an FD in purely asynchronous systems, since this would contradict the classic FLP impossibility result [3]. We regard our paper as consisting of two parts: In the first DC part we present the system model (Section 2) which is the closest to pure asynchrony to our knowledge since it is totally time and timer free and does not postulate upper bounds on end-to-end message delays. Section 3 then presents design and analysis of a novel implementation of $\mathcal{P}$. After that, in the RT part, we present an architecture built atop Deterministic Ethernet (which has various properties we will explore, e.g. upper bounds on physical message delays). This architecture follows the fast FD approach [6] where failure detection is done low level and higher level application can make use of it (similar to clock synchronization, where the clocks are synchronized low level in order to reduce the uncertainty). For this architecture we discuss how our DC solution can be implemented in it efficiently by using suitable queuing and bus arbitration schemes. Finally we show that with our approach implementing $\mathcal{P}$ (and hence consensus) has a totally time free solution which can be implemented in real networks such that it exhibits real-time behavior.

2 Model

For our presentation of the algorithm and our formal analysis 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[1]. 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.

Processes may stop by halting. A process is considered correct until it stops operation. Since we will immerse our solution into a broadcast network (Deterministic Ethernet) later, we assume that any broadcast 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 will denote by η^m the end-to-end computational + transmission delay of a message sent between two correct processes; η^m can be different for each message.

We now give two diverse models. The model of Section 2.1 will just be considered for coverage analysis.

[1] As already noted this is just an abstraction. In the model of a real network is given in Section 4.

It describes timing where upper bounds on message end-to-end delays do not exist. There is just a relation of long and short transmission times just of those messages which are simultaneously in transit. In order to keep the analysis simple we give the computational model in Section 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 2.1).

2.1 System Model

Processes communicate by message passing. The time interval a message m is in transit consists of three parts: Local message preparation (includes queuing) at the sender, transmission over the link, and local receive computation (includes queuing) at the receiver. We denote t_s^m the instant the preparation of message m starts. The instant the receive computation is finished we denote as t_r^m.

In our system model we say that message m is *in transit* during real-time interval $[t_s^m, t_r^m)$. We denote $\eta^m = t_r^m - t_s^m$ the end-to-end computational + 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 real-time t. Let $\delta(t)$ be a lower envelope function on transmission delays of all messages that are in transit at real-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 $\Theta \in \mathrm{I\!R}, \Theta \geq 1$ 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 Computational Model

Our timing model 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. Measures for the timing uncertainty are the *transmission delay uncertainty* $\varepsilon = \Delta - \delta$ and the *transmission delay ratio* $\Theta = \Delta/\delta$. In this paper we assume that processes have a priori knowledge of some integer Ξ which is a function of Θ as described later, but no knowledge on real-time bounds.

Theorem 2.1. *The system model and the computational model have the same computational power.*

Proof. Let us assume an omniscient observer in both systems who is equipped with a clock whose timebase β in relation to real-time t is given by $\beta(t) = \int \frac{1}{\delta(t)}\, dt$.

In the computational model, where $\delta(t) = \delta$, we hence get as time base $\beta(t) = t/\delta + c$, thus all correct message transmission delays are measured between 1 and Θ here.

We now consider a message m between two correct processes in the system model. In order to get the measured value of the end-to-end delay we have to consider $\beta(t_r^m) - \beta(t_s^m)$. At any real-time t in the interval $[t_s^m, t_r^m)$ the gradient of $\beta(t)$ is $1/\delta(t)$. Since $\delta^m = t_r^m - t_s^m \geq \delta(t)$ for all times t, $t_s^m \leq t < t_r^m$, it follows that $\beta(t_r^m) - \beta(t_s^m) \geq 1$. Thus any message has a measured end-to-end delay ≥ 1. Moreover, $\Delta(t) = \Theta\delta(t)$ and $\Delta(t) \geq \delta^m$ for all times t such that the measured end-to-end delay cannot be greater than Θ. When monitoring the execution of a distributed algorithm in system S the observer is hence not able to decide whether S is determined by the computational or by the system model. $\square$

3 Failure Detector Implementation

The need for a definition of FDs [2] emanated from the impossibility [3] of deterministic fault-tolerant consensus in asynchronous distributed systems. In their seminal paper Chandra and Toueg [2] described FDs by two properties, i.e. completeness and accuracy and showed that if an asynchronous distributed system was equipped with (even unreliable) FDs, consensus is solvable. In this paper we will give an implementation of perfect failure detector $\mathcal{P}$ whose definition reads as follows:

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 [10] and is given in Figure 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, which 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 5`; 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 7` is a simple software clock synchronization algorithm, i.e. the clock value 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 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 9` 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 (compare Section 4). After that, k and i are updated and the next instantiation is started after the *control_overhead* command, which is just used to make silent intervals between two instantiations. It does so by scheduling the transmission of the given message for some later point in time. Note that this timeout is just used to reduce overhead. If the message would be sent immediately our solution would still reach the required properties, but this would lead to prohibitive overhead for FD messages in real systems. The chosen value of this timeout—we will introduce τ later as upper bound on it—has neither influence on the correctness of the FD implementation (it could as well be set to 0) nor must it be the same at every process. Moreover if there is no local timer accessible, 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 Section 5), we are using a timeout value which will lead to a overhead of 5% for FD messages. Note that the major difference to most other FD implementations is not that we do not use hardware clocks to reach FD properties, but that we do not require fixed upper bounds on message delays. Other FD implementations, however, use local clocks resp. 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 upper bounds on end-to-end delays in order to detect a crashed process; thus our solution is basically time free.

To complete the description of our algorithm: The code of the statement in `line 19` just stores which processes have sent messages for which round. Only the message for the largest round number has to be stored here. (Note that declaration of saw_max in `line 3` includes ∞ lines in the matrix just for conciseness of presentation. In real implementation this could be realized using dynamic lists since information from former instantiations need not be stored.)

```
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:  /* Initialization */
5:  send (round, 0, 0) to all [once];

6:  /* Round Synchronization */
7:  if received (round, i, k) from at least n − f distinct processes
8:        →  k := k + 1;
9:           if k > Ξ
10:                →  ∀q: if saw_max[q][i] = 0 → SL[q]:= true;
11:                   k := 0;
12:                   i := i + 1;
13:                   control_overhead;
14:                   send (round, i, k) to all;       /* start next round             */
15:          else
16:                →  send (round, i, k) to all;       /* start next round             */
17:          fi
18: fi

19: /* Store received messages*/
20: if received (round, j, ℓ) from q
21:       → saw_max[q][j] := max(ℓ, saw_max[q][j]);
22: fi
```

Figure 1: Perfect Failure Detector Implementation

In order to show that our algorithm in fact implements $\mathcal{P}$ we first analyze some properties of its included software clock synchronization algorithm (`line 7`) where k is the local clock value. Since messages for different instantiations do not interfere logically with each other, we just examine the clocks for one instantiation here. For conciseness we therefore suppress i in the following. We follow the analysis of [12] 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 clock synchronization properties. We start with some preliminary definitions.

Definition 3.1 (Local Clock Value). *$C_p(t)$ denotes the local clock value of a correct process p at real-time t; s_p^k, where $k \geq 0$, is the sequence of real-times when p sets its local clock value to $k+1$.*

Definition 3.2 (Maximum Local Clock Value). *$C_{max}(t)$ denotes the maximum of all local clock values of correct processes at real-time t. Further, let $s_{first}^k = s_p^k \leq t$ be the real-time when the first correct process p sets its local clock to $k+1 = C_{max}(t)$.*

Theorem 3.3 (Properties). *In the presence of $f < n$ faults, the algorithm given in Figure 1 satisfies the following properties:*

(P) Uniform Progress. *If all correct processes set their clocks to k by time t, then every process sets its clock at least to $k+1$ by time $t+\Delta$.*

(U) Uniform Unforgeability. *If no process sets its clock to k by time t, then no process sets its clock to $k+1$ by time $t+\delta$ or earlier.*

(S) Uniform Simultaneity. *If some process sets its clock to k at time t, then every process sets its clock at least to k by time $t+\varepsilon$.*

Proof. Uniform Progress. By assumption at least $n-f$ correct processes send $(round, k)$ by time t. These messages are received by time $t+\Delta$ by all processes (that are still alive) which then set their clocks to $k+1$ by `line 7` if they have not already done so.

Uniform Unforgeability. Assume by contradiction that a process sets its clock to $k+1$ by time $t+\delta$. Thus the process has received at least $n-f$ $(round, k)$ messages. These messages must have been sent by t. Since processes send messages only for ticks less or equal their clock value we derive a contradiction.

Uniform Simultaneity. Let process p be the first process to set its clock to k. This happens based on $n-f$ $(round, k-1)$ messages by distinct processes, which must be received by all processes (that are still alive) by $t+\varepsilon$. They then set their clocks to k by `line 7`. If process p is not the first process which sets its clock to k, then at least one correct processes has done so before at time $t' < t$. By the same reasoning as above all correct processes must set their clocks to k by time $t'+\varepsilon < t+\varepsilon$. $\square$

Lemma 3.4 (Fastest Progress). *Let the first correct process set its clock to k at time t. Then no correct process can reach a larger clock value $k' > k$ before $t + \delta(k' - k)$.*

Proof. By induction on $l = k' - k$. For $l = 1$, Lemma 3.4 is identical to (U) and therefore true. Assume that no correct process has set its clock to $k + l$ before $t + l\delta$ for some l. Thus no correct process may set its clock to $k + l + 1$ before $t + l\delta + \delta = t + \delta(l + 1)$ by (U). Hence Lemma 3.4 is true for $l + 1$ as well. □

In our analysis we will require a bound upon the maximum progress of C_{max} during a given real-time interval $[t_1, t_2]$. Lemma 3.4 can be applied for this purpose if $t_1 = s^k_{first}$, but not if t_1 is arbitrary. As in [11], we unify those cases via the following Definition 3.5.

Definition 3.5 (Synchrony). *Real-time t is in synchrony with $C_{max}(t)$ iff $t = s^k_{first}$ for some arbitrary k, as defined in Definition 3.2. Let the indicator function of non-synchrony be defined as*

$$I_{t \neq s} = I_s(t) = \begin{cases} 0 \ \textit{if } t \textit{ is in synchrony with } C_{max}(t), \\ 1 \ \textit{otherwise.} \end{cases}$$

Lemma 3.6 (Maximum Increase of C_{max} within Time Interval). *Given any two real-times $t_2 \geq t_1$, then $C_{max}(t_2) - C_{max}(t_1) \leq \lfloor \frac{t_2 - t_1}{\delta} \rfloor + I_s(t_1)$.*

Proof. Let $k = C_{max}(t_1) - 1$. We have to distinguish the two cases $s^k_{first} = t_1$ and $s^k_{first} < t_1$.

Let $s^k_{first} = t_1$ such that $I_s(t_1) = 0$. From Lemma 3.4 follows that C_{max} may increase every δ time units, hence $\lfloor \frac{t_2 - t_1}{\delta} \rfloor$ times before t_2. Since $I_s(t_1) = 0$, Lemma 3.6 is true for this case.

Now let $s^k_{first} < t_1$, such that $I_s(t_1) = 1$, and let the real-time $t' = s^{k+1}_{first} > t_1$. We can now apply Lemma 3.4 starting from time t'. Since $t_2 - t_1 > t_2 - t'$ it follows from Lemma 3.4 that C_{max} cannot increase more often than $\lfloor \frac{t_2 - t_1}{\delta} \rfloor$ times between t' and t_2. At instant t', C_{max} increases by one such that $C_{max}(t_2) - C_{max}(t_1) \leq \lfloor \frac{t_2 - t_1}{\delta} \rfloor + 1$. Since $I_s(t_1) = 1$ Lemma 3.6 is true. □

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

Definition 3.7 (Instantiation). *The start of instantiation i is defined to be the earliest real-time b_i by which $n-f$ processes have sent $(round, i, 0)$. Further, the end of instantiation i is defined to be the real-time e_i the last process sets its clock to $k > \Xi$ such that it ends its current instantiation i.*

The following corollary follows directly from (P) and Definition 3.7.

Corollary 3.8 (Instantiation Termination Time). *For all instantiations i it holds that $e_i - b_i \leq D$, where $D = (\Xi + 1)\Delta$.*

To be able to control the overhead of the FD algorithm we have to insert 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 of τ can be arbitrary (depending on the required overhead) has no influence on the correctness and need not be the same at every process.

Definition 3.9 (Intermission). *$\tau \geq 0$ is the upper bound on the timeout of the* control_overhead *call.*

Corollary 3.10 (Intermission Period). *For all instantiations i and $i+1$ it holds that $b_{i+1} - e_i \leq \tau$.*

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

Theorem 3.11 (Strong Completeness). *Let $\Xi \geq \lceil 2\Theta \rceil$. In a system with $n > f$ processes, the algorithm given in Figure 1 ensures that each process p that crashes by time b_i is suspected by all correct processes by time e_i.*

Proof. By the assumption p crashed before b_i, i.e. by `line 7` no process—including p—has sent $(round, i, 1)$ by then. By (P) it is guaranteed that all correct processes reach round $k > \Xi$ such that they execute `line 9` where they must add p to SL since it did not send $(round, i, 1)$. □

Theorem 3.12 (Strong Accuracy). *Let $\Xi \geq \lceil 2\Theta \rceil$. In a system with $n > f$ processes, the algorithm given in Figure 1 ensures that no process is suspected before it crashes.*

Proof. We have to show that Ξ is chosen sufficiently large such that every correct process which reaches a clock value $k > \Xi$ at time t has already received messages at least for tick 1 by every correct process.

Let process p be the first to set its clock to k at instant $s_p^{k-1} = s_{first}^{k-1}$; hence $k = C_{max}(s_p^{k-1})$. Let time $t' = s_p^{k-1} - (\Delta + \varepsilon)$. From Lemma 3.6, it follows that $C_{max}(t') \geq k - \lceil \frac{\Delta+\varepsilon}{\delta} + 1 \rceil = k - \lceil \frac{2\Delta}{\delta} \rceil = k - \lceil 2\Theta \rceil \geq k - \Xi \geq 1$. By (S) every correct process must have sent its $(round, i, C_{max}(t'))$ message by time $t' + \varepsilon$ (by this time every correct process must reach this instantiation by (S)), such that p must have received it by time s_p^{k-1}. Thus Ξ is chosen sufficiently large such that p will not suspect any correct process. □

To reach real-time behavior we require an upper bound on detection time. We give such a bound for crashes only 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 3.11.

Theorem 3.13 (Detection Latency). *Let $\Xi \geq \lceil 2\Theta \rceil$. In a system with $n > f$ processes, the algorithm given in Figure 1 ensures that a crash at real-time $t \geq b_0$ is detected by time $t + \tau + 2D$.*

Proof. Let process p crash at real-time t where $b_{i-1} < t \leq b_i$ for $i > 0$. By Theorem 3.11 p will be suspected by all correct processes by time e_i. We hence get a worst case detection time by an upper bound on $e_i - b_{i-1}$. This time consists of the times to terminate instantiations $i-1$ and i and the intermission interval between these two instantiations. Thus $e_i - b_{i-1} \leq 2D + \tau$. □

In the solution we presented completeness and accuracy just depend on Θ while only 'timeliness' depends on Θ and Δ. If Δ is violated but Θ still holds (which is probable in many real networks [10]) we still guarantee completeness and accuracy while just timely detection is lost. This is the best one can hope for, given Δ is violated here!

Despite Theorem 3.13 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. We do so in the following section in order to give a complete solution to the RT consensus problem for a network architecture based on Deterministic Ethernet and suitable scheduling schemes.

4 Architectural Model

In this section, we sketch out a generic architectural model of the systems under consideration. See [6] 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 Figure 2. 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. Processors act as sources of messages. There are two 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, 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.

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 service times corresponding to the four waiting queues, respectively, that is the worst-case times for

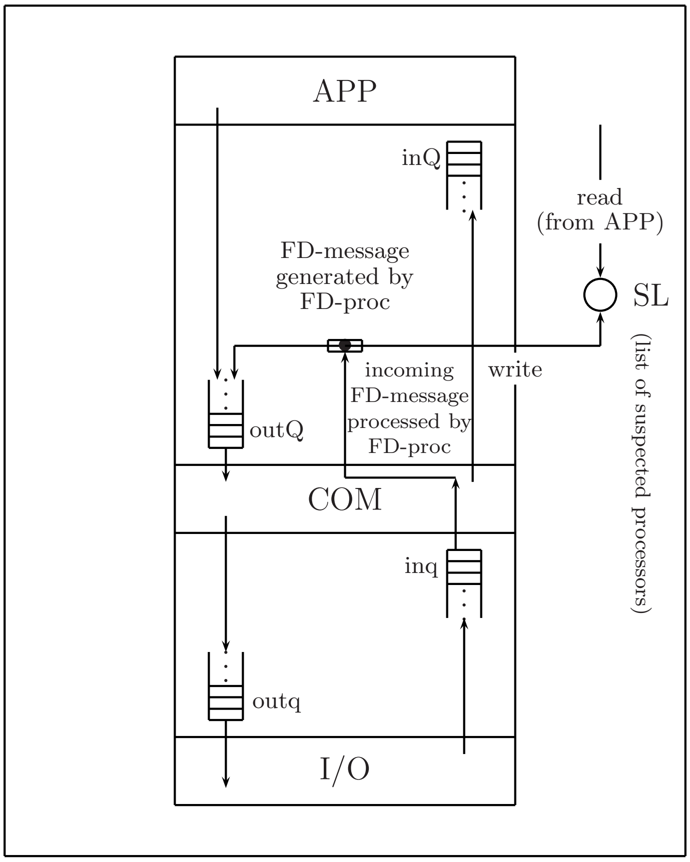

Figure 2: Architectural model of a processor

servicing a message pending in each of these waiting 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 algorithms that implement the well-known head-of-the-line policy. Consequently, we retain the following $\mathcal{S}W$ algorithm:

$\boldsymbol{\mathcal{S}W}$: 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 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:

$\boldsymbol{\mathcal{S}N}$: 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 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 $\mathcal{S}W$, $x' = 1$ if $\nu \geq w_{inq}$, $x' = \left\lceil x\left(1 - \frac{\nu}{w_{inq}}\right)\right\rceil$ if $\nu < w_{inq}$, and $\psi'(x') = x'\, w_{inq}$.

Bound $\psi(x)$ is determined by algorithms $\mathcal{S}W$ 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: $\gamma(x) = w_{outQ} + w_{outq} + d_m + \psi(x) + \theta + x'\, w_{inq}$.

In the following section, 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.

5 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, $\gamma(x) = w_{outQ} + w_{outq} + d_m + \psi(x) + x'\, w_{inq}$. Being concerned with real-time systems, we must consider a deterministic variant of the original Ethernet CSMA/CD protocol. This variant, called CSMA/DCR (Carrier Sense Multi Access/Deterministic Collision Resolution), which is based on deterministic balanced m-ary tree searches [9], has been implemented in COTS products.

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 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 ξ_x^l be the maximum number of steps needed to search x leaves in a m-ary tree of l leaves. In [5] and [4], one shows:

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

This formula applies for any assignment of x indices over l sources. 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 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 will later see how these CSMA/DCR features can be applied to our generic solution.

5.2 Behavior of the FD in our Architecture

As already discussed, our FD algorithm requires a priori knowledge of Ξ. Therefore, we examine the timing of our architecture in order to derive a value for Θ and hence Ξ. Due to the physical properties of Net, all messages are received by all processes at I/O level within σ. Due to our queuing scheme $\mathcal{S}W$, all FD-messages are taken out of *inq* by the FD-processes within σ as well. Since they all execute the same algorithms they send the same messages again within σ such that these messages—due to the queuing schemes at *outQ* and *outq*—arrive at the I/O level within σ. The first of these FD-messages triggers the jamming signal of length $log_m(l)\,\sigma > \sigma$ (see Section 5.3) such that all FD-messages for some round are transmitted during exactly one collision resolution.

When considering this behavior and our proofs in Section 3, we see that uncertainty ε for one broadcast is σ here. The behavior however is the same—compared to the system model in Section 2.1—as if $\varepsilon = 0$ and hence $\Theta = 1$ such that we can set $\Xi = 2$ here. In our architecture an instantiation is terminated when $k = 3$, i.e., we require 3 full collision resolutions for the duration of an instantiation D.

An assumption we make to reach such a behavior is that we use local timers to get a τ such that instantiations do not overlap[2]. By the assumptions that instantiations do not overlap we do not require i

[2]Since τ has no influence on correctness, this assumption is generally not required. In systems where overlapping cannot be excluded by some τ, deriving Ξ would require tedious analysis which is out of the scope of this paper since it would hide the ideas we would like to present here.

to be part of the message and for messages with $k < 3$ we just require a two-bit data field in FD-messages.

More discussions on the behavior with respect to assumption coverage can be found in Section 6.

5.3 Bounds γ and L

Jamming Sequence $\boldsymbol{\mathcal{JS}}$: 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 d_m). After such a sequence has been generated, only FD-messages are transmitted. 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 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 interarrival delay is σ. Hence, $x' = \left\lceil x\left(1 - \frac{\sigma}{w_{inq}}\right)\right\rceil$ if $\sigma < w_{inq}$, $x' = 1$ if $\sigma \geq w_{inq}$. Tight bound γ for the xth FD-message is:

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

With our Perfect FD implementation and Deterministic Ethernets, we have: $\Theta = 1$, i.e., $\Delta = \delta = \gamma$. Integer Ξ is such that: $\Xi \geq \lceil 2\Theta \rceil \geq 2$. We pick up smallest value 2 for Ξ.

From Corollary 3.8, termination time D is $D = 3\gamma$. From Theorem 3.13, detection latency L is $L = 2D+\tau$.

5.4 Numerical examples

Let us consider 10 MBit/s Ethernets. According to the ISO/OSI standard, $n \leq 1,024$ and $\sigma = 51.2\,\mu s$. We assume that the size of the longest ordinary message (I/O level framing) is 10,000 bits, i.e., $d_m = 1\,ms$. Let us pick $250\,\mu s$ 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, $x = n$. We get the following results.

- Case #1: $n = l = 16$

  $x' = 13$, $\psi(16) = \psi_1 = 1.18\ ms$, hence $\gamma(16) = \gamma_1 = 5.93\ ms$. $D_1 = 3\gamma_1 = 17.78\ ms$.

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

  $x' = 815$, $\psi(1,024) = \psi_2 = 70.14\ ms$, hence $\gamma(1,024) = \gamma_2 = 275.39\ ms$. $D_2 = 3\gamma_2 = 826.18\ ms$.

Worst-case overhead ρ 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).

- Case #1: $n = l = 16$: $\tau_1 = 292.87\ ms$ and $L_1 = 328.44\ ms$.
- Case #2: $n = l = 1,024$: $\tau_2 = 18.74246\ s$ and $L_2 = 20.39482\ s$.

Let us compare these new results with those presented in [6]. In the following illustration, we see that the price for improved coverage is measured by a ratio of **3**.

- Case #1: $n = l = 16$

[HLL02]	[HW04]
$D = 5.93\, ms$	$D = 17.78\, ms$
$\tau = 103.55\, ms$	$\tau = 292.87\, ms$
$L = 114.61\, ms$	$L = 328.44\, ms$

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

[HLL02]	[HW04]
$D = 275.39\, ms$	$D = 826.18\, ms$
$\tau = 6.52288\, s$	$\tau = 18.74246\, s$
$L = 7.07287\, s$	$L = 20.39482\, s$

6 Discussions

In our architecture we have seen that the Θ we have to consider is much better than the ratio of absolute bounds on worst-case and best-case message end-to-end delays. Since we are interested in coverage improvement we now discuss what is really required for our solution to work and which assumptions may be violated such that our solution still works: When taking a second look at our implementation in the architecture one discovers when considering the shared medium that all round k messages are on the bus before the first round $k+1$ message. From this behavior we derive $\Theta = 1$. We have seen that our assumptions are such that this behavior can be reached. Given that all processes send round k messages within σ, systems in which $w_{outQ} + w_{outq}$—the time it takes to get a round k message on the bus—at the slowest process is greater than $2w_{outQ} + 2w_{outq} + d_m + \psi(x) + x' \, w_{inq}$—the time to get the first round $k+1$ message on the bus—still reach this behavior. Our strict assumptions that w_{outQ} and w_{outq} are the same at all processes are hence not required for the correctness of the algorithm but just for timeliness. This is just one example of what kind of assumption may be violated such that safety and liveness are still ensured.

Acknowledgments

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

References

[1] Marcos K. Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. 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.

[2] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[3] Michael J. Fischer, Nancy A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty processor. *Journal of the ACM*, 32(2):374–382, April 1985.

[4] J.-F. Hermant. Quelques problèmes et solutions en ordonnancement temps réel pour systèmes répartis. *PhD thesis, Paris-VI-Pierre-et-Marie-Curie Univ.*, September 1999.

[5] J.-F. Hermant and Gérard Le Lann. A protocol and correctness proofs for real-time high-performance broadcast networks. *Proc. IEEE Int'l Conf. Distributed Computing Systems*, pages 360–369, May 1998.

[6] J.-F. Hermant and Gérard Le Lann. Fast asynchronous uniform consensus in real-time distributed systems. *IEEE Transactions on Computers*, 51(8):931–944, August 2002.

[7] Gérard 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, Springer-Verlag*, 972:51–70, September 1995.

[8] Gérard Le Lann. Proof-based system engineering and embedded systems, *Proc. European School on Embedded Systems*, invited paper. *Lecture Notes in Computer Science, Springer-Verlag*, 1494:208–248, November 1996.

[9] Gérard Le Lann and Pierre 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*, November 1984.

[10] Gérard Le Lann and Ulrich Schmid. How to implement a timer-free perfect failure detector in partially synchronous systems. Technical Report 183/1-127, Department of Automation, Technische Universität Wien, January 2003. (submitted).

[11] Ulrich Schmid and Klaus Schossmaier. Interval-based clock synchronization. *Real-Time Systems*, 12(2):173–228, March 1997.

[12] Josef Widder. Booting clock synchronization in partially synchronous systems. In *Proceedings of the 17th International Symposium on Distributed Computing (DISC'03)*, volume 2848 of *LNCS*, pages 121–135. Springer Verlag, October 2003.