Proc. Fifth European Dependable Computing Conference (EDCC-5) Budapest, Hungary

January 31, 2005

Failure Detection with Booting in Partially Synchronous Systems*

Josef Widder[1], Gérard Le Lann[2], and Ulrich Schmid[1]

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

Abstract. Unreliable failure detectors are a well known means to enrich asynchronous distributed systems with time-free semantics that allow to solve consensus in the presence of crash failures. Implementing unreliable failure detectors requires a system that provides some synchrony, typically an upper bound on end-to-end message delays. Recently, we introduced an implementation of the perfect failure detector in a novel partially synchronous model, referred to as the Θ-Model, where only the ratio Θ of maximum vs. minimum end-to-end delay of messages that are simultaneously in transit must be known a priori (while the actual delays need not be known and not even be bounded). In this paper, we present an alternative failure detector algorithm, which is based on a clock synchronization algorithm for the Θ-Model. It not only surpasses our first implementation with respect to failure detection time, but also works during the system booting phase.

1 Introduction

Asynchronous distributed algorithms maximize systems coverage, i.e., the probability that a fault-tolerant distributed real-time system works as required during its lifetime. In particular, systems coverage is known to be higher with asynchronous algorithms than with synchronous algorithms, for identical performance figures (e.g., response times) [1, 2]. Due to the well-known FLP impossibility result [3], however, important generic problems like consensus cannot be solved deterministically in purely asynchronous systems if just a single process may crash. Solving such generic problems hence requires a purely asynchronous system augmented with some semantics.

Dolev, Dwork and Stockmeyer [4] investigated how much synchronism is required in order to solve consensus. They identified five synchrony parameters (processors, communication, message order, broadcast facilities and atomicity of actions), which can be varied into 32 different *partially synchronous* models. For each of those, they investigated whether consensus is solvable. A different — more abstract — approach was taken by Chandra and Toueg [5, 6], who introduced

* Supported by the Austrian START program Y41-MAT, the BM:vit FIT-IT project *DCBA* (proj. no. 808198), and by the FWF project *Theta* (proj. no. P17757-N04).

the concept of unreliable *failure detectors* (FDs). A local failure detector module is associated with every process, which provides the consensus algorithm with hints about processes that (seem to) have crashed. Several different classes of unreliable failure detectors sufficient for solving consensus have been identified in [5]. In this paper we focus on the perfect FD $\mathcal{P}$. Informally, this FD has the semantics that (1) all correct processes will detect all crashes and (2) no processes will be falsely suspected of having crashed.

Failure detectors are particularly attractive, since they encapsulate synchrony assumptions in a time-free manner. Consensus algorithms using FDs are hence time-free in the sense that no local clocks are needed and no real-time variables show up in the algorithm's code. Therefore such algorithms share the coverage maximization property proper to purely asynchronous algorithms. Nevertheless, the question of coverage arises also when *implementing* an FD, which of course requires the underlying system to satisfy some synchrony assumptions. In fact, existing implementations of the perfect FD $\mathcal{P}$ (see [7] for a comprehensive overview of existing work) rest upon knowing an upper bound on the end-to-end transmission delays of messages and hence require a synchronous system [8].

In [7] and [2], we introduced a novel system model, referred to as Θ-Model[3], which is essentially the purely asynchronous FLP model [3] augmented with a bound upon the ratio Θ of maximum vs. minimum end-to-end computation + transmission delay between correct processes. Since just a bound upon the ratio Θ — but not on the maximum delay itself — must be known, this type of partial synchrony is not covered in the classic literature on synchrony [4, 9, 8]: The existing models require both an upper bound upon (1) the relative speed Φ of any two correct processes and (2) the absolute message transmission delay Δ. In sharp contrast, the Θ-Model does not incorporate any absolute bound on delays and works even in situations where actual delays are unbounded. (This is formalized in [10].) Unlike the global stabilization time model of [9], which assumes that the system is synchronous from some unknown point in time on, it belongs to the class of models where bounds on transmission and computation delays are unknown but are always assumed to hold.

Another issue where the Θ-Model differs from the partially synchronous models of [4, 9, 8] is the fact that it allows message-driven algorithms only, where every computation event at a process is a direct response to some (remote) message reception. In [7] we showed that spontaneous local events — e.g. due to clocks or timers — are in fact not necessary for solving generic agreement problems. Θ-algorithms are hence time-free in that they do not rely upon local time information (no clocks, no a priori bounds on the duration of computation steps) and can only make decisions based on and triggered by received messages.

In [7], we also introduced an algorithm for implementing $\mathcal{P}$ in the Θ-Model. By definition, this algorithm circumvents both the impossibility of implementing $\mathcal{P}$ in the presence of *unknown* delay bounds of [11] and the impossibility of consensus in presence of *unbounded* delays of [4], by resorting to the assumption of an a priori known ratio between largest and shortest end-to-end delays.

[3] Visit *http://www.ecs.tuwien.ac.at/projects/Theta/* for our papers on the Θ-Model.

This paper presents an alternative implementation of $\mathcal{P}$ in the Θ-Model, which surpasses the failure detector implementation [7] with respect to detection time. The new solution is based upon a clock synchronization algorithm introduced in [12, 13], which employs an extension of the non-authenticated clock synchronization algorithm by Srikanth and Toueg [14]. Among its particularly attractive features is its ability to properly handle system booting: Given that the Θ-Model allows just message-driven algorithms, Θ-algorithms must implement any required functionality without resorting to time or timers. This also constrains the solution space for the important — but often neglected — system startup problem considerably: Since our FD algorithm requires a quorum of processes in order to achieve its properties, we cannot simply assume that it works also during the booting phase, where processes get up independently of each other at unpredictable times. After all, processes that get up late typically miss at least some of the messages sent by earlier started ones.

Straightforward system startup solutions (see [12] for an overview of existing approaches) either constrain the maximum allowed duration of the booting phase via timeouts, or considerably increase the system size n for a given number of failures to be tolerated. Both deficiencies are avoided by the clock synchronization algorithm of [12], which in fact guarantees some of its properties (including precision) also during system startup. Using this algorithm, we provide an implementation of $\mathcal{P}$ for the Θ-Model which behaves like an eventually perfect failure detector $\diamond\mathcal{P}$ during system startup and becomes $\mathcal{P}$ when sufficiently many processes have completed booting.

Organization of the paper: After an overview and a short discussion of the Θ-Model in Section 2 and Section 3, respectively, we provide some required results related to the clock synchronization algorithm of [12, 13] in Section 4. Section 5 starts with the presentation of the new FD algorithm for the simplified case where all processes boot simultaneously, which is then extended to handle realistic system startup scenarios. A short discussion of our results and some remarks on coverage issues in Section 6 complete the paper.

2 System Model

We consider an asynchronous distributed system of n processes denoted by $p, q, \ldots$, which communicate through a reliable, error-free and fully connected point-to-point network. Even under the perfect communication assumption, messages that reach a process that is not booted are lost. The communication channels do not necessarily provide FIFO delivery of messages. We assume that every (non-faulty) receiver of a message knows its sender, which is actually ensured by our point-to-point assumption. Our algorithms do not require an authentication service. (Except when our algorithms have to be implemented in systems where a point-to-point network must be simulated via a shared channel like Ethernet, since authentication is required to prevent masquerading here.)

Failure model: Among the n processes, there is a maximum of f faulty ones. When considering just the clock synchronization algorithm in Section 4, no restriction is put on the behavior of faulty processes; they may exhibit Byzantine failures. Since the existing work on muteness detector specifications [15–19] suggests to also consider more severe types of failures in FD-based applications, we decided to retain the Byzantine failure model for the implementation of our FD as well. More advanced hybrid versions of our algorithm, which tolerate hybrid processor and link failures, can be found in [2, 13]. Note that the perfect communications assumption could also be dropped in favor of fair lossy links by using the simulation of reliable links proposed by Basu, Charron-Bost and Toueg [20]. Since we also investigate system startup, correct processes that have not booted yet are not counted as faulty.

Despite of using the Byzantine failure model for implementing our FD, it is nevertheless true that the classic perfect failure detector specification is only meaningful for crash failures. When our FD is used in conjunction with a classic FD-based consensus algorithm [5], the f faulty processes should hence exhibit crash failures only. Actually, even a classic FD-based consensus algorithm using our FD would also tolerate early timing failures, i.e., up to f processes that inconsistently output correct messages too early.

Computational model: Let *FD-level* be the abstraction/implementation level of our FD and the underlying clock synchronization algorithm. Following the *fast failure detector* approach of [1], this level should typically be thought of as a level fairly close to the raw computing and communication facilities. In contrast, such algorithms as consensus or atomic broadcast are typically thought of as middleware-level algorithms. Combining this with appropriate scheduling algorithms, it follows that FD-level end-to-end delays are significantly smaller than those end-to-end delays proper to consensus or atomic broadcast algorithms—typically, one order of magnitude smaller (see [1]). This feature is of particular interest with the Θ-Model [7], which is the computational model employed in this paper.

For simplicity, we employ the basic version of the Θ-Model [7] here: Assume that the FD-level end-to-end delay δ_{pq} between any two correct processes p and q satisfies $\tau^- \leq \delta_{pq} \leq \tau^+$, where the maximum and minimum $\tau^+ < \infty$ and $\tau^- > 0$, respectively, are not known a priori. δ_{pq} includes computation delays at sender and receiver, communication delays, and sojourn times in waiting queues. Note that $\tau^- > 0$ must also capture the case $p = q$ here; $\tau^+ < \infty$ secures that every message is eventually delivered. The timing uncertainty is determined by the *transmission delay uncertainty* $\varepsilon = \tau^+ - \tau^-$ and the *transmission delay ratio* $\Theta = \tau^+/\tau^-$. Note that neither τ^- nor τ^+ show up in our algorithm's code, but only Ξ, which is a function of an a priori given[4] upper bound $\bar{\Theta}$ upon Θ. (We discuss some consequences of the fact that just $\bar{\Theta}$ needs to be known a priori in Section 3.)

[4] Overbars are used for given bounds ($\bar{\Theta}$) on actual values (Θ). Such bounds must be derived from worst case and best case schedulability analyses.

In [10], it has been shown formally that for ensuring safety and liveness properties of Θ-algorithms, τ^+ and τ^- need not even be invariant, i.e., that they may vary during operation. The only requirement is that they increase and decrease together such that Θ always holds. Note that this time-variance prohibits to compute a valid upper bound on τ^+ as the product of some measured message delay and Θ, since this bound may already be invalid when it is eventually applied. A general theorem allows algorithms to be analyzed for constant τ^+ and τ^- (like it is done in this paper), however, and the results — e.g. Theorem 5 and Theorem 7 — to be translated directly into the variable timing model of [10].

Note finally that there are more advanced versions of the Θ-Model, which allow a more accurate (i.e., less pessimistic e.g. w.r.t. detection time) modeling of the behavior of our FD algorithm in real systems. Lacking space does not allow us to elaborate on those extensions here.

Model of the Initialization Phase: Initially, all correct processes are down, i.e., do not send or receive messages. Every message that arrives at a correct process while it is down is lost. A correct process decides independently when it wishes to participate in the system (or is just switched on). As faulty processes may be Byzantine, we can safely assume that faulty processes are always up or at least booted before the first correct one. Correct processes go through the following operation modes:

1. *down*: A process is down when it has not been started yet or has not completed booting.
2. *up*: A process is up if it has completed booting. To get a clean distinction of up and down, we assume that a process flushes the input queues of its network interface as first action after booting is completed. Hence, it receives messages only if they have arrived when it was up.

3 Discussion of the Θ-Model

In this section, we provide a short justification and discussion of the Θ-Model taken from [7]. Our arguments will show why a timing model where a bound on message delays is replaced by a bound on the ratio of largest and shortest end-to-end message delays makes sense for distributed fault-tolerant real-time systems.

In real systems, the end-to-end message delay δ_{pq} consists not only of physical data transmission and processing times. Rather, *queuing delays* due to the inevitable *scheduling* of the concurrent execution of multiple processes and message arrival interrupts/threads on every processor must be added to the picture. Figure 1 shows a simple queuing system model of a fully connected distributed system: All messages that drop in over one of the $n-1$ incoming links of a processor must eventually be processed by the single CPU. Every message that arrives while the CPU processes former ones must hence be put into the CPU queue for later processing. In addition, all messages produced by the CPU must be scheduled for transmission over every outgoing link. Messages that find an outgoing

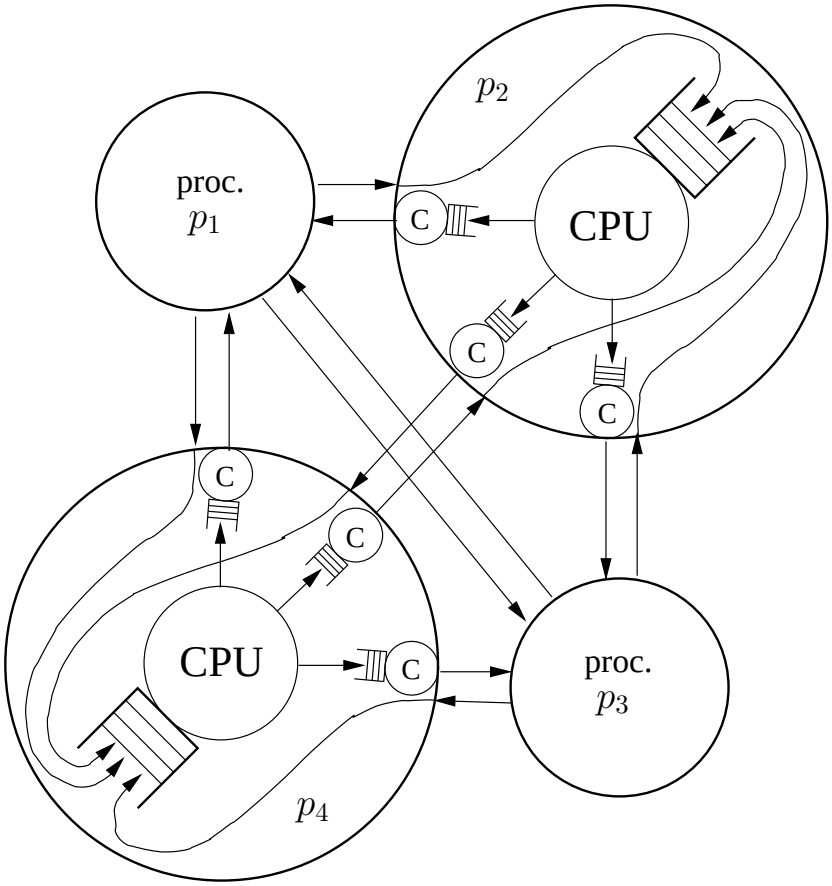

Fig. 1. A simple queuing system representation of a fully connected distributed system.

link busy must hence be put into the send queue of the link's communication controller for later transmission.

Consequently, the end-to-end delay $\delta_{pq} = d_{pq} + \omega_{pq}$ between sender p and receiver q consists of a "fixed" part d_{pq} and a "variable" part ω_{pq}. The fixed part $d_{pq} > 0$ is solely determined by the processing speeds of p and q and the data transmission characteristics (distance, speed, etc.) of the interconnecting link. It determines the minimal conceivable δ_{pq} and is easily determined from the physical characteristics of the system. The real challenge is the variable part $\omega_{pq} \geq 0$ which captures all scheduling-related variations of the end-to-end delay:

- Precedences, resource sharing and contention, with or without resource preemption, which creates waiting queues,
- Varying (application-induced) load,

– Varying process execution times (which may depend on actual values of process variables and message contents),
– Occurrence of failures.

It is apparent that ω_{pq} and thus δ_{pq} depend critically upon (1) the scheduling strategy employed (determining which message is put at which place in a queue), and (2) the particular distributed algorithm(s) executed in the system: If the usual FIFO scheduling is replaced by head-of-the-line scheduling favoring FD-level messages and computations over all application-level ones, as done with the fast failure detectors of [1], the variability of ω_{pq} at the FD-level can be decreased by orders of magnitude, see Table 1. That the queue sizes and hence the end-to-end delays δ_{pq} increase with the number and processing requirements of the messages sent by the particular distributed algorithm that is run atop of the system is immediately evident. Interestingly, however, fast FDs diminish the effect of the latter upon FD-level end-to-end delays as well, as the highest priority processing and communication activities involved with FDs are essentially "adversary immune" (the "adversary" being all activities other than FD-level related ones, in particular, application-level ones) here. This reduces both the order of magnitude of δ_{pq} and the complexity of schedulability analyses (see below) very significantly.

The above queuing system model thus reveals that the popular synchronous and partially synchronous models rest upon a very strong assumption: That an a priori given upper bound B exists which is — as part of the model — essentially independent of the particular algorithm or service under consideration, independent of the scheduling algorithm(s) used, as well as independent of the "loads" generated by algorithms or services other than the one being considered. Since the distinction between FD-level and application level is almost never made, it is almost always the case that $B \gg \bar{\tau}^+ \geq \delta_{pq}$.

In reality, such a bound B can only be determined by a detailed *worst-case schedulability analysis*[5] [21, 22]. In order to deal with all the causes of delays listed above, this schedulability analysis requires complete knowledge of the underlying system, the scheduling strategies, the failure models, the failure occurrence models and, last but not least, the particular algorithms that are to be executed in the system. Compiling B into the latter algorithms, as required by solutions that rest upon timing assumptions, hence generates a cyclic dependency. Moreover, conducting a detailed worst-case schedulability analysis for a solution that is not "adversary immune" is notoriously difficult. Almost inevitably, it rests on simplified models of reality (environments, technology) that may not always hold. As a consequence, B and hence any non time-free solution's basic assumptions might be violated at run time in certain situations.

[5] Measurement-based approaches are a posteriori solutions. Asserting a priori knowledge of an upper bound implies predictability, which is achievable only via worst-case schedulability analysis. With measurement-based approaches, the actual bounds remain unknown (even "a posteriori"), which might suffice for non-critical systems, but is out of question with many real-time embedded systems, safety-critical systems in particular.

The actual value of $\bar{\Theta}$, which obviously depends upon many system parameters, can only be determined by a detailed schedulability analysis as well. Note, however, that $\bar{\tau}^+$ has a coverage which can only be greater than the coverage of B, since our design for FDs is "adversary immune". Note also that $\bar{\tau}^-$ which is the result of a best case analysis (just d_{pq}) can be considered to have an assumption coverage higher than $\bar{\tau}^+$.

Values for $\bar{\Theta}$ depend of the physical properties of the system considered. They may range from values close to 1 in systems based on satellite broadcast channels to higher values in systems where processing and queuing delays dominate propagation delays.

To get an idea of how $\Theta(t)$ behaves in a real system ($\Theta(t)$ is the relation of longest and shortest end-to-end delays just of the messages that are simultaneously in transit at time t), we conducted some experiments [23] on a network of Linux workstation running our FD algorithm. A custom monitoring software was used to determine bounds $\bar{\tau}^- \leq \tau^-(t)$ and $\bar{\tau}^+ \geq \tau^+(t)$ as well as $\bar{\Theta} \geq \Theta(t)$ under a variety of operating conditions. The FD algorithm was run at the application level (AL-FD), as an ordinary Linux process, and as a fast failure detector (F-FD) using high-priority threads and head-of-the-line scheduling [1]. Table 1 shows some measurement data for 5 machines (with at most $f = 1$ arbitrary faulty one) under low (5 % network load) and medium (0–60 % network load, varying in steps) application-induced load.

FD	**Load**	$\bar{\tau}^-$ **(μs)**	$\bar{\tau}^+$ **(μs)**	$\bar{\Theta}$	$\frac{\bar{\tau}^+}{\bar{\tau}^-} : \bar{\Theta}$
AL-FD	low	55	15080	228.1	1.20
F-FD	low	54	648	9.5	1.26
F-FD	med	56	780	10.9	1.27

Table 1. Some typical experimental data from a small network of Linux workstations running our FD algorithm.

Our experimental data thus reveal that a considerable correlation between $\tau^+(t)$ and $\tau^-(t)$ indeed exists: The last column in Table 1 shows that $\bar{\Theta}$ is nearly 30 % smaller than $\bar{\tau}^+/\bar{\tau}^-$. Hence, when τ^+ increases, τ^- goes up to some extent as well. Note that τ^- must in fact only increase by α/Θ to compensate an increase of τ^+ by α without violating the Θ-assumption.

Informally, this correlation between τ^+ and τ^- for systems where all communication is by (real or simulated) broadcasting can be explained as follows: If some message m experiences a delay greater than τ^+, this is due to messages scheduled ahead of it in the queues along the path from sender p to q. Due to broadcast communication, those messages must also show up somewhere in the path from p to r, however. The copy of m dedicated to receiver r will hence see at least some of those messages also ahead of it. In other words, this message cannot take on the smallest possible delay value in this case, as it does not arrive

in an "empty" system. Hence, the smallest delays must be larger than τ^-, at least for some messages.

4 Clock Synchronization in the Θ-Model

In [13, 12], we introduced and analyzed a clock synchronization algorithm that can be employed in the Θ-Model which will be the core of the novel failure detector algorithm of Section 5. It assumes that every process p is equipped with an adjustable integer-valued clock $C_p(t)$, which can be read at arbitrary real-times t. The clock synchronization algorithm at p is in charge of maintaining $C_p(t)$, in a way that guarantees the following system-wide properties:

(P) *Precision:* There is some constant *precision* $D_{max} > 0$ such that

$$|C_p(t) - C_q(t)| \leq D_{max} \tag{1}$$

for any two processes p and q that are correct up to real-time t.

(A) *Accuracy:* There are some constants $R^-, O^-, R^+, O^+ > 0$ such that

$$O^-(t_2 - t_1) - R^- \leq C_p(t_2) - C_p(t_1) \leq O^+(t_2 - t_1) + R^+ \tag{2}$$

for any process p that is correct up to real-time $t_2 \geq t_1$.

Informally, (P) states that the difference of any two correct clocks in the system must be bounded, whereas (A) guarantees some relation of the progress of clock time with respect to the progress of real-time; (A) is also called *envelope requirement* in the literature. Note that (P) and (A) are *uniform* [24], in the sense that they hold also for processes that crash or become otherwise faulty later on.

Figure 2 shows a simplified version of the clock synchronization algorithm in [13].[6] Based upon the number of processes that have completed booting, two modes of operation must be distinguished here: In *degraded mode*, less than $n-f$ correct processes are up and running. Our algorithm maintains both precision (P) and the upper envelope bound (i.e., fastest progress) in (A) here for all processes. Soon after the $n - f$-th correct process gets up, the system makes the transition to *normal mode*, where it also guarantees the lower envelope (i.e., slowest progress). This holds for all processes that got synchronized (termed *active* in [13]) by executing `line 19` in Figure 2 at least once.

In this section, we review some results of our detailed analysis in [13], as far as they are required for this paper. We start with some useful definitions.

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

[6] Additionally to the algorithm, [13] presents a full analysis of the algorithm under the perception-based hybrid failure model of [25].

```
0   VAR k : integer := 0;
1
2   /* Initialization */
3   send (init, 0) to all [once];
4
5   if received (init, 0) from process p
6     → if an (echo) was already sent
7         → re-send last (echo) to p
8       else → re-send (init, 0) to p
9       fi
10  fi
11
12  if received (init, k) from at least f + 1 distinct processes
13    → send (echo, k) to all [once];
14  fi
15
16  if received (echo, k) or (echo, k + 1) from at least f + 1 distinct
      processes
17    → send (echo, k) to all [once];
18  fi
19  if received (echo, k) or (echo, k + 1) from at least n − f distinct
      processes
20    → C := k + 1; /* update clock */
21      k := k + 1;
22      send (init, k) to all [once]; /* start next round */
23  fi
24  if received (echo, ℓ) or (echo, ℓ + 1) from at least f + 1 distinct
      processes with ℓ > k
25    → C := ℓ; /* update clock */
26      k := ℓ; /* jump to new round */
27      send (echo, k) to all [once];
28  fi
```

Fig. 2. Clock Synchronization Algorithm

Definition 2 (Maximum Local Clock Value). *$C_{max}(t)$ denotes the maximum of all local clock values of correct processes that are up at real-time t. Further, let $\sigma^k_{first} = \sigma^k_p \leq t$ be the real-time when the first correct process p sets its local clock to $k + 1 = C_{max}(t)$.*

We proceed with the properties that can be guaranteed during both degraded mode and normal mode. The following Lemma 1 gives the maximum rate at which clock values of correct processes could increase.

Lemma 1 (Fastest Progress). *Let p be the first correct process that sets its clock to k at time t. Then no correct process can reach a larger clock value $k' > k$ before $t + 2\tau^-(k' - k)$.*

Theorem 1 specifies the precision D_{MCB} that is eventually achieved by all correct clocks. For processes that boot simultaneously, it holds right from the start. Late starting processes could suffer from a larger precision $D_{max} = \lfloor \Theta + 2 \rfloor$ during a short time interval (duration at most $2\tau^+$) after getting up, but are guaranteed to also reach precision D_{MCB} after that time.

Theorem 1 (Precision). *Let an arbitrary number $n_{up} \leq n$ of initially synchronized processes and/or processes that are up sufficiently long participate in the algorithm of Figure 2 for $n \geq 3f+1$. Then, $|C_p(t) - C_q(t)| \leq D_{MCB}$ for any two processes p and q that are correct up to real-time t. $D_{MCB} = \lfloor \frac{1}{2}\Theta + \frac{3}{2} \rfloor$.*

The properties stated so far can be achieved during degraded mode, with any number n_{up} of participating processes. Unfortunately, they do not guarantee progress of clock values. In normal mode, however, the following additional properties can be guaranteed:

Lemma 2 (Slowest Progress). *Let p be the last correct process that sets its clock to k at time t. In normal mode, no correct process can have a smaller clock value than $k' > k$ at time $t + 2\tau^+(k' - k)$.*

Theorem 2 (Simultaneity). *If some correct process sets its clock to k at time t, then every correct process that is up sufficiently long sets its clock at least to k by time $t + \tau^+ + \varepsilon$.*

Theorem 3 (Precision in Normal Mode). *During normal mode, the algorithm of Figure 2 satisfies $|C_p(t) - C_q(t)| \leq D'_{MCB}$ for any two processes p and q that are correct up to time t and $D'_{MCB} = \lfloor \Theta + \frac{1}{2} \rfloor$.*

Finally, the following Theorem 4 shows that normal mode is entered within bounded time after the $n - f$-th correct process got up.

Theorem 4 (Progress Into System). *Let t be the time when $n - f$ correct processes have completed booting and started the algorithm of Figure 2 for $n \geq 3f+1$. Then, normal mode is entered by time $t + 5\tau^+ + \varepsilon$ and all booted processes that are correctly up by then are within $\min\{D_{MCB}, D'_{MCB}\}$ of each other.*

5 Perfect Failure Detection in the Θ-Model

In this section, we show how to extend the clock synchronization algorithm of Section 4 in order to obtain a perfect failure detector $\mathcal{P}$. We first recall the properties that must be provided by $\mathcal{P}$ [5]:

(SC) *Strong completeness*: Eventually every process that crashes is permanently suspected by every correct process.

(SA) *Strong accuracy*: No process is suspected before it crashes.

```
0   VAR suspect[∀q] : boolean := false;
1   VAR saw_max[∀q] : integer := 0;

2   Execute Clock Synchronization from Figure 2

3   if received (init, ℓ) or (echo, ℓ) from q
4     → saw_msg[q] := max(ℓ, saw_msg[q]);
5   fi

6   whenever clock C is updated do (after updating)
7     → ∀q suspect[q] := (C − Ξ) > saw_msg[q];
```

Fig. 3. Failure Detector Implementation

Our failure detector exploits the fact that correct processes always send their clock values — via *(init, k)* or *(echo, k)* messages — to all. Due to bounded precision, a correct process p can determine a minimal clock value that it must have seen from every process by some specific time. Consequently, if appropriate messages are missing, p must have crashed. Like the algorithm of [7], our solution needs a priori knowledge of an integer constant Ξ only, which is a function of $\bar{\Theta}$ (see Theorem 5). No a priori knowledge of a bound for τ^+ is required here.

5.1 A Simple Perfect FD Algorithm

In order to properly introduce the details of our clock synchronization-based failure detector, we first ignore system startup: We assume in this subsection that all correct processes are initially up and listening simultaneously, i.e., that they cannot miss each others' messages. Faulty processes may be initially dead or may crash at arbitrary times during operation.

The algorithm given in Figure 3 is a simple extension of the clock synchronization algorithm of Figure 2; note that it could dispose of the *join*-protocol (`line 5-10` in Figure 2) since we consider simultaneous booting in this section. The first addition is the vector $saw\_max[\forall q]$ that stores, for every process q, the maximum clock tick k received via *(init, k)* or *(echo, k)*. It is written upon every message reception. Whenever a process updates its clock to k (compare `line 19` and `line 24` in Figure 2), it checks $saw\_max[\forall q]$ to find out which processes failed to send messages for tick $k-\Xi$ at least. All those processes are entered into the vector $suspect[\forall q]$, which is the interface to upper layer programs that use the failure detector module. We will now show that, if Ξ is chosen appropriately, the algorithm given in Figure 3 implements indeed the perfect failure detector.

Theorem 5. *Let* $\Xi \geq min\left\{\left\lceil \frac{3}{2}\bar{\Theta} + \frac{1}{2}\right\rceil, \left\lceil \bar{\Theta} + \frac{3}{2}\right\rceil\right\}$*, where* $\bar{\Theta}$ *is a given upper bound upon* Θ*. In a system with* $n \geq 3f + 1$ *processes, the algorithm given in Figure 3 implements the perfect failure detector.*

Proof. We have to show that (SC) and (SA) are satisfied.

For showing (SC), it is sufficient to notice that a process that crashes before it updates its clock to $\ell + 1$ will be suspected by every correct process p when p reaches clock value $k \geq \ell + \Xi$. Since progress of clock values is guaranteed by Lemma 2, every correct process will eventually reach clock value k (in systems with bounded τ^+ such clock value is reached within bounded time — the exact time bound is derived below in Theorem 6).

To prove (SA), we have to show that Ξ is chosen sufficiently large such that every correct process that reaches a clock value k at time t has already received messages for ticks at least $k - \Xi$ by every correct process. In the worst case setting, a correct process p sets its clock to k at instant $\sigma_p^{k-1} = \sigma_{first}^{k-1}$; hence $k = C_{max}(\sigma_p^{k-1})$. From Lemma 1, it follows that $C_{max}(\sigma_p^{k-1} - \tau^+) \geq k - \lceil \frac{1}{2}\Theta \rceil$. Assuming a maximum precision D_{max}, a bound for the smallest possible clock value of a correct process reads $C_{min}(\sigma_p^{k-1} - \tau^+) \geq C_{max}(\sigma_p^{k-1} - \tau^+) - D_{max} = k - \lceil \frac{1}{2}\Theta \rceil - D_{max}$. Consequently, every correct process must have sent a message for tick $C_{min}(\sigma_p^{k-1} - \tau^+)$ by time $\sigma_p^{k-1} - \tau^+$ that arrives at p by time σ_p^{k-1}. Thus, choosing $\Xi \geq \lceil \frac{1}{2}\Theta \rceil + D_{max}$ is sufficient to ensure that p does not incorrectly suspect any correct process.

Since $\lceil x \rceil + \lceil y \rceil \geq \lceil x + y \rceil$ and $\lceil -x \rceil = -\lfloor x \rfloor$, it follows from setting $x = z + w$ and $y = -w$ that $\lceil z + w \rceil \geq \lceil z \rceil + \lfloor w \rfloor$. By setting $z = \frac{1}{2}\Theta$ and $\lfloor w \rfloor = D_{max}$ we employ $\lceil z + w \rceil$ to choose Ξ. Depending on Θ, precision D'_{MCB} or D_{MCB} is smaller and can be used to calculate Ξ (by replacing D_{max}). Hence, using $D'_{MCB} = \lfloor \Theta + \frac{1}{2} \rfloor$, we get $\Xi \geq \lceil \frac{3}{2}\bar{\Theta} + \frac{1}{2} \rceil$. Performing the same calculation for $D_{MCB} = \lfloor \frac{1}{2}\Theta + \frac{3}{2} \rfloor$, we get $\Xi \geq \lceil \bar{\Theta} + \frac{3}{2} \rceil$. □

To find the worst case detection time of our FD algorithm, we have to determine how long it may take from the time a process p crashed with clock value k until all correct processes reach a clock value of $k + \Xi$ and hence suspect p. In the worst case setting, the process p with maximum clock value crashes immediately after reaching it. All other processes must first catch up to the maximum value, and then make progress for Ξ ticks until correctly suspecting p.

Theorem 6 (Detection Time). *The algorithm of Figure 3 with the value Ξ chosen according to Theorem 5 implements a perfect failure detector. Its detection time is bounded by $(2\Xi + 2)\tau^+ - \tau^-$.*

Proof. Assume the worst case: A process p is crashing at time t_c where $k = C_p(t_c) = C_{max}(t_c)$. By Theorem 2 every correct process must reach clock value k by time $t' = t_c + \tau^+ + \varepsilon$. When a correct process reaches clock value $k + \Xi$ it will suspect p. By Lemma 2 every correct process must reach that clock value by time $t = t' + 2\Xi\tau^+ = t_c + \tau^+ + \varepsilon + 2\Xi\tau^+ = t_c + (2\Xi + 2)\tau^+ - \tau^-$. □

Theorem 6 reveals that the detection time depends upon the actual τ^+, i.e., adapts automatically to the current system load. Nevertheless, following the design immersion principle [1, 26], a bound on the detection time can be computed when our algorithm is immersed in some real system. By conducting a worst-case schedulability analysis of our FD, a bound for τ^+ can be established.

5.2 A Failure Detector Algorithm with Startup

In this subsection, we will add system startup to the picture: All processes are assumed to be initially down here, and correct processes get up one after the other at arbitrary times. Faulty processes either remain down or get up before they finally crash. Note that this setting is stricter than the crash-recovery model of [27], since there are no unstable[7] processes allowed in this paper.

Recalling the semantics of $\mathcal{P}$, the properties of our clock synchronization algorithm suggest two approaches for adding system startup to our FD. First, we noted already in Section 4 that our algorithm maintains some precision $D_{max} > D_{MCB}$ during the whole system lifetime. Hence, if we based Ξ upon $D_{max} = \lfloor \Theta + 2 \rfloor$, most of the proof of Theorem 5 would apply also during system startup: (SC) is guaranteed, since progress of clock values is eventually guaranteed, namely, when normal mode is entered. The major part of the proof of (SA) is also valid, provided that D_{MCB} is replaced by D_{max}.

There is one remaining problem with this approach, however: During system startup, the resulting algorithm could suspect a correct process that simply had not started yet. When this process eventually starts, it is of course removed from the list of suspects — but this must not happen in case of the perfect failure detector. Note that not suspecting processes that never sent any message until transition to normal mode does not work either, since a process cannot reliably detect when the transition to normal mode happens. Consequently, unless the perfect FD specification is extended by the notion of "not yet started" processes, there is no hope of implementing $\mathcal{P}$ also during system startup.

The alternative is to accept degraded failure detection properties during system startup: We will show below that the failure detector of Figure 3 implements actually an *eventually perfect* failure detector $\diamond\mathcal{P}$. This FD is weaker than $\mathcal{P}$, since it assumes that there is some time t after which (SA) must hold. In our case, t is the time when the last correct process has completed booting and normal mode is entered. Nevertheless, viewed over the whole system lifetime, our FD algorithm only provides eventual semantics.

Theorem 7 (Eventually Perfect FD). *The algorithm given in Figure 3 implements a failure detector of class $\diamond\mathcal{P}$.*

Proof. We have to show that (SC) and (SA) are eventually satisfied. Let t_{up} be the time when the last correct process gets up. Theorem 4 shows that progress of clock values comes into the system and all correct processes are within precision $\min\{D_{MCB}, D'_{MCB}\}$ by time $t_{up} + 5\tau^+ + \varepsilon$. Hence, after that time, the proof of Theorem 5 applies literally and reveals that our algorithm implements $\mathcal{P}$ and hence belongs to the class $\diamond\mathcal{P}$ during the whole system lifetime. □

Theorem 7 reveals that any consensus algorithm that uses $\diamond\mathcal{P}$ under the generalized partially synchronous system model of [5] solves the booting problem if used in conjunction with our FD implementation. After all, the model of [5]

[7] Unstable processes change between up and down infinitely often.

allows arbitrary message losses to occur until the (unknown) global stabilization time GST. $\diamond\mathcal{P}$-based consensus algorithms that work with eventually reliable links can also be used immediately. Such solutions are useful even in the context of real-time systems, since we can bound the time until $\diamond\mathcal{P}$ becomes $\mathcal{P}$. Even if the consensus algorithm is designed to work with $\diamond\mathcal{P}$ it is possible to give termination times when the FD in fact provides the semantics of $\mathcal{P}$.

6 Discussion

It has been taken for granted for many years that fault-tolerant distributed real-time computing problems admit solutions designed in synchronous computational models only. Unfortunately, given the difficulty of ensuring that stipulated bounds on computation times and transmission delays are always met (which is notoriously difficult with many systems, especially those built out of COTS products), the safety/liveness/timeliness properties achieved with such systems may have a poor coverage. This holds true for any design that rests upon (some) timed semantics, including timed asynchronous systems [28] and the Timely Computing Base [29]. With such solutions, the core questions are: How do you set your timers? How do you know your response times?

Some safety properties (e.g. agreement in consensus), and liveness properties, can be guaranteed in purely asynchronous computational models, however. Since asynchronous algorithms do not depend upon timing assumptions, those properties hold regardless of the underlying system's actual timing conditions. The coverage of such time-free solutions is hence necessarily higher than that of a solution involving some timing assumptions. The apparent contradiction between time-free algorithms and timeliness properties can be resolved by following the *design immersion* principle, which was introduced in [26] and referred to as the *late binding* principle in [1]. Design immersion permits to consider time-free algorithms for implementing system or application level services in real-time systems, by enforcing that timing-related conditions (like "delay for time X") are expressed as time-free logical conditions (like "delay for x round-trips" or "delay for x events"). Safety and liveness properties can hence be proved *independently* of the timing properties of the system where the time-free algorithm will eventually be run. Timeliness properties are established only late in the design process, by conducting a worst-case schedulability analysis, when a time-free solution is immersed in a real system with its specific low-level timing properties.

Design immersion can of course be applied to FD-based consensus algorithms, which are purely asynchronous algorithms, and to our partially synchronous time-free FD algorithms (this paper, as well as [7]). Immersion of the Θ-Model into some synchronous model permits to conduct schedulability analyses strictly identical to the analysis given in [1]. The FD algorithm presented in this paper surpasses the one from [7] with respect to detection time: The former algorithm's detection time is bounded by $(3\Theta+3)\tau^+-\tau^-$, whereas our clock synchronization-based solution achieves $(2\Theta+5)\tau^+-\tau^-$. The latter is hence at least as good as

the former one, and even better when $\Theta > 2$. Note that the detection time can hence be bounded a priori if a priori bounds $\bar{\tau}^+$, $\bar{\tau}^-$ and $\bar{\Theta}$ are available.

In sharp contrast to the solution of [7], our new FD algorithm properly handles system startup as well, without requiring undue additional assumptions or an increased number of processes. Due to the uncertainty introduced by initially down correct processes, however, it can only provide the semantics of $\diamond\mathcal{P}$. In conjunction with a $\diamond\mathcal{P}$-based consensus algorithm that can handle eventually reliable links, our FD hence allows to solve consensus even during system booting. Since we can bound the time until $\diamond\mathcal{P}$ becomes $\mathcal{P}$, this solution can even be employed in real-time systems.

As said before, our failure detector has advantageous properties regarding assumption coverage. Compared to solutions where timeouts are increased during periods of high system/network load, our approach has the advantage that during overload no increase of timeout values has to be effected. This is due to the fact that, in real systems, there is typically some sufficient correlation between τ^+ and τ^- such that Θ is always maintained. Note that it suffices to have this correlation property holding at least once in a system's lifetime for making the coverage of the Θ-Model greater than the coverage of a non time-free model. (See [10] for a more detailed discussion on the Θ assumption.)

It follows that our failure detector provides the required (time-free) properties (SC) and (SA) while just the detection time possibly increases. Note carefully that the detection time is always as good as provided by the underlying system/network, i.e., the FD timing properties "emerge" naturally from the system/network capabilities [1, 26]. Moreover, if the system/network load returns to expected behavior, our algorithm is still as exact and fast as predicted, while algorithms that adapt their timeouts would have larger detection latencies then.

Since failure detection is often considered as a system service provided to application-level algorithms, the overhead of the failure detector implementation is an important issue. It might seem that this paper's algorithm induces an excessive overhead as there are always FD-level messages in transit. This is not true, however, since one must distinguish between message delay (in time units) and the throughput of a link (in messages per time unit).

A logical link in real distributed systems consists of various outbound message queues, the physical link and the inbound message queues. If a message m is in transit for δ real-time units, obviously not all resources belonging to a logical links are used by m during the whole interval δ but only at most one at a time (assuming m is in some queue, waiting for being chosen by the scheduler to be processed next, one could say no resource is allocated to m at this time); thus the overhead is never 100%. In systems with a reasonably large delay×bandwidth product, the overhead is in fact quite small.

Consider the extreme case of a satellite broadcast communication link, for example, where the end-to-end propagation delay typically is in the order of 300 ms. With up to $n = 10$ processors, 1,000 bit long FD messages, and a link throughput of 2 megabit/second, the link occupancy time for this paper's algorithm would be 5 ms per round, entailing a communication overhead smaller

than 2%. Moreover, the overhead can be reduced further by introducing local pauses between rounds (which does not even need timers as it can be done by counting suitable local events), see [2] for details.

References

1. Hermant, J.F., Le Lann, G.: Fast asynchronous uniform consensus in real-time distributed systems. IEEE Transactions on Computers **51** (2002) 931–944
2. Le Lann, G., Schmid, U.: How to maximize computing systems coverage. Technical Report 183/1-128, Department of Automation, Technische Universität Wien (2003)
3. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty processor. Journal of the ACM **32** (1985) 374–382
4. Dolev, D., Dwork, C., Stockmeyer, L.: On the minimal synchronism needed for distributed consensus. Journal of the ACM **34** (1987) 77–97
5. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM **43** (1996) 225–267
6. Chandra, T.D., Hadzilacos, V., Toueg, S.: The weakest failure detector for solving consensus. Journal of the ACM **43** (1996) 685–722
7. 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)
8. Larrea, M., Fernandez, A., Arevalo, S.: On the implementation of unreliable failure detectors in partially synchronous systems. IEEE Transactions on Computers **53** (2004) 815–828
9. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. Journal of the ACM **35** (1988) 288–323
10. Widder, J.: Distributed Computing in the Presence of Bounded Asynchrony. PhD thesis, Vienna University of Technology, Fakultät für Informatik (2004)
11. Larrea, M., Fernández, A., Arévalo, S.: On the impossibility of implementing perpetual failure detectors in partially synchronous systems. In: Proceedings of the 10th Euromicro Workshop on Parallel, Distributed and Network-based Processing (PDP'02), Gran Canaria Island, Spain (2002)
12. 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
13. Widder, J., Schmid, U.: Booting clock synchronization in partially synchronous systems with hybrid node and link failures. Technical Report 183/1-126, Department of Automation, Technische Universität Wien (2003) (submitted for publication).
14. Srikanth, T.K., Toueg, S.: Optimal clock synchronization. Journal of the ACM **34** (1987) 626–645
15. Dolev, D., Friedman, R., Keidar, I., Malkhi, D.: Failure detectors in omission failure environments. In: Proc. 16th ACM Symposium on Principles of Distributed Computing, Santa Barbara, California (1997) 286
16. Malkhi, D., Reiter, M.: Unreliable intrusion detection in distributed computations. In: Proceedings of the 10th Computer Security Foundations Workshop (CSFW97), Rockport, MA, USA (1997) 116–124
17. Kihlstrom, K.P., Moser, L.E., Melliar-Smith, P.M.: Solving consensus in a byzantine environment using an unreliable fault detector. In: Proceedings of the International Conference on Principles of Distributed Systems (OPODIS), Chantilly, France (1997) 61–75

18. Doudou, A., Garbinato, B., Guerraoui, R., Schiper, A.: Muteness failure detectors: Specification and implementation. In: Proceedings 3rd European Dependable Computing Conference (EDCC-3). Volume 1667 of LNCS 1667., Prague, Czech Republic, Springer (1999) 71–87
19. Doudou, A., Garbinato, B., Guerraoui, R.: Encapsulating failure detection: From crash to byzantine failures. In: Reliable Software Technologies - Ada-Europe 2002. LNCS 2361, Vienna, Austria, Springer (2002) 24–50
20. Basu, A., Charron-Bost, B., Toueg, S.: Simulating reliable links with unreliable links in the presence of process crashes. In Babaoglu, Ö., ed.: Distributed algorithms. Volume 1151 of Lecture Notes in Computer Science. (1996) 105–122
21. Liu, J.W.S.: Real-Time Systems. Prentice Hall (2000)
22. Stankovic, J.A., Spuri, M., Ramamritham, K., Buttazzo, G.C.: Deadline Scheduling for Real-Time Systems. Kluwer Academic Publishers (1998)
23. Albeseder, D.: Experimentelle Verifikation von Synchronitätsannahmen für Computernetzwerke. Diplomarbeit, Embedded Computing Systems Group, Technische Universität Wien (2004) (in German).
24. Hadzilacos, V., Toueg, S.: Fault-tolerant broadcasts and related problems. In Mullender, S., ed.: Distributed Systems. 2nd edn. Addison-Wesley (1993) 97–145
25. Schmid, U., Fetzer, C.: Randomized asynchronous consensus with imperfect communications. In: 22nd Symposium on Reliable Distributed Systems (SRDS'03), Florence, Italy (2003) 361–370
26. Le Lann, G.: On real-time and non real-time distributed computing. In: Proceedings 9th International Workshop on Distributed Algorithms (WDAG'95). Volume 972 of Lecture Notes in Computer Science., Le Mont-Saint-Michel, France, Springer (1995) 51–70
27. Aguilera, M.K., Chen, W., Toueg, S.: Failure detection and consensus in the crash-recovery model. Distributed Computing **13** (2000) 99–125
28. Cristian, F., Fetzer, C.: The timed asynchronous distributed system model. IEEE Transactions on Parallel and Distributed Systems **10** (1999) 642–657
29. Veríssimo, P., Casimiro, A., Fetzer, C.: The timely computing base: Timely actions in the presence of uncertain timeliness. In: Proceedings IEEE International Conference on Dependable Systems and Networks (DSN'01 / FTCS'30), New York City, USA (2000) 533–542