Research Report 28/2005, Technische Universität Wien, Institut für Technische Informatik, 2005 (replaces Tech.Rep. 183/1-127, Dept. of Automation, TU Vienna, January 2003) November 11, 2008

How to Implement a Time-Free Perfect Failure Detector in Partially Synchronous Systems

Gérard Le Lann
INRIA Rocquencourt
Domaine de Voluceau BP 105
F-78153 Le Chesnay Cedex (France)
Email: gerard.le_lann@inria.fr

Ulrich Schmid
Technische Universität Wien
Embedded Computing Systems Group E182/2
Treitlstraße 3, A-1040 Vienna (Austria)
Email: s@ecs.tuwien.ac.at

***Abstract*— This paper[1] introduces our partially synchronous Θ-Model, which is essentially the well-known FLP model augmented with a bound Θ on the ratio between the maximum and minimum end-to-end delays of messages simultaneously in transit between correct processes. We show that the Θ-Model admits the implementation of a perfect failure detector, which allows to employ classic solutions for solving important distributed computing problems like consensus. Since Θ may still hold when some assumed upper bound on the maximum delay is violated, those solutions work correctly in scenarios where synchronous implementations would fail. We show that our algorithm, which employs heartbeat messages and a timer-free timeout mechanism based upon synchronized heartbeat rounds, indeed satisfies the properties of a perfect failure detector and provides a number of attractive additional features.**

Keywords: Fault-tolerant distributed systems, partially synchronous system models, asynchronous algorithms, perfect failure detectors, coverage.

I. Introduction

It is a widespread belief that fault-tolerant distributed algorithms for real-time applications can be designed in synchronous computational models only. A more careful analysis reveals that this is not necessarily the case, however: If the targeted system is synchronous, in that it satisfies certain computing step time and transmission delays bounds, even a purely asynchronous algorithm running atop of it provides bounded response times.

Of course, compiling some postulated time bounds into an algorithm—as done in purely synchronous approaches as well as in many partially synchronous ones—makes it easier to guarantee and prove safety and liveness properties. Moreover, this usually leads to convenient programming abstractions, like the lock-step round model. Whereas *timed* algorithms hence make the system designer's life easier, they may make the system's life more difficult: Safety and liveness properties can be guaranteed only when the postulated time bounds hold true. Unfortunately, there are many run-time conditions (e.g., overloads, unanticipated excessive process execution times, failures) that may cause violations of postulated time bounds. For fail-operational systems, this is often not acceptable: An airline reservation system may occasionally be allowed to miss some response time deadlines, but may never lose consistency of replicated data or just stop.

We illustrate this by a simple example, namely, a timed algorithm for a single non-blocking message-roundtrip with some possibly faulty process q: The caller process p sends a message *ping* to q, who will send back a message *pong* when it has got *ping*. Process p shall return the received *pong* to its higher protocol layers if q has not crashed. If q has crashed, the special message NIL shall eventually be returned. Fig. 1 shows a simple solution, which is not time-free as it involves a bound[2] $\overline{\tau}^+$ (unit "seconds") on the maximum end-to-end delay of any message exchanged between correct processes. Moreover, the algorithm is not timer-free as it needs a bounded-drift local clock for correctness.

```
/* Code for process p */
send ping to q;
delay_for(2τ̄⁺); /* max. e.t.e. delay is τ̄⁺ [sec] */
if pong did not arrive then
    return(NIL)
else
    return(pong)

function delay_for(time) /* delay for time seconds */
    TIMEOUT := C(t) + time /* read clock C(t) */
    while C(t) < TIMEOUT do nothing;
return;
```

Fig. 1. *A simple timed algorithm implementing a non-blocking fault-tolerant round-trip*

Clearly, if the actual maximum delay τ^+ at run-time exceeded $\overline{\tau}^+$ for some reason, the algorithm would return NIL — although p is still alive. Since some higher-level algorithm might depend upon non-NIL round-trips with all correct processes in the system, this could lead to a violation of some safety property.

[1]This work has been supported by the Austrian START programme Y41-MAT and the FWF project P17757.

[2]Overbars are used in this paper for assumed (= a priori known) bounds on actual (= unknown) values.

Given the difficulty of computing $\overline{\tau}^+$ for a real distributed system, which requires a worst-case schedulability analysis [1], [2] taking into account all processes and resources in the system, it is obvious that a *time-free* solution, i.e., one that does not incorporate $\overline{\tau}^+$ [nor a bounded-drift clock] should achieve a higher coverage.[3] Asynchronous algorithms are a promising alternative here: Since they do not depend upon timing assumptions at all, their safety and liveness properties hold regardless of the underlying system's actual timing behavior.

The apparent contradiction between time-free algorithms and the achievement of desired timeliness properties is superficial, as mentioned earlier: It can be resolved by adhering to the *design immersion* principle, which was introduced in [3] and referred to as the *late binding* principle in [4]. Design immersion permits to consider time-free algorithms for implementing services in real-time systems, by enforcing that timing-related conditions (like "delay for X seconds") are expressed as time-free logical conditions (like "delay for x round-trips"), cp. Fig. 2 below. 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. It is only at this final stage that the specific timing properties of the underlying system are taken into account.

Unfortunately, however, many important problems in fault-tolerant distributed computing do not have deterministic solutions in the purely asynchronous model [5]. Asynchronous algorithms must hence be employed in a system that matches pure asynchrony enriched with some additional semantics for circumventing impossibility results. According to the exposition above, such added semantics should be time-free to maximize coverage. The present paper explores the following fundamental theoretical and practical question: Is there a *time-free* solution for implementing a sufficiently powerful time-free semantics? If the case, such a solution could be proved correct independently of the timing properties of the system where it is eventually run, and just immersed into the system to be fielded like any other time-free algorithm.

Accomplishments of this paper:

(1) We introduce[4] our novel partially synchronous Θ-Model, which is essentially the well-known asynchronous FLP model [5] augmented with a bounded ratio Θ of the maximum and minimum end-to-end delay of the messages simultaneously in transit between correct processes. The Θ-Model has been inspired by the results of [7], [8], which revealed that the precision of a certain clock synchronization algorithm [employed in a synchronous system] depends only upon Θ, but not on the actual delays.
Note that systems where the Θ-Model applies do exist: The Θ-assumption is quite natural for distributed systems based upon shared channels [9], and holds even in systems employing (simulated) broadcast communication over point-to-point links, as verified by an experimental evaluation of a round-based algorithm in a network of workstation interconnected by switched Ethernet technology [10], [11].

(2) We present a perfect failure detector $\mathcal{P}$ for the Θ-Model, which is time- and timer-free and hence answers the above fundamental question in the affirmative. In systems where the Θ-Model applies, the numerous existing failure detector-based distributed algorithms for achieving consensus, atomic broadcast, ... can hence be used for building robust mission-critical applications.
Our detailed analysis reveals that our failure detector indeed satisfies the properties of $\mathcal{P}$, has a well-defined detection time, and generates low and easily tuneable overhead. From a theoretical point of view, our solution allows to escape from both the impossibility of implementing $\mathcal{P}$ in presence of *unknown* delay bounds of [12] and the impossibility of consensus in presence of *unbounded* delays of [13]. This is a consequence of the correlation Θ between minimum and maximum end-to-end delay, which allows to circumvent those impossibility results.

In order to illustrate how Θ-algorithms typically exploit the Θ-assumption, we conclude this section with a simplistic Θ-implementation of the non-blocking fault-tolerant round-trip algorithm of Fig. 1.

```
/* Code for process p */
send ping to q;
delay_for(⌈Θ⌉); /* max. # (fast) RTs per (slow) RT */
if pong did not arrive then
    return(NIL)
else
    return(pong)

function delay_for(roundtrips)
    for i := 1 to roundtrips do
        send delay_ping(i) to processor r;
        wait for delay_pong(i) from r;
    od
return;
```

Fig. 2. *A simple time- and timer-free algorithm implementing a non-blocking fault-tolerant round-trip in the Θ-Model. It assumes that processor r replies with* delay_pong *whenever it receives* delay_ping.

As shown in Fig. 2, the function delay_for(roundtrips) implements a timeout by conducting additional round-trips with some processor r, which is assumed to never crash for simplicity[5] and replies by sending back *delay_pong*(i). The algorithm is correct if no more than $\lceil\Theta\rceil$ (fast) round-trips may occur between p and r during any single round-trip between p and q. This is guaranteed by the Θ-assumption, however. Note

[3]The coverage of an assertion is the probability or the likelihood that this assertion holds true in some given universe. Coverage is hence the central issue in many systems, critical systems in particular.

[4]Part of the material provided in this paper has already been presented at EDCC'05 [6].

[5]It is easy to remove this assumption in the crash failure model, by doing round-trips with $f + 1$ processors instead of just r, see [14] for a [self-stabilizing] algorithm using this idea.

that the algorithm given in Fig. 2 is purely message-driven, in that all computing steps (exept the very first one after booting) are solely triggered by message receptions.

Outline of the rest of the paper: Following a detailed relation to existing work in Section II, we introduce the Θ-Model in Section III and discuss its applicability in Section IV. Section V introduces the basic operation principle of our novel perfect failure detector. The detailed algorithm is presented and proved correct in Section VI. A summary of our accomplishments and some directions of future work in Section VII eventually conclude the paper.

II. Related Work

From a timing assumption coverage point of view, the most preferable computational model is of course the purely asynchronous model: The well-known FLP model [5] assumes a system of n processes communicating via reliable point-to-point links where processes may fail by crashing. There is no bounded-drift clock available, and both computing step times and transmission delays are non-negative, finite but unbounded. Since there are no timing assumptions considered here, nothing can be violated at runtime. Unfortunately, however, in order to solve problems like consensus, some properties must be added to the FLP model — and the stronger the added properties, the smaller the coverage in real systems.

A. Partially Synchronous Models

Several *partially synchronous* computational models [13], [15]–[20] have been proposed in the literature, which differ in the properties added to the FLP model. Most of them involve some explicit timing assumptions, but there are also a few models that incorporate timing assumptions only implicitly. We will start with a very brief survey of those models.

The familiar synchronous model may be viewed as the FLP model augmented by bounded-drift local clocks and an a priori bound Δ and σ on real-time transmission delays and computing step times, respectively. It is well-known that lock-step rounds can be simulated in this model. In the timed asynchronous model [21], the same a priori bounds are employed. However, during "unstable periods", those bounds may be violated arbitrarily; a safe state is entered when this happens. Another attempt to implement asynchronous real-time systems is the TCB approach [22], which requires a timely subsystem (the timely computing base) that provides timing failure detection and other time-related services to the asynchronous payload-part of the system. Clearly, the algorithms employed in all those approaches are not time-free.

Weaker properties are added to the FLP model in the "classic" partially synchronous [16] and semi-synchronous [17], [18] computational models: Instead of the computing step time, they only incorporate a bound Φ on the relative speed of processes, in addition to the synchronous transmission delay bound Δ. All those models allow a process to time-out messages: The semi-synchronous models [17], [18] assume that local real-time clocks are available, and the models of [16] use the computing step time of the fastest process as the units of "real-time". Hence, a spin-loop with loop-count Δ is sufficient for timing out the maximum message delay here.

Interestingly, the bounds Δ and Φ need not necessarily be known a priori to the processes, as they can be learned during the execution via incremental timeouts: In some models of [16], Φ and Δ can either be unknown, or known but hold only after some unknown *global stabilization time* GST. Note that the unknown delay model has also be investigated for shared memory systems, see e.g. [23]. A generalized partially synchronous model integrating those two models has been introduced in [19]: It assumes that relative speeds, delays and message losses are arbitrary up to GST; after GST, no message may be lost and all relative speeds and all communication delays must be smaller than the unknown upper bounds Φ and Δ, respectively.

A particularly interesting partially synchronous model in our context is the Archimedean model introduced in [15]. It adds a bounded ratio $s \geq u/m$ on the minimum computing step time m and the maximum computing step time + transmission delay u to the FLP model. Again, any process can timeout a message by means of a spin-loop with loop count s here. Note that the Archimedean model is similar to the partially synchronous models of [16] in that the minimum computing step time is used as the unit of "real-time". Moreover, since u is basically the end-to-end delay, the Archimedean model differs only in a minor (but essential) way from our Θ-Model: It uses local computing steps for timing out messages, which do not enjoy the correlation property of the Θ-Model, see Section IV.

A novel computational model that also allows to solve consensus in presence of crash failures is the finite average model (FAR-Model) introduced in [20]. The properties added to the FLP model are an unknown lower bound for the computing step time, and an unknown finite average of the round-trip delays between any pair of correct processes. Note that the latter allows round-trip delays to be increasing without bound, provided that there are sufficiently many short round-trips in between that compensate the resulting increase of the average. Due to the computing step time lower bound, any process can implement a weak bounded-drift clock (via a spin-loop), which allows to safely timeout messages by using suitable timeout-values learned at run-time. Although the algorithms in the FAR-Model are not time-free, they have of course much better coverage than synchronous ones.

B. Failure Detectors

In the seminal paper [19], it has been shown that impossibility results like [5] can also be circumvented by augmenting the FLP model with some totally time-free semantics, namely, unreliable *failure detectors* (FD's). FDs consist of a set of (low level) modules, one per processor, which can be queried locally by the higher level processes in order to obtain information about which processors have crashed. This information neither needs to be always correct nor consistent at different processes. It must be in accordance with some axiomatic (time-free) FD specification, however, which is usually made up of two parts: *Completeness*, addressing

an FD's ability to correctly suspect crashed processors, and *accuracy*, addressing an FD's ability not to incorrectly suspect non-crashed processors.

Several classes of FD specifications have been introduced in [19], which differ primarily in their accuracy requirements. The strongest one is the *perfect failure detector* $\mathcal{P}$, defined by

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

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

It has been shown in [19] and in the wealth of subsequent work that most problems in fault-tolerant distributed computing, such as consensus, atomic broadcast or leader election, can be solved in asynchronous systems augmented with $\mathcal{P}$. Note that there are FD-based consensus algorithms that do not even need *perpetual* failure detectors like $\mathcal{P}$ but can live with *eventual* FDs. An example is the eventually perfect failure detector $\diamond\mathcal{P}$, where (SA) is replaced by eventual strong accuracy ("there is a time after which correct processors are not suspected by any correct processor").

To be useful in real systems, failure detectors must of course be implemented correctly. However, an implementation of $\mathcal{P}$ in a purely asynchronous system would yield a correct consensus algorithm and hence contradict [5]. Even worse, according to [13], no consensus algorithm (and hence no FD) can cope with unbounded processing or communication delays. Finally, in [12] is has been shown that no perpetual FD like $\mathcal{P}$ can be implemented in partially synchronous systems with unknown delay bounds. In view of those impossibility results, it was taken for granted until recently that implementing perpetual failure detectors requires accurate knowledge of delay bounds and hence a synchronous system.

Still, determining correct delay bounds in distributed systems remains a difficult problem. However, [4] revealed that the required schedulability analysis is easier for a specific low-level service like a failure detector: The problem of implementing perpetual failure detectors in a synchronous system comprising n processors initially was stated as a generic real-time scheduling problem here. A *fast failure detector* algorithm was given as a function of x, which serves to dynamically recompute timeout values. By assigning specific values to x, the whole spectrum of perpetual FDs introduced in [19] could be implemented. The proposed FD achieves very low detection time [24], [25] by exploiting the fact that failure detection can be separated from the application, i.e., the atop running consensus algorithm and any other application or system dependent algorithm. The perpetual failure detector service can in fact be run as a low-level service implemented as high-priority processes that exchange high-priority FD-level messages, scheduled according to some head-of-the-line policy, for example. Delays involved in the processing and the transmission of FD-messages are hence "adversary-immune", the "adversary" being all system activities other than FD-related ones. As a consequence, fast failure detectors facilitate an accurate worst-case schedulability analysis, which provides tight bounds on the (inherently small) end-to-end delays of FD-level messages.

It turns out that fast failure detectors can also be used to actually implement the assumption that underlies the work in [26]. This paper describes a time-free implementation of $\mathcal{P}$ in systems where it can be assumed that every correct processor is connected to a set of $f+1$ processors via links that are not among their f slowest ones. The original algorithm cannot verify whether the underlying system actually satisfies this assumption, however, and no idea of how to enforce this link property in a time-free partially synchronous model (if at all possible) was given in [26].

In sharp contrast to perpetual failure detectors, there are many papers that deal with the implementation of eventual-type FDs [19], [20], [24], [26]–[38], primarily in the generalized partially synchronous system model of [19]. However, since algorithms running atop of an eventual-type FD are usually guaranteed to terminate only after the FD became perfect, i.e., after the unknown GST and only after they have learned the unknown bounds Δ and Φ, they are not compatible with the timeliness requirements of real-time systems. For the sake of completeness, we will nevertheless survey the core ideas below.

In [19], a simple implementation of an eventually perfect failure detector $\diamond\mathcal{P}$ for the generalized partially synchronous model was given. It is based upon monitoring periodic "I am alive"-messages using adaptive (increasing) timeouts at all receiver processors. Starting from an a priori given initial value, the timeout value is increased every time a false suspicion is detected (which occurs when an "I am alive"-message from a suspected processor drops in). By restricting the recipients of "I am alive"-messages from all processors to suitably chosen subsets, a less costly implementation of an eventually strong failure detector $\diamond\mathcal{S}$ was derived in [32].

Alternative FD implementations, which use polling by means of ping/reply roundtrips instead of "I am alive"-messages, have also been proposed for partially synchronous systems. The message-efficient algorithms of [30], [33] use a logical ring, where processors poll only their neighbors and use an adaptive (increasing) timeout for generating suspicions. A similar technique is used in the adaptive failure detection protocol implementing $\diamond\mathcal{P}$ in systems with finite average delays [36]. It uses piggybacking of FD messages upon application messages in order to reduce the message load.

A major deficiency of most existing adaptive timeout approaches is their inability to also decrease the timeout value, cp. [32]. Obviously, such solutions cannot adapt to (slowly) varying delays over time. Somewhat an exception is the eventually perfect failure detector for the FAR-Model proposed in [20], which adapts the timeout-value according to the measured average round-trip times. Generally, stochastic delay estimation techniques as in [24], [37] could be used if one accepts decreased performance w.r.t. accuracy. QoS aspects—as well as scalability and overall system load—in a probabilistic framework are addressed in [35].

A different type of unreliable failure detectors that received considerable attention is Ω, which outputs just a single—eventually common—processor that is considered up and running. Ω also allows to solve consensus [29] and can be implemented very efficiently even in partially synchronous systems where only some links eventually respect communication delay bounds [38].

FDs with very weak specifications [28], [34] have also been proposed in order to solve distributed computing problems that are weaker than consensus: The *heartbeat failure detectors* of [34] do not output a list of suspects but rather a list of unbounded counters. Like $\diamond\mathcal{P}$, they permit to solve the important problem of quiescent reliable communication in the presence of crashes, but unlike $\diamond\mathcal{P}$, they can easily be implemented in purely asynchronous systems.

III. System Model

As in the FLP computational model [5], we consider an asynchronous distributed system of n processors connected by reliable links. There is no need to fix a particular failure model in this section; we will just assume that up to f processors may be faulty in some way. Let δ_{pq} denote the end-to-end delay of a message sent from some correct processor p to some correct processor q, which also includes the execution of the (failure detector) algorithm at both p and q.

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 3 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 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),

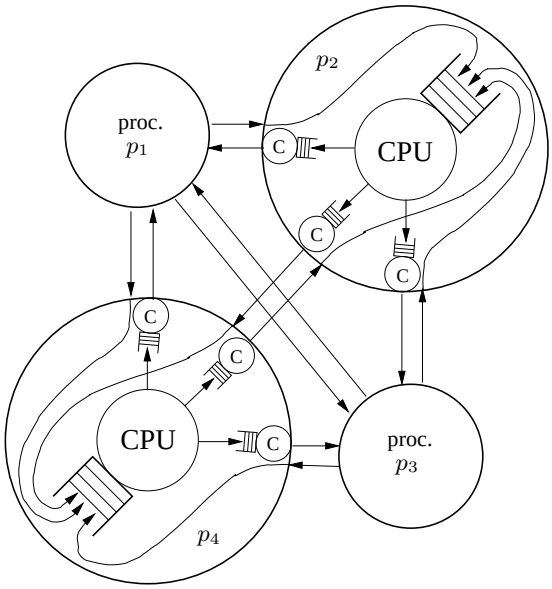

Fig. 3. A simple queuing system representation of a fully connected distributed system.

- Occurrence of failures.

As a consequence, 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 head-of-the-line scheduling favoring FD-level messages and computations over application-level ones, as proposed in the fast failure detector approach of [4], is employed instead of uniform FIFO scheduling, for example, the variability of ω_{pq} and hence the relevant $\overline{\tau}^+$ can be decreased by orders of magnitude, cp. Table I. That all queue sizes and hence end-to-end delays δ_{pq} increase with the number and processing requirements of the messages sent by the particular distributed algorithm is immediately evident.

Consequently, a safe a priori upper bound $\overline{\tau}^+$ for δ_{pq}, as required by (partially) synchronous models, can only be computed by conducting a detailed *worst-case schedulability analysis*. This schedulability analysis requires detailed 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. In the general case [1], [2], those analyses are prohibitively complex. Any $\overline{\tau}^+$ computed via a simplified[6] schedulability analysis is overly conservative or has a poor coverage, however.

[6]In the case of fast failure detectors [4], the complexity of the schedulability analysis is considerably reduced a priori. Hence, $\overline{\tau}^+$ may have some high coverage in this particular case. Nevertheless, there is always a non-zero probability that $\overline{\tau}^+$ is violated at runtime.

A. The Θ-Model

In sharp contrast to (partially) synchronous models, the Θ-Model does not need an a priori timing bound $\overline{\tau}^+$. Rather, it builds upon the relevant[7] messages $M(t)$ in transit at some point in time t, in particular, the *ratio* $\Theta(t) = \tau^+(t)/\tau^-(t)$ of their unknown minimum $\tau^-(t)$ resp. maximum $\tau^+(t)$ end-to-end computational, queueing + transmission delays: It just requires that $\Theta(t)$ always respects a given[8] a priori upper bound $\overline{\Theta}$. Note carefully that a bound on the end-to-end delay ratio does not imply bounds on the individual ratio of processing speeds and communication delays, respectively, as required e.g. by the classic partially synchronous models of [16].

Definition 1 (Θ-Model): An asynchronous system complies to the Θ-Model if there is an a priori given bound $\overline{\Theta}$ such that $\Theta(t) = \tau^+(t)/\tau^-(t) \leq \overline{\Theta}$ for all relevant messages $M(t)$ being in transit at time t.

A simple approach for determining $\overline{\Theta}$ in practice is to compute it from $\overline{\tau}^+$ and $\overline{\tau}^-$: In a real distributed system, there is always[9] a lower bound $\overline{\tau}^- > 0$ on the end-to-end delays that guarantees $\tau^-(t) \geq \overline{\tau}^-$ for all t. It is easily determined, with a coverage close to 1, by a simple best case analysis [40], [41] involving the physical properties of processors and transmission links under ideal conditions, i.e., no queueing delays. In addition, an upper bound $\overline{\tau}^+ \geq \tau^+(t)$ for all t can be determined via some worst case schedulability analysis, as for synchronous systems. Obviously, setting $\overline{\Theta} = \overline{\tau}^+/\overline{\tau}^-$ guarantees Definition 1 to hold, although this choice is usually overly conservative. Typical values reported for $\overline{\tau}^+/\overline{\tau}^-$ in real-time systems research [41], [42] are in the range $2 \ldots 10$.

A better value of $\overline{\Theta}$, which obviously depends upon many system parameters, can only be determined by a detailed and non-standard Θ-schedulability analysis. As opposed to the classic worst case and best case analysis, such a Θ-schedulability analysis must consider *joint* best and worst case scenarios rather than independent ones, see Section IV.

B. Implications of the Θ-Model on Θ-Algorithms

Definition 1 implies that Θ-algorithms cannot rely upon any knowledge of timing parameters related to $\tau^+(t)$, $\tau^-(t)$ or the uncertainty $\varepsilon(t) = \tau^+(t) - \tau^-(t)$, as they can vary arbitrarily with t. Only the a priori bound $\overline{\Theta}$ on $\Theta(t)$ may appear in a Θ-algorithm's code or variables.

In particular, the time-variance of $\tau^+(t)$ and $\tau^-(t)$ rules out a straightforward approach for simulating a synchronous system in the Θ-Model: One could try to determine an upper bound $\overline{\tau}^+ = \overline{\Theta} \cdot t_{rt}/2$ on the end-to-end delays δ_{pq} by using just a single (correct) round-trip delay measurement t_{rt}, which must of course satisfy $t_{rt} \geq 2\tau^-(t')$. The bound $\overline{\tau}^+$ computed via this approach is not necessarily valid at the future time t when it is used, however, since it is based upon a round-trip taken at some past time $t' < t$. Hence, in general, it is impossible to reliably compute a message timeout value for some future real-time out of present measurements.

Although $\tau^+(t) \equiv \tau^+$, $\tau^-(t) \equiv \tau^-$, $\varepsilon = \tau^+ - \tau^-$ and $\Theta(t) \equiv \Theta$ cannot appear in a Θ-algorithm, we will nevertheless use those quantities as (unvalued) variables in the *analysis* of such algorithms. Note carefully that those values will be considered as constants in the sequel, i.e., time-invariant, in order to simplify our presentation and proofs. Surprisingly, this can be done without loss of generality: By introducing a suitably defined notion of "stretched" real-time $t' = g(t)$, time-varying quantities w.r.t. t can be transformed in constant quantities w.r.t. t'. The inverse function $g^{[-1]}(t')$ may then be used to translate the results back to the real-time domain t again. Hence, the simplified analysis conducted in this paper is in fact sufficient for dealing with the time-varying case as well. The formal proofs can be found in [39].

Θ-algorithms typically employ continuous round-trips, cp. Fig. 2. Hence, one may be concerned about the system load generated by Θ-algorithms in general: Could it be that the generated messages exhaust all computational and communication resources? For systems with some reasonably large bandwidth * delay product, the answer is no: Consider the extreme case of a satellite broadcast communication link, for example, where the end-to-end communication 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 percent.

For systems with a lower bandwidth * delay product, on the other hand, it is possible to use some (non-bounded-drift) local "delay timer" for introducing pauses in between consecutive round-trips, see Section VI for an example. Note that those "delay timers", which can easily be implemented without hardware clocks (e.g. by counting suitable computing events), are not critical w.r.t. correctness but control the generated traffic only, cp. also [4].

IV. Justification of the Θ-Model

The purpose of this section is to show that the Θ-assumption makes sense for certain classes of distributed systems. More specifically, we will provide evidence that there is indeed some correlation between the maximum $\tau^+(t)$ and minimum $\tau^-(t)$ end-to-end delay in systems that employ broadcast-type communication.

The stipulated correlation is almost immediately evident in case of a distributed system based upon a shared channel like Ethernet or CAN. It is impossible in such a system that some message experiences the minimal end-to-end delay when the channel and hence all in- and outbuffers and processors are heavily loaded (i.e., that the system is heavily loaded, except one process pair): The worst case and best case scenario for the underlying distributed real-time scheduling problem cannot occur at the same time here, see [9] for the detailed analysis.

[7] Inspired by the fast failure detector approach [4], we will restrict our attention to failure detector computations and messages in this paper.

[8] There are variants of the Θ-Model where $\overline{\Theta}$ is unknown, or known but holds after some unknown GST, as in [16], [19], see [39] for details.

[9] Stipulating $\overline{\tau}^- = 0$ would be equivalent to assuming infinite processing and networking speeds. In this case, $\overline{\tau}^+ = 0$ as well.

That there is also some correlation between $\tau^+(t)$ and $\tau^-(t)$ in systems connected via point-to-point links (or a simulation of those, like switched Ethernet) is perhaps more surprising. Nevertheless, experiments [10], [11] in a network of Pentium-based workstations under Linux, interconnected by a 100 Mbit/sec switched Ethernet using a 1 Gbit/sec backbone switch, revealed that the Θ-assumption—with a surprisingly small $\overline{\Theta}$—holds also in such "bad" system architectures if a round-based algorithm (that communicates via simulated broadcasting) is executed: A custom monitoring software was used to determine bounds $\overline{\tau}^- \leq \tau^-(t)$ and $\overline{\tau}^+ \geq \tau^+(t)$ as well as $\overline{\Theta} \geq \Theta(t)$ for a simple variant of the clock synchronization algorithm of [6] under a variety of operating conditions. In order to keep the amount of monitoring data reasonably low, an intermediate local delay (in the ms-range) was introduced between successive rounds of execution, as in the algorithm of Section VI.

The clock synchronization algorithm was run both at the application level (AppL), as an ordinary Linux process, and at system level (SysL) using high-priority threads and head-of-the-line scheduling, as in the fast failure detector approach of [4]. Table I 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. Note that neither higher loads than 60%, which is already substantial, nor mixtures of CPU and network load did change the results significantly. Detailed information on our experiments can be found in [10], [11].

FD	**Load**	τ^- (μs)	τ^+ (μs)	Θ	$\frac{\tau^+}{\tau^-}/\Theta$
AppL	low	55	15080	228.1	1.20
SysL	low	54	648	9.5	1.26
SysL	med	56	780	10.9	1.27

TABLE I

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

Our experimental data thus reveal that there is indeed a considerable correlation between $\tau^+(t)$ and $\tau^-(t)$: The last column in Table I shows that $\overline{\Theta}$ is between 20% and 27% smaller than $\overline{\tau}^+/\overline{\tau}^-$. Hence, when $\tau^+(t)$ increases, $\tau^-(t)$ goes up to some extent as well. Note carefully that if $\tau^+(t)$ exceeds $\overline{\tau}^+$ by some $\alpha > 0$, if suffices that $\tau^-(t)$ goes up by just $\alpha/\overline{\Theta}$ to maintain $\Theta(t) \leq \overline{\Theta}$. For $\overline{\Theta} = 10$, for example, $\tau^-(t)$ needs only grow by $\alpha/10$ in order to maintain $\overline{\Theta}$.

Intuitively, this correlation can be explained as follows, cp. Fig. 4: If some message m experiences a significant end-to-end delay, this is due to messages/processing scheduled ahead of it in the queues along the path from sender p to q. Since broadcast communication is used, those messages must also show up somewhere on the path from p to r at some time. In particular, the copy of m sent to receiver r will see at least some of those messages scheduled ahead of it. This is even true for some other message m' sent from s to r, which will see some of those messages scheduled ahead of it at the receiver r. Consequently, no such message can take on the smallest possible delay value here, as it does not arrive in an "empty" system.

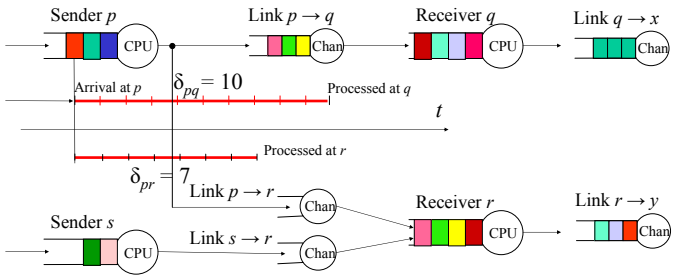

Fig. 4. *Motivation of the expected correlation between maximum and minimum end-to-end delays.*

In other words, even if the computations of interest are the highest-priority ones and messages are transmitted using head-of-the-line policies, it is usually the case that the worst case (= high-load) situation does not allow the (often quite unlikely per se) best case scenario $\delta_{rs} = \overline{\tau}^-$ to occur at some processor pair s, r simultaneously: In fact, $\overline{\tau}^-$ usually assumes that no blocking of high-priority processes by non-preemptable operations like sending a low-priority message occurs, which becomes increasingly difficult to maintain if the number of such messages increases. Consequently, the ratio of maximum over minimum delay need not grow as fast (if at all) as the maximum delay when the load increases for any reason.

Given such correlations between minimum and maximum end-to-end delays, the probability of violating $\overline{\Theta}$ is indeed smaller than the probability of violating $\overline{\tau}^+$. Consequently, a Θ-algorithm may still work correctly in scenarios where a synchronous algorithm would fail. According to our findings, this is likely or even certain to be true, depending on the particular system under consideration.

V. FD Operation Principle

In this section, we introduce the principle of operation of our time-free perfect failure detector, which is based upon the consistent broadcasting primitive of [43]. Consistent broadcasting is implemented by means of two functions, *broadcast* and *accept*, which can be used to trigger a nearly simultaneous global event in the system: For example, a processor executing the clock synchronization algorithm of [43] (see Fig. 5) invokes *broadcast* to signal that it is ready for the resynchronization event to happen. It waits for the occurrence of this resynchronization event by calling *accept*, where it blocks until the event happens. The detailed semantics of consistent broadcasting is captured by three properties, namely, correctness, unforgeability and relay, which are defined as follows:

Definition 2 (Consistent Broadcasting Properties):

(C) *Correctness*: If at least $f + 1$ correct processors call *broadcast* by time t, then *accept* is unblocked at every correct processor by time $t + \tau^+_{rt}$, for some $\tau^+_{rt} \geq 0$.

(U) *Unforgeability*: If no correct processor calls *broadcast* by time t, then *accept* cannot unblock at any correct processor by $t + \tau^-_{rt}$ or earlier, for some $\tau^-_{rt} \geq 0$.

(R) *Relay*: If *accept* is unblocked at a correct processor at time t, then every correct processor does so by time $t + \tau_\Delta$, for some $\tau_\Delta \geq 0$.

Implementations of consistent broadcasting, providing specific values of τ^+_{rt}, τ^-_{rt} and τ_Δ, can be found in [44], [45]. Fig. 6 shows the version used in our failure detector.

Now consider a very simple algorithm, which executes infinitely many instances of consistent broadcasting $R = 0, 1, \ldots$ in direct[10] succession: If *broadcast*$_0$ is called by sufficiently many processors, *accept*$_0$ will eventually unblock and trigger *broadcast*$_1$, which in turn starts the same sequence for instance 1, etc. Let σ^R_p denote the time when instance R terminates, that is, *accept*$_R$ unblocks and *broadcast*$_{R+1}$ starts at processor p. It is easy to show that the following holds for this algorithm:

Lemma 1 (Consecutive Consistent Broadcasting): Let $R \geq P \geq 0$ be two arbitrary instances of successive consecutive consistent broadcasting. Then,

$$|\sigma^R_p - \sigma^R_q| \leq \tau_\Delta \quad (1)$$

$$\sigma^R_p - \sigma^P_q \geq (R-P)\tau^-_{rt} - \tau_\Delta \quad (2)$$

$$\sigma^R_p - \sigma^P_q \leq (R-P)\tau^+_{rt} + \tau_\Delta \quad (3)$$

for any two correct processors p, q.

Proof: Equation (1) follows immediately from the relay property (R), which establishes (2) and (3) for $P = R$ as well. We can hence assume that (2) holds for $k - 1 = R - P - 1 \geq 0$ and have to show that it holds for $k = R - P$ as well. Since processor p unblocks *accept*$_R$ at time σ^R_p, at least one non-faulty processor p_1 must have called *broadcast*$_R$ by time $\sigma^R_p - \tau^-_{rt}$ by the unforgeability property (U) of consistent broadcasting. Hence $\sigma^R_p - \sigma^{R-1}_{p_1} \geq \tau^-_{rt}$. By the induction hypothesis, we have $\sigma^{R-1}_{p_1} - \sigma^P_q \geq (R-P-1)\tau^-_{rt} - \tau_\Delta$, from which (2) follows immediately.

For the upper bound, we can assume that (3) holds for $k - 1 = R - P - 1 \geq 0$ and have to show that it holds for $k = R - P$ as well. Since we know that even the last correct processor unblocks *accept*$_{R-1}$ at some time $\sigma^{R-1}_{\max}$ satisfying $\sigma^{R-1}_{\max} - \sigma^P_q \leq (R-P-1)\tau^+_{rt} + \tau_\Delta$ by the induction hypothesis, it follows from the correctness property (C) of consistent broadcasting that processor p must unblock *accept*$_R$ by time $\sigma^{R-1}_{\max} + \tau^+_{rt}$, from where (3) follows again. □

Equipped with this result, we can make the following important observation: Let us extend our simple successive consistent broadcasting algorithm by additionally broadcasting an $R+1$-*heartbeat* message (containing $R+1$ as data) when calling *broadcast*$_{R+1}$ at every processor. Now consider some correct processor p at time σ^R_p (when it sends its $R+1$-heartbeat): If p did not see a $P+1$-heartbeat, $P \leq R$, from some processor q (which should have been sent at time σ^P_q) by time σ^R_p, it must be the case that this heartbeat travels longer than $\sigma^R_p - \sigma^P_q \geq (R-P)\tau^-_{rt} - \tau_\Delta$ according to (2). Since we assumed that a heartbeat from a correct processor takes at most τ^+ to arrive, choosing

$$\Xi = R - P \geq \frac{\tau^+ + \tau_\Delta}{\tau^-_{rt}} \quad (4)$$

such that $\tau^+ \leq \Xi\tau^-_{rt} - \tau_\Delta$ ensures that p cannot miss q's $P+1$-heartbeat if q is correct. So if p did not see the $P+1$-heartbeat from q by time σ^R_p, q must have crashed and can safely be put on the list of suspects at processor p.

Of course, as it appears in (4), Ξ depends upon τ^+ and cannot be computed if this value is unknown. Our detailed analysis of consistent broadcasting in [44], [45] reveals, however, that actually $\tau^-_{rt} = 2\tau^-$ and $\tau_\Delta = 2\tau^+ - \tau^-$. Plugging this into our expression for Ξ yields $\Xi \geq 1.5\tau^+/\tau^- - 0.5$, i.e., a value that *only depends upon the ratio* $\Theta = \tau^+/\tau^-$. Since an upper bound $\overline{\Theta}$ on Θ is known, we can compute Ξ even when bounds $\overline{\tau}^+$ and $\overline{\tau}^-$ on τ^+ and τ^-, respectively, are unknown!

To further illustrate this important issue, we discuss an alternative implementation of $\mathcal{P}$, which works only in synchronous systems. The alternative algorithm is based upon the simple fact that consistent broadcasting allows to implement approximately synchronized clocks [43]. Each processor p must be equipped with a clock device $C_p(t)$ for this purpose, which is used to implement round R's clock $C^R_p(t) = C_p(t) + c^{R-1}_p$, by choosing a suitable time offset c^{R-1}_p set by the clock synchronization algorithm in round $R-1$. Note that we stipulate a round -1 clock $C^{-1}_p(t) = C_p(t)$ to be used in round 0. A single "composite" clock $C'_p(t)$ that equals the round clocks at the respective round switching times can be implemented atop of this, see [43, Sec. 7] for details.

Fig. 5 shows the simple clock synchronization algorithm of [43], which concurrently executes one instance of consistent broadcasting per resynchronization round R. Invoking *broadcast*$_R$ indicates that the calling processor is ready to resynchronize and hence starts resynchronization round R, which is done every D seconds. Note carefully that the period D measures time according to every processor's clock, hence is a parameter of the algorithm that must assume some specific value. Consistent broadcasting ensures nearly simultaneous unblocking of *accept*$_R$—and hence resynchronization—of all non-faulty processors within τ_Δ by the relay property (R). A suitably chosen offset α is used to ensure that a clock is never set backwards.

```
/* Round R ≥ 0: */
if C^{R-1}(t) = R · D /* ready to start C^R */
    → broadcast_R;
fi if accept_R
    → C^R(t) := R · D + α; /* resynchronize */
fi
```

Fig. 5. *The clock synchronization algorithm based upon consistent broadcasting from [43]*

[10]To reduce the system load caused by FD messages, some delay $D \geq 0$ is introduced in between in our final FD algorithm, see Section VI.

The proof of correctness of this algorithm only depends upon the properties of consistent broadcasting. A detailed analysis in [43], [44] yields the following worst-case synchronization precision of the composite clock $C'_p(t)$:

$$\pi_{\max} = [(D-\alpha)(1+\rho) + \tau^+_{rt}]dr + \tau_\Delta(1+\rho) + \alpha, \quad (5)$$

where ρ is the worst-case clock drift and $dr = \rho(2+\rho)/(1+\rho)$.

In order to implement a failure detector, it suffices to add the same $R+1$-heartbeat broadcast as before when *broadcast*$_{R+1}$ is called. Obviously, processor q's $P+1$-heartbeat is sent at time $(P+1)D$ on q's clock, when processor p's clock reads at most $(P+1)D+\pi_{\max}$. When this heartbeat does not arrive by the time processor p performs *broadcast*$_{R+1}$, i.e., at time $(R+1)D$ on p's clock, q must have crashed if $\Xi = R - P$ is chosen such that $(R-P)D - \pi_{\max} \geq \tau^+$ (assuming clock drift $\rho = 0$ for simplicity). With $\alpha = 0$ and $D = \tau^-_{rt}$ as the minimal conceivable[11] choice of D, we find $\pi_{\max} = \tau_\Delta$ and hence

$$\Xi = R - P \geq \frac{\tau^+ + \tau_\Delta}{\tau^-_{rt}} \quad (6)$$

exactly as in (4). Consequently, both the algorithm of Fig. 7 and the one based upon Fig. 5 can (ideally) achieve approximately the same detection time. However, apart from the fact that the correctness of the above algorithm depends upon the correct operation of a processor's clock, it also needs the parameter D for properly adjusting the periodic *broadcasts*. In order to ensure that the computed precision $\pi_{\max}$ according to (5) is valid, D must be sufficiently large to ensure that any instance of consistent broadcasting is completed before the next one starts, which implies $D \geq 2\tau^+$. Therefore, the failure detector built atop of the algorithm in Fig. 5 is not time-free and may fail if τ^+ increases beyond the limit set forth by the choice of D.

VI. The Failure Detector Algorithm

As explained in Section V, our failure detector uses consistent broadcasting for implementing a time- and timer-free timeout mechanism based upon approximately synchronized rounds. Fig. 6 shows an implementation of the pivotal consecutive consistent broadcasting primitive, which works in presence of at most f arbitrary processor failures[12] if $n \geq 3f+1$. The consistent broadcast for round R starts with sending message (init, R) to all processors. Every processor emits (echo, R) when either $f+1$ (init, R) or else $f+1$ (echo, R) messages have been received. A processor terminates ("accepts") round R upon the arrival of the $2f+1$-th (echo, R) message.

The algorithm of Fig. 6 has been derived from [43, Fig. 2] by simply replacing "*accept*" with "*delay*$_D$" followed by "*broadcast*$_{R+1}$", which starts the next instance after some delay D satisfying $D^- \leq D \leq D^+$, where $D^- \geq 0$ and, in particular, $D^+ < \infty$ can be unknown. This delay is introduced to reduce both Ξ and the system load due to FD-messages.

```
/* Implementation of broadcast_R: */
    send (init, R) to all;

/* Code for round R: */
cobegin
/* Concurrent block 1 */
if received (init, R) from f + 1 distinct processors
    → send (echo, R) to all [once]; /* sufficient evidence */
fi
/* Concurrent block 2 */
if received (echo, R) from f + 1 distinct processors
    → send (echo, R) to all [once]; /* sufficient evidence */
fi
if received (echo, R) from 2f + 1 distinct processors
    /* accept */
    → delay_D; broadcast_{R+1}; /* start next instance */
fi
coend
```

Fig. 6. *A simple consecutive consistent broadcasting algorithm with delay*

Note carefully that all instances $R = 0, 1, \ldots$ of our algorithm must be created at all processors in the system at boot time, cp. [6]. Otherwise, a processor booting late could miss some messages of processors that booted early.

Theorem 1 (Properties Consistent Broadcasting): In a system with $n \geq 3f+1$ processors, where at most f may be arbitrary faulty during the entire execution, the consistent broadcasting primitive of Fig. 6 created at boot time guarantees the properties of Definition 2 with $\tau^+_{rt} = 2\tau^+$, $\tau^-_{rt} = 2\tau^-$, and $\tau_\Delta = \varepsilon + \tau^+ \leq \tau^+_{rt}$. System-wide, at most $2n$ broadcasts of $(1 + \log_2 C_R)$-bit messages are performed by non-faulty processors per round, where C_R denotes the cardinality of the round number space (for R).

Proof: The proof has been adopted from [44] and just adds dealing with τ^- to the original one of [43].

(*Correctness*.) Since at least $f+1$ non-faulty processors broadcast (init, R) by t by assumption, every non-faulty processor p gets at least $f+1$ (init, R) from different processors by time $t+\tau^+$. It thus achieves sufficient evidence in the first **if** of Figure 6, where it sends (echo, R) to all processors. This in turn implies that every non-faulty processor q gets at least $n - f \geq 2f+1$ (echo, R) messages from different processors by $t + 2\tau^+$, which causes q to *accept* by $t + \tau^+_{rt}$ as asserted.

(*Unforgeability*.) The proof is by contradiction: Assume that there is a non-faulty processor q that *accepts* by $t + 2\tau^-$. This implies that q received $2f+1$ (echo, R) from different processors by time $t + 2\tau^-$, according to the third **if** in Figure 6. Since only at most f of those messages could originate from faulty processors, some (actually several) non-faulty processor p must also have sent (echo, R). This can

[11] Those settings are smaller than the ones required by the analysis in [43], but serve as a means to illustrate the smallest conceivable lower bound on $R - P$ only.

[12] The classic perfect failure detector specification is of course only meaningful for simple crash failures. The existing work on muteness detectors [27], [31], [46]–[48] suggests not to rule out a priori more severe types of failures, however. Note also that there is a simpler implementation for f omission faulty processors, which only needs $n \geq 2f+1$ [43]. We use the more demanding version here, since it also tolerates arbitrary processor timing failures, i.e., f processors that inconsistently emit apparently correct messages at arbitrary times. An even more advanced hybrid version of consistent broadcasting, which tolerates hybrid processor and link failures, can be found in [45].

happen due to either (1) p got at least $f+1$ (init, R) in the first **if**, or (2) at least $f+1$ (echo, R) in the third **if**, by time $t+\tau^-$. By the same argument as before, (2) requires that at least one non-faulty processor r sent (echo, R) by time t, which in turn can only happen if (1) applies to r for some time $t' \leq t+\tau^-$. For case (1) to hold, however, at least one non-faulty processor must have sent (init, R) by time t, which contradicts the assumption in the unforgeability property.

(*Relay*.) Since some non-faulty processor p *accepts* at time t, it must have received at least $2f+1$ (echo, R) from different processors by time t, according to the third **if** in Figure 6. Hence, every other non-faulty processor must get at least $f+1$ of those (echo, R) by time $t+\varepsilon$ as well, namely, the ones that have been sent by non-faulty processors. It follows that all non-faulty processors achieve sufficient evidence in the second **if** of Figure 6 by time $t+\varepsilon$ and send (echo, R) to all processors. As in the proof of correctness, this implies that every non-faulty processor r gets at least $n-f \geq 2f+1$ (echo, R) messages from different processors by $t+\varepsilon+\tau^+$, which causes q to *accept* by $t+\tau^+_{rt}$ as asserted.

As far as the claimed message complexity of consistent broadcasting is concerned, it is of course impossible to bound the number of message broadcasts by faulty processes. Each of the at most n processes that faithfully executes the algorithm of Figure 6, however, performs at most one broadcast of (init, R) and one broadcast of (echo, R) per round. Since both messages have size $1+\log_2 C_R$ bits, the expression stated in Theorem 1 follows. □

We also need a generalization of Lemma 1 that takes in to account delay_D. We assume that this function just delays starting $\mathit{broadcast}_{R+1}$ by D real-time seconds, for some (unknown) $0 \leq D^- \leq D \leq D^+ < \infty$. Let $\hat{\sigma}_p^R$ denote the time when $\mathit{broadcast}_R$ is called, and σ_p^R be the time when round R of consistent broadcasting terminates (i.e., p "accepts" and starts delay_D for the next round $R+1$). Note that obviously $D^- \leq \hat{\sigma}_p^{R+1} - \sigma_p^R \leq D^+$.

Lemma 2 (Delayed Consecutive Consistent Broadcasting): Let $R \geq P \geq 0$ be two arbitrary instances of consecutive consistent broadcasting of Figure 6 with delay_D. Then,

$$|\sigma_p^R - \sigma_q^R| \leq \tau_\Delta \quad (7)$$

$$\sigma_p^R - \hat{\sigma}_q^P \geq (R-P)(\tau_{rt}^- + D^-) + \tau_{rt}^- - \tau_\Delta \quad (8)$$

$$\sigma_p^R - \hat{\sigma}_q^P \leq (R-P)(\tau_{rt}^+ + D^+) + \tau_{rt}^+ + \tau_\Delta \quad (9)$$

for any two correct processors p, q.

Proof: Equation (7) follows immediately from the relay property (R). In addition, (8) and (9) for $P = R$ follow directly from (2) and (3) in Lemma 1 for $R - P = 1$, since $\hat{\sigma}_y^R$ here is just σ_y^{R-1} there, and the meaning of σ_x^R is the same.

We can hence assume that (8) holds for $k-1 = R-P-1 \geq 0$ and have to show that it holds for $k = R-P$ as well. Since processor p terminates round R at time σ_p^R, at least one non-faulty processor p_1 must have called delay_D (preceding $\mathit{broadcast}_R$) by time $\sigma_p^R - \tau_{rt}^- - D^-$ at latest, by the unforgeability property (U) of consistent broadcasting and our assumption of the minimum delay $D \geq D^-$ of delay_D. Hence $\sigma_p^R - \sigma_{p_1}^{R-1} \geq \tau_{rt}^- + D^-$. By the induction hypothesis, we have $\sigma_{p_1}^{R-1} - \sigma_q^P \geq (R-P-1)(\tau_{rt}^- + D^-) + \tau_{rt}^- - \tau_\Delta$, from which (8) follows immediately.

For the upper bound, we can assume that (9) holds for $k-1 = R-P-1 \geq 0$ and have to show that it holds for $k = R-P$ as well. Since we know that even the last correct processor terminates round $R-1$ at some time $\sigma_{\max}^{R-1}$ satisfying $\sigma_{\max}^{R-1} - \sigma_q^P \leq (R-P-1)(\tau_{rt}^+ + D^+) + \tau_{rt}^+ + \tau_\Delta$ by the induction hypothesis, it follows from the correctness property (C) of consistent broadcasting and the maximum delay $D \leq D^+$ of delay_D that processor p must terminate round R by time $\sigma_{\max}^{R-1} + \tau_{rt}^+ + D^+$, from where (9) follows. □

From Section V, we know that we only have to add the heartbeat-broadcast and the suspicion technique to the algorithm in Fig. 6 to obtain our timer-free failure detector as given in Fig. 7. Fortunately, we can simply use the (init, R) message as a processor's R-*heartbeat* message, since it is emitted at the time $\hat{\sigma}_p^{R-1}$ when $\mathit{broadcast}_R$ is started. Note that initializing saw_max$[\forall q] := 0$ ensures that the very first 0-heartbeat is not used for suspecting a processor. This must be avoided, since we should not assume that all correct processors start their execution within τ_Δ (which would be required to extend Lemma 2 to $R = -1$).

```
/* Global variables: */
    suspect[∀q] :=false; /* list of suspected processors */
    saw_max[∀q] := 0; /* maximum heartbeat seen */

/* Implementation start_R: */
    send (init, R) to all; /* Start heartbeat round R */

/* Code for round R: */
cobegin
/* Concurrent block 1 */
if received (init, R) from q
    → if R > saw_max[q]
        → saw_max[q] := R; /* saw new heartbeat */
      fi
fi
if received (init, R) from f + 1 distinct processors
    → send (echo, R) to all [once]; /* sufficient evidence */
fi
/* Concurrent block 2 */
if received (echo, R) from f + 1 distinct processors
    → send (echo, R) to all [once]; /* sufficient evidence */
fi
if received (echo, R) from 2f + 1 distinct processors
    → ∀q: if R − Ξ > saw_max[q]
            → suspect[q] :=true; /* q has crashed */
          fi
      delay_D; start_{R+1}; /* Start next heartbeat round */
fi
coend
```

Fig. 7. *Our time-free perfect failure detector algorithm*

The following major Theorem 2 shows that the algorithm of Fig. 7 indeed implements a perfect failure detector for systems adhering to the Θ-Model of Definition 1, and gives some performance data, like detection time[13] and message

[13]The detection time given in Theorem 2 involves the unknown τ^+. Like other timeliness properties, it can be bounded if some $\overline{\tau}^+ \geq \tau^+$ is assumed.

complexity. Recall that it is only the classic specification of the perfect failure detector $\mathcal{P}$ that forces us to consider crash failures; our implementation works even in presence of arbitrary faulty processors.

Theorem 2 (Failure Detector Properties): Let Ξ be an integer with $\Xi \geq \left\lceil \frac{3(\overline{\Theta}-1)\cdot R}{2} \right\rceil$ with $R = 2\overline{\tau}^+/(2\overline{\tau}^+ + D^-)$, where always $0 \leq R \leq \min\{2\overline{\tau}^+/D^-, 1\}$. In a system with $n \geq 3f+1$ processors, the algorithm given in Fig. 7 with all processes created at boot time implements a perfect failure detector with detection time at most $(\Xi+1)(2\tau^+ + D^+) + 4\tau^+ - \tau^-$. During $C_R \geq 1$ rounds, every correct processor has running time within $\left[C_R(2\overline{\tau}^- + D^-) - 2\overline{\tau}^+ + \overline{\tau}^-, C_R(2\overline{\tau}^+ + D^+) + 2\overline{\tau}^+ - \overline{\tau}^-\right]$, and at most $2n$ messages of size at most $(1 + \log_2 C_R)$ bits are broadcast system-wide by correct processors in every round.

Proof: We have to show that the properties (SC) and (SA) of a perfect failure detector, given in Section II, are satisfied.

As far as strong accuracy (SA) is concerned, we use the argument developed in Section V: We know that processor q emits its P-heartbeat at time $\hat{\sigma}_q^P$, and processor p checks whether it has seen the $P = R - \Xi$-heartbeat (or a later one) at time σ_p^R. Hence, if $\Xi = R - P$ is chosen such that $\sigma_p^R - \hat{\sigma}_q^P \geq \tau^+$, processor p must get the P-heartbeat unless q has crashed. Using (8) in Lemma 2 reveals that $\sigma_p^R - \hat{\sigma}_q^P \geq (R-P)(\tau_{rt}^- + D^-) + \tau_{rt}^- - \tau_\Delta$, which requires choosing $\Xi(\tau_{rt}^- + D^-) + \tau_{rt}^- - \tau_\Delta \geq \tau^+$ and hence

$$\Xi \geq \frac{\tau^+ + \tau_\Delta - \tau_{rt}^-}{\tau_{rt}^- + D^-} = \frac{3\tau^+ - 3\tau^-}{2\tau^- + D^-} = \frac{3\tau^+ - 3\tau^-}{2\tau^-} \cdot \frac{2\tau^-}{2\tau^- + D^-} \tag{10}$$

according to the latencies of consistent broadcasting given in Theorem 1. Note that the "reduction factor" $R' = 2\tau^-/(2\tau^- + D^-)$ is monotonically increasing in τ^- and obviously satisfies

$$0 \leq \frac{2\tau^-}{2\tau^- + D^-} \leq \min\{2\tau^-/D^-, 1\}. \tag{11}$$

We cannot hence plug in the lower bound $\overline{\tau}^-$ into (10) and (11) but must rather use the upper bound $\overline{\tau}^+$ here. This confirms the expressions for Ξ and R given in our theorem.

As far as strong completeness is concerned, it is obvious that a processor q that has crashed stops emitting heartbeats. Hence, eventually, every alive obedient process recognizes that the expected heartbeat is missing and suspects q. The worst case for detection time occurs when a process q crashes immediately after broadcasting its $P-1$-heartbeat, for some $P > 0$, at time $\hat{\sigma}_q^{P-1}$. At time σ_p^R with $R - P = \Xi$, processor p recognizes that the P-heartbeat from q (which should have been emitted at time $\hat{\sigma}_q^P$) is missing. According to (9) in Lemma 1, the detection time must hence be less than

$$\begin{aligned} \sigma_p^R - \hat{\sigma}_q^{P-1} &\leq (\Xi+1)(\tau_{rt}^+ + D^+) + \tau_{rt}^+ + \tau_\Delta \\ &\leq (\Xi+1)(2\tau^+ + D^+) + 4\tau^+ - \tau^- \end{aligned} \tag{12}$$

according to Theorem 1 as asserted.

The bounds for the algorithm's running time $\hat{\sigma}_p^{C_R} - \hat{\sigma}_p^0$ during $C_R \geq 1$ rounds are determined by the running times of consistent broadcasting. Plugging in $q = p$ in (8) and (9) and recalling $D^- \leq \hat{\sigma}_p^{R+1} - \sigma_p^R \leq D^+$, we obtain $\hat{\sigma}_p^{C_R} - \hat{\sigma}_p^0 \in [C_R(\tau_{rt}^- + D^-) - \tau_\Delta, C_R(\tau_{rt}^+ + D^+) + \tau_\Delta]$. The running time bounds given in our theorem, as well as the number of messages broadcast (= "send to all") and the message complexity, follow immediately from plugging in the characteristic values given by Theorem 1. □

From the running time bounds given in Theorem 2, it is apparent that *delay*$_D$ allows to arbitrarily stretch the duration of a round: A normal round has a duration within $[2\overline{\tau}^- + D^-, 2\overline{\tau}^+ + D^+]$. Consequently, increasing D^- decreases both the generated system load and Ξ. The value of D^+, on the other hand, does not affect the correctness of the algorithm in Fig. 7, but only the FD's detection time.

Our failure detector has a number of interesting additional properties, which we will now describe briefly. First, in the presence of crash failures, it guarantees that the properties (SC) and (SA) are maintained at all processors, i.e., even at faulty processors, until they crash. This is due to the fact that Theorem 1 is actually *uniform* [49], i.e., holds for processors that become faulty later on as well.

Second, the algorithm does not need to know n, except for the size of the suspect list. Still, as presented, the message size is unbounded since it incorporates R. However, solutions to this problem—construction of a bounded round numbering scheme—are known, given that Ξ is known.

Finally, the perfect failure detector algorithm of Fig. 7 could easily be transformed into an eventually perfect failure detector $\diamond\mathcal{P}$ that works also when Θ is unknown but finite, possibly after some unknown GST. All that needs to be done here is to increment Ξ every time a message from a suspected processor drops in later, see [39] for further information.

VII. Conclusions

In this paper, we showed how to implement a time- and timer-free perfect failure detector $\mathcal{P}$ for the Θ-Model, which is essentially the FLP model augmented with a bound $\overline{\Theta}$ on the ratio of maximum vs. minimum end-to-end delays of messages simultaneously in transit. Using the design immersion principle, our FD allows to build high-coverage distributed real-time systems based upon time-free (asynchronous) algorithms.

Our results have some interesting theoretical implications as well. First, our FD works in a partially synchronous system where delay bounds are unknown, apparently contradicting [12]. Second, Θ may remain bounded even when τ^+ exceeds some assumed bound $\overline{\tau}^+$. In other words, our failure detector and hence any FD-based consensus algorithm works also in presence of unbounded delays, apparently contradicting [13] as well. The apparent contradictions are due to the fact that both [12] and [13] consider τ^- and τ^+ to be uncorrelated: After all, those works assume $\tau^- = 0$. Consequently, it can be assumed in [12] that the time t until correct suspicion in the strong completeness property (SC) is independent of τ^+, since the latter is unknown. This is not true if $\overline{\Theta}$ is known, however, since our FD can infer something about the maximum delay τ^+ of messages still in transit from already received messages

via $\overline{\Theta}$. Hence, an unknown upper bound upon the transmission delay alone is not sufficient to cause the impossibility result to apply in case of Θ-systems. A similar argument applies for the impossibility of consensus in case of unbounded delays [13].

Part of our current/future work in this area is devoted to a generalized Θ-Model and the development of a framework for Θ-schedulability analysis.

VIII. Acknowledgments

We are grateful to Josef Widder and Martin Hutle for their contributions and the many stimulating discussions.

References

[1] Jane W. S. Liu, *Real-Time Systems*, Prentice Hall, 2000.

[2] John A. Stankovic, Marco Spuri, Krithi Ramamritham, and Giorgo C. Buttazzo, *Deadline Scheduling for Real-Time Systems*, Kluwer Academic Publishers, 1998.

[3] Gérard Le Lann, "On real-time and non real-time distributed computing," in *Proceedings 9th International Workshop on Distributed Algorithms (WDAG'95)*, Le Mont-Saint-Michel, France, September 1995, vol. 972 of *Lecture Notes in Computer Science*, pp. 51–70, Springer.

[4] J.-F. Hermant and Gérard Le Lann, "Fast asynchronous uniform consensus in real-time distributed systems," *IEEE Transactions on Computers*, vol. 51, no. 8, pp. 931–944, Aug. 2002.

[5] Michael J. Fischer, Nancy A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM*, vol. 32, no. 2, pp. 374–382, Apr. 1985.

[6] Josef Widder, Gérard Le Lann, and Ulrich Schmid, "Failure detection with booting in partially synchronous systems," in *Proceedings of the 5th European Dependable Computing Conference (EDCC-5)*, Budapest, Hungary, Apr. 2005, vol. 3463 of *LNCS*, pp. 20–37, Springer Verlag.

[7] Josef Widder, "Booting clock synchronization in partially synchronous systems," in *Proceedings of the 17th International Symposium on Distributed Computing (DISC'03)*, Sorrento, Italy, Oct. 2003, vol. 2848 of *LNCS*, pp. 121–135, Springer Verlag.

[8] Josef Widder and Ulrich Schmid, "Booting clock synchronization in partially synchronous systems with hybrid node and link failures," Tech. Rep. 183/1-126, Department of Automation, Technische Universität Wien, January 2003, (submitted for publication).

[9] Jean-François Hermant and Josef Widder, "Implementing time free designs for distributed real-time systems (a case study)," Research Report 23/2004, Technische Universität Wien, Institut für Technische Informatik, May 2004, Joint Research Report with INRIA Rocquencourt. (preliminary version of [50]).

[10] Daniel Albeseder, "Evaluation of message delay correlation in distributed systems," in *Proceedings of the Third Workshop on Intelligent Solutions for Embedded Systems*, Hamburg, Germany, May 2005.

[11] Daniel Albeseder, "Experimentelle Verifikation von Synchronitätsannahmen für Computernetzwerke," Diplomarbeit, Embedded Computing Systems Group, Technische Universität Wien, May 2004, (in German).

[12] Mikel Larrea, Antonio Fernández, and Sergio Arévalo, "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, Jan. 2002.

[13] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer, "On the minimal synchronism needed for distributed consensus," *Journal of the ACM*, vol. 34, no. 1, pp. 77–97, Jan. 1987.

[14] Martin Hutle and Josef Widder, "Brief announcement: On the possibility and the impossibility of message-driven self-stabilizing failure detection," in *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing (PODC'05)*, Las Vegas, Nevada, USA, July 2005.

[15] Paul M.B. Vitányi, "Time-driven algorithms for distributed control," Report CS-R8510, C.W.I., May 1985.

[16] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer, "Consensus in the presence of partial synchrony," *Journal of the ACM*, vol. 35, no. 2, pp. 288–323, Apr. 1988.

[17] Stephen Ponzio and Ray Strong, "Semisynchrony and real time," in *Proceedings of the 6th International Workshop on Distributed Algorithms (WDAG'92)*, Haifa, Israel, November 1992, pp. 120–135.

[18] Hagit Attiya, Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer, "Bounds on the time to reach agreement in the presence of timing uncertainty," *Journal of the ACM (JACM)*, vol. 41, no. 1, pp. 122–152, 1994.

[19] Tushar Deepak Chandra and Sam Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM*, vol. 43, no. 2, pp. 225–267, March 1996.

[20] Christof Fetzer and Ulrich Schmid, "Brief announcement: On the possibility of consensus in asynchronous systems with finite average response times," in *Proceedings of the 23th ACM Symposium on Principles of Distributed Computing (PODC'04)*, Boston, Massachusetts, 2004, p. 402.

[21] Flaviu Cristian and Christof Fetzer, "The timed asynchronous distributed system model," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 6, pp. 642–657, 1999.

[22] Paulo Veríssimo, António Casimiro, and Christof Fetzer, "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, pp. 533–542.

[23] Rajeev Alur, Hagit Attiya, and Gadi Taubenfeld, "Time-adaptive algorithms for synchronization," *SIAM J. Comput.*, vol. 26, no. 2, pp. 539–556, 1997.

[24] Wei Chen, Sam Toueg, and Marcos Kawazoe Aguilera, "On the quality of service of failure detectors," in *Proceedings IEEE International Conference on Dependable Systems and Networks (ICDSN / FTCS'30)*, New York City, USA, 2000.

[25] Marcos Aguilera, Gérard Le Lann, and Sam Toueg, "On the impact of fast failure detectors on real-time fault-tolerant systems," in *Proceedings of the 16th International Symposium on Distributed Computing (DISC'02)*, Toulouse, France, Oct 2002, vol. 2508 of *LNCS*, pp. 354–369, Springer Verlag.

[26] Anchour Mostefaoui, Eric Mourgaya, and Michel Raynal, "Asynchronous implementation of failure detectors," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN'03)*, San Francisco, CA, June 22–25, 2003.

[27] Kim Potter Kihlstrom, Louise E. Moser, and P. M. Melliar-Smith, "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, Dec. 1997, pp. 61–75.

[28] Vijay K. Garg and J. Roger Mitchell, "Implementable failure detectors in asynchronous systems," in *Proceedings of the 18th Int. Conference on Foundations of Software Technology and Theoretical Computer Science (FST & TCS'98)*, New-Dehli, India, 1998, LNCS 1530, pp. 158–169, Springer.

[29] Leslie Lamport, "The part-time parliament," *ACM Transactions on Computer Systems*, vol. 16, no. 2, pp. 133–169, May 1998.

[30] Mikel Larrea, Antonio Fernández, and Sergio Arévalo, "Efficient algorithms to implement unreliable failure detectors in partially synchronous systems," in *Proceedings of the 13th International Symposium on Distributed Computing (DISC'99)*, Bratislava, Slovaquia, Sept. 1999, LNCS 1693, pp. 34–48, Springer.

[31] Assia Doudou, Benoit Garbinato, Rachid Guerraoui, and André Schiper, "Muteness failure detectors: Specification and implementation," in *Proceedings 3rd European Dependable Computing Conference (EDCC-3)*, Prague, Czech Republic, September 1999, vol. 1667 of *LNCS 1667*, pp. 71–87, Springer.

[32] Mikel Larrea, Antonio Fernández, and Sergio Arévalo, "Optimal implementation of the weakest failure detector for solving consensus," in *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing (PODC'00)*, Portland, OR, USA, 2000, p. 334.

[33] M. Larrea, A. Fernandez, and S. Arevalo, "On the implementation of unreliable failure detectors in partially synchronous systems," *IEEE Transactions on Computers*, vol. 53, no. 7, pp. 815–828, 2004.

[34] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg, "On quiescent reliable communication," *SIAM Journal of Computing*, vol. 29, no. 6, pp. 2040–2073, April 2000.

[35] Indranil Gupta, Tushar D. Chandra, and Germán S. Goldszmidt, "On scalable and efficient distributed failure detectors," in *Proceedings*

of the 20th ACM Symposium on Principles of Distributed Computing (PODC'01), Newport, RI, Aug. 2001, pp. 170–179.

[36] Christof Fetzer, Michel Raynal, and Frederic Tronel, "An adaptive failure detection protocol," in *Pacific Rim International Symposium on Dependable Computing (PRDC 2001)*, Seoul, Korea, Dec. 2001.

[37] Marin Bertier, Olivier Marin, and Pierre Sens, "Implementation and performance evaluation of an adaptable failure detector," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN'02)*, Washington, DC, June 23–26, 2002, pp. 354–363.

[38] 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.

[39] Josef Widder and Ulrich Schmid, "Achieving synchrony without clocks," Research Report 49/2005, Technische Universität Wien, Institut für Technische Informatik, 2005, (submitted).

[40] Ola Redell and Martin Sanfridson, "Exact best-case response time analysis of fixed priority scheduled tasks," in *Proceedings of the 14th Euromicro Workshop on Real-Time Systems*, Vienna, Austria, June 2002, pp. 165–172.

[41] J.C. Palencia Gutiérrez, J.J. Gutiérrez Garcia, and M. Gonzáles Harbour, "Best-case analysis for improving the worst-case schedulability test for distributed hard real-time systems," in *Proceedings of the 10th Euromicro Workshop on Real-Time Systems*, Berlin, Germany, June 1998, pp. 35–44.

[42] R. Ernst and W. Ye, "Embedded program timing analysis based on path clustering and architecture classification," in *Digest of Technical Papers of IEEE/ACM International Conference on Computer-Aided Design*. Apr. 1997, pp. 598–604, IEEE Computer Society.

[43] T. K. Srikanth and Sam Toueg, "Optimal clock synchronization," *Journal of the ACM*, vol. 34, no. 3, pp. 626–645, July 1987.

[44] Ulrich Schmid, "How to model link failures: A perception-based fault model," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN'01)*, Göteborg, Sweden, July 1–4, 2001, pp. 57–66.

[45] Gérard Le Lann and Ulrich Schmid, "How to maximize computing systems coverage," Tech. Rep. 183/1-128, Department of Automation, Technische Universität Wien, April 2003.

[46] Danny Dolev, Roy Friedman, Idit Keidar, and Dahlia Malkhi, "Failure detectors in omission failure environments," in *Proc. 16th ACM Symposium on Principles of Distributed Computing*, Santa Barbara, California, 1997, p. 286.

[47] Dahlia Malkhi and Michael Reiter, "Unreliable intrusion detection in distributed computations," in *Proceedings of the 10th Computer Security Foundations Workshop (CSFW97)*, Rockport, MA, USA, June 1997, pp. 116–124.

[48] Assia Doudou, Benoit Garbinato, and Rachid Guerraoui, "Encapsulating failure detection: From crash to byzantine failures," in *Reliable Software Technologies - Ada-Europe 2002*, Vienna, Austria, June 2002, LNCS 2361, pp. 24–50, Springer.

[49] Vassos Hadzilacos and Sam Toueg, "Fault-tolerant broadcasts and related problems," in *Distributed Systems*, Sape Mullender, Ed., chapter 5, pp. 97–145. Addison-Wesley, 2nd edition, 1993.

[50] Jean-François Hermant and Josef Widder, "Implementing reliable distributed real-time systems with the Θ-Model," in *Proceedings of the 9th International Conference on Principles of Distributed Systems (OPODIS 2005)*, Pisa, Italy, Dec. 2005, LNCS, Springer Verlag, (to appear).