

The *Timed* Asynchronous System Model

Flaviu Cristian and Christof Fetzer
Dept. of Computer Science, UCSD
La Jolla, CA 92093–0114*
CSE97-519

Abstract

*We propose a formal definition for the **timed asynchronous system model**, we describe extensive measurements of actual message and process scheduling delays and hardware clock drifts that confirm that this model adequately describes current distributed systems built from networked workstations, and we give an explanation of why practically needed services, such as consensus or leader election, which are not implementable in the time-free asynchronous system model, are implementable in the timed model.*

1 Introduction

Depending on whether the underlying communication and process management services make inter-process communication by messages certain or not, distributed systems can be classified as either *synchronous* or *asynchronous* [7]. Communication certainty in a synchronous system means that any message sent by a correct process to a correct destination process is received and processed at the destination within a bounded time. This is achieved in practice by 1) using hard real-time scheduling and flow control techniques to guarantee the existence of constant upper bounds for message transmission and process scheduling delays, and by 2) assuming that the rate of failures that can occur in a system is bounded. The bounded failure rate assumption then allows system designers to use space [8] or time redundancy [24] to mask lower level communication failures and provide the abstraction of ‘certain communication’.

When no assumptions are made about the existence of bounds on communication, process scheduling delays or the rate of failures, the system is asyn-

chronous. The vast majority of distributed systems encountered in practice are asynchronous.

Most published research on asynchronous systems is based on the well-known *time-free* model [18], characterized by the following properties: 1) services are time-free: their specification describes what outputs and state transitions should occur in response to inputs without placing any bounds on the time it takes these outputs and state transitions to occur, 2) interprocess communication is reliable: any message sent between two non-crashed processes is eventually delivered to the destination process, and 3) processes have crash failure semantics, and 4) processes have no access to hardware clocks. Because in the time-free model an observer cannot distinguish between correct, slow or crashed processes, most of the services that are of importance in practice, such as consensus, election or membership, are not implementable [18, 21, 2].

Since [4], we have been using a different model for asynchronous systems, which we have called later [10] the *timed asynchronous model*, to avoid confusion with the time-free model. The timed model assumes that 1) all services are timed: their specification prescribes not only the outputs and state transitions that should occur in response to inputs, but also the time intervals within which a client can expect these outputs and transitions to occur, 2) interprocess communication is via an unreliable datagram service with omission/performance failure semantics, 3) processes have crash/performance failure semantics, 4) processes have access to hardware clocks that run within a linear envelope of real-time, and 5) no bound exists on the rate of communication and process failures that can occur in a system. Our use of this model was based on intuition that 1) it adequately describes existing distributed systems built from networked workstations and, 2) in contrast with the time-free model, the timed model allows practically needed services such as clock synchronization, membership, consensus, election, and atomic broadcast to be implemented [4, 10, 12, 6, 17].

Since it does not assume the existence of hardware

*This research was partially supported by a grant from the Air Force Office of Scientific Research. The authors can be contacted by e-mail at: flaviu,cfetzer@cs.ucsd.edu

clocks or timed services, the time-free model may appear to be more general than the timed model. However, all workstations currently on the market have high-precision quartz clocks, so the presence of clocks in the timed model is not a *practical* restriction. Moreover, while it is true that many of the services encountered in practice, such as Unix processes and UDP, do not make any response-time promises, it is also true that all such services become de facto “timed” whenever a higher level of abstraction that depends on them, in the worst case the human user, fixes a timeout for deciding of their failure. Thus, the requirement that services be timed and processes have access to hardware clocks do not make the timed model less general than the time-free model from a practical point of view. In fact, the failure semantics of interprocess communication in the time-free model is much stronger than in the timed model: while in the time-free model there cannot exist system runs in which correct processes are disconnected for the entire run, the timed model allows runs in which correct processes are permanently disconnected. Thus, while the time-free model excludes the possibility that correct processes be partitioned for arbitrary lengths of time, the timed model allows such partitioning to be naturally modeled as the occurrence of sufficiently many omission or performance communication failures. This characteristic of the timed model reflects in a very natural way the situations in which communication partitions can be observed for hours, days, or even weeks in real systems, especially those based on wide area networks, like the Internet. Thus, from a practical point of view, the timed model is more general than the time-free one, because 1) it allows partitions to be modeled naturally, and 2) its assumptions that services are timed and processes have access to hardware clocks are not restrictive from a practical point of view.

The goal of this paper is to 1) propose a formal definition for the timed asynchronous system model, 2) provide extensive measurements of actual message and process scheduling delays and clock drifts that confirm that this model adequately describes current run-of-the-mill, distributed systems built from networked workstations, and 3) give an intuitive explanation of why practically important services such as consensus or leader election, which are not implementable in the time-free asynchronous system model, are implementable in the timed model.

2 Related Work

Distributed system models can be classified according to what they assume about *network topology*,

synchrony, *failure model*, and *message buffering* [19]. According to this taxonomy, the timed asynchronous model can be characterized as follows:

- *network topology*: any process knows the complete set of processes and can send messages to any process. The problem of routing messages for irregular topologies is assumed to be solved by a lower level routing protocol.
- *synchrony*: services are timed and processes have access to local hardware clocks whose drift rates from real-time are bounded. The timed service specifications allow the definition of *timeout* delays for message transmission and process scheduling delays.
- *failure model*: processes can suffer crash or performance failures; the communication service can suffer omission or performance failures.
- *message buffering*: finite message buffers and non-FIFO delivery of messages. Buffer overflows do not block senders, but result in communication omission failures.

The timed asynchronous system model was introduced (without being named) in [4]. It was further refined in [10] and renamed to avoid confusion with the time-free model [18]. In particular, [10] introduces system *stability predicates* and *conditional timeliness* properties to capture the intuition that as long as the system is stable, that is, the number of failures affecting the system is below a certain threshold, the system will make progress within a bounded time. Well-tuned systems are expected to alternate between long periods of stability and short periods of instability, in which the failure rate raises beyond the assumed threshold. In [12] we introduced *progress assumptions* as an extension of the timed asynchronous system model: a progress assumption states that after an unstable period there always exists a time interval of bounded length in which the system will be stable. Progress assumptions allow to solve problems like consensus, that were originally specified by using unconditional termination conditions, as opposed to our use of conditional timeliness properties. To further highlight the similarities and differences which exist between the synchronous and the timed asynchronous system models, [7] compares the properties of fundamental synchronous and asynchronous services such as membership and atomic broadcast. In [16] we introduced the notion of *fail-awareness* as a systematic means of transforming synchronous service specifications into (fail-aware) specifications that are implementable in timed asynchronous systems.

Progress assumptions, which require that, infinitely often, some majority set of processes becomes “stable” for a certain amount of time, have a certain similarity to the *global stabilization* requirement of [11], which postulates that eventually a system must permanently stabilize, in the sense that there must exist a time beyond which all messages and all non-crashed processes become timely. Progress assumptions have also a certain similarity with *failure detectors* [3] which are mechanisms for adding synchrony to the time-free model: certain failure detector classes provide their desired behavior based on the observation that the system eventually stabilizes.

The *quasi-synchronous model* [23] is another approach to define a model that is in between synchronous systems and time-free asynchronous systems. It requires (P1) bounded and known processing speeds, (P2) bounded and known message delivery times, (P3) bounded and known drift rates for correct clocks, (P4) bounded and known load patterns, and (P5) bounded and known deviations among local clocks. The model allows for at least one of the properties (Px) to have a non-one *assumption coverage*, that is, a non-zero probability that the bound postulated by (Px) is violated at run-time. In comparison, the timed asynchronous system model assumes that the coverage of (P3) is 1, the coverage of (P1) and (P2) can be any value, and it does not make any assumptions about load patterns or the deviation between local clocks.

3 The Model

A timed asynchronous system consists of a finite set of processes \mathcal{P} linked by a datagram service. Processes run on the nodes of a physical network (see Figure 1). Lower level software in the nodes and the network implement the datagram service. Two processes are said to be *remote* if they run on distinct nodes, otherwise they are *local*. Each process p has access to a local hardware clock. The process management service that runs in each node uses this clock to manage alarm clocks that allow the local processes to request to be awakened whenever desired. We use $o, p, q,$ and r to denote processes, $s, t, u,$ and v to denote real-times, $S, T, U,$ and V to denote clock times, and $m,$ and n to denote messages. Subscripts are used whenever needed.

3.1 Hardware Clocks

All processes that run on a node can access the node’s *hardware clock*. Such a clock consists of an

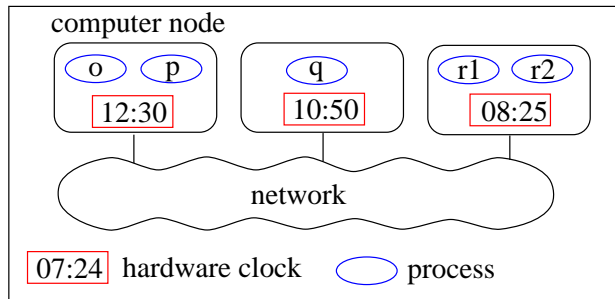


Figure 1: Processes in a timed asynchronous system have access to local hardware clocks and communicate via datagram messages across a network.

oscillator and a counting register that is incremented by the ticks of the oscillator. Each tick increments the clock value by a positive constant G , called the clock *granularity*. Correct clocks are therefore monotonically increasing. We denote \mathcal{RT} the set of real-times and \mathcal{CT} the set of clock values. The clock of process p is a function H_p from real-time to clock-time:

$$H_p : \mathcal{RT} \rightarrow \mathcal{CT}.$$

$H_p(t)$ denotes the value displayed by the clock of p at real-time t . Local processes access the same clock, while remote processes access different clocks.

Because of imprecision of the oscillator, temperature changes, and aging, a hardware clock usually drifts from real-time. The drift rate of correct clocks is bounded by a *maximum drift rate* constant ρ : a clock H_p is *correct* at time u , if for all times s and t such that $s \leq t \leq u$, H_p measured the passage of the real-time duration $t-s$ with an error of at most $\rho(t-s)+G$:

$$(1-\rho)(t-s)-G \leq H_p(t)-H_p(s) \leq (1+\rho)(t-s)+G.$$

The ρ bound on the drift rate causes any correct clock to be within a narrow linear envelope of real-time (see Figure 2).

3.1.1 Measurements

Modern operating systems, such as Solaris, provide processes access to hardware clocks that are not subject to adjustments. These clocks are ideally suited to implement a calibrated hardware clock. For example, Solaris provides a function *gethrtime* (get high resolution time) that returns a clock value expressed as nanoseconds. In particular, these clocks are not affected by erroneous operator-requested clock adjustments that can be the cause of hardware clock failures [1].

For current workstation technology, the granularity of a hardware clock is typically between $1ns$ and $1\mu s$, and the constant ρ is of the order of 10^{-4} to

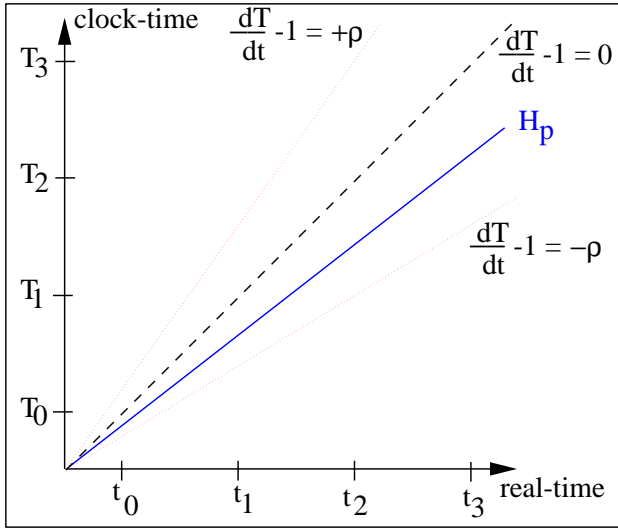


Figure 2: The drift rate of a process p 's hardware clock H_p is within $[-\rho, +\rho]$.

10^{-6} . We measured the drift rate of the uncalibrated, unsynchronized hardware clocks of the SUN workstations in our Dependable Systems Laboratory at UCSD over a period of several weeks. The Figure 3 shows the drift rate of four hardware clocks with respect to a set of externally synchronized clocks (which approximate real-time). The average drift rate of all four hardware clocks stayed constant over the measured period. The “spikes” in the graph are due to the adjustments made by the external synchronization algorithm to the set of externally synchronized clocks, they are not due to changes in the drift rate of the hardware clocks.

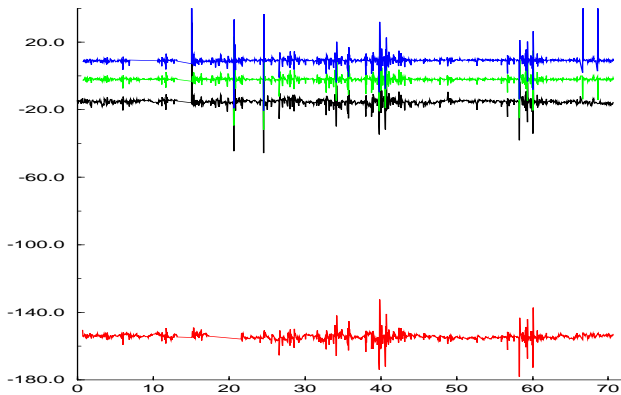


Figure 3: Measured drift rate (in $\frac{\mu s}{s}$) of four hardware clocks over a period of more than 70 days. The drift rate was determined every hour. The “spikes” in the graphs are due to measuring errors.

Clock calibration means changing the average drift rate of a clock so that it becomes about zero. Calibrating a hardware clock allows us to decrease its maximum drift rate by two orders of magnitude: from 10^{-4} to 10^{-6} . For example, clock calibration

allowed us to reduce the measured average drift from about $155 \frac{\mu s}{s}$ to about $0 \frac{\mu s}{s}$. Because we use in our protocols calibrated hardware clocks, we can use a ρ of $2 \frac{\mu s}{s}$ instead of a ρ of $200 \frac{\mu s}{s}$. Since ρ is such a small quantity, we ignore terms ρ^i for $i \geq 2$. For example, we equate $(1 + \rho)^{-1}$ with $(1 - \rho)$ and $(1 - \rho)^{-1}$ with $(1 + \rho)$. For simplicity, we assume that the clock granularity G is negligible. We assume that hardware clock failures can be detected and are transformed into process crash failure. Hence, we assume that each non-crashed process has a correct hardware clock.

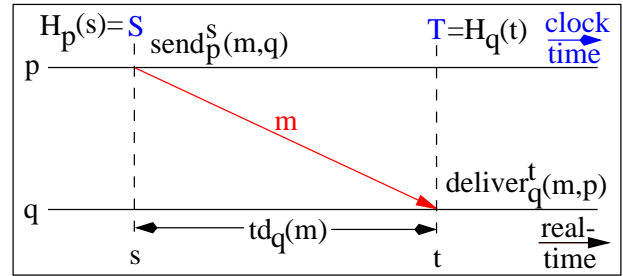


Figure 4: Process p sends unicast message m to q at real-time s and q receives m at real-time t . The transmission delay of m is $td_q(m) = t - s$.

3.2 Datagram Service

The datagram service provides primitives for transmitting unicast (see Figure 4) and broadcast messages (see Figure 5). The primitives are:

- $send(m, q)$: to send a unicast message m to process q ,
- $broadcast(m)$: to broadcast m to all processes including the sender of m , and
- $deliver(m, p)$: upcall initiated by the datagram service to deliver message m sent by process p .

To simplify the specification of the datagram service, we assume that each datagram message is uniquely identified. In other words, two messages are different even when they are sent by the same process (at two different points in time) and have the same “contents”. Let Msg denote the set of all messages. We use the following predicates to denote datagram related events:

- $deliver_q^t(m, p)$: the datagram service delivers message m sent by p to q at real-time t . We say that process q receives m at t .

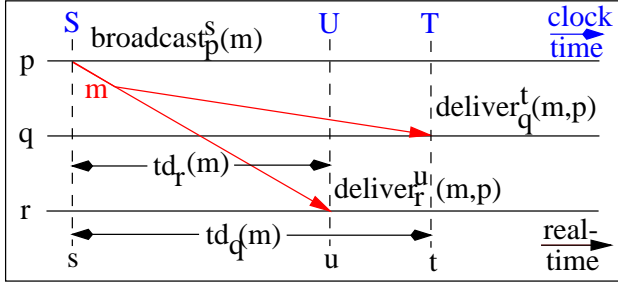


Figure 5: Process p sends broadcast message m at s and q receives m at t , while r receives m at u . The transmission delays of m are $td_q(m) = t - s$ and $td_r(m) = u - s$.

- $send_p^t(m, q)$: p transmits unicast message m to q at real-time t by invoking the primitive $send(m, q)$.
- $broadcast_p^t(m)$: p transmits broadcast message m at real-time t by invoking the primitive $broadcast(m)$.

The requirements for the datagram service are as follows:

- *Only “real” messages are delivered.*
If the datagram service delivers m to p at t and identifies q as m 's sender, then q has indeed sent m at some earlier time $s < t$:
 $\forall p, q, m, t: deliver_p^t(m, q) \Rightarrow \exists s < t: send_q^s(m, p) \vee broadcast_q^s(m)$.
- *Each message has a unique sender and is delivered at a destination process at most once.*
 $\forall p, q, r, m, t, u: deliver_p^t(m, q) \wedge deliver_r^u(m, r) \Rightarrow q = r \wedge t = u$.

Let m be a message that p sends (see Figure 4) or broadcasts (see Figure 5) at s . Let q receive m at t . We call s and t the *send* and *receive* times of m , and we denote them by $st(m)$ and $rt_q(m)$, respectively. The transmission delay $td_q(m)$ of m is defined by,

$$td_q(m) \triangleq rt_q(m) - st(m).$$

The *send time stamp* $ST(m)$ and the *receive time stamp* $RT_q(m)$ of m are defined as,

$$ST(m) \triangleq H_p(st(m)) \text{ and } RT_q(m) \triangleq H_q(rt_q(m)).$$

The function $sender(m)$ returns the sender of m :

$$sender(m) = p \Leftrightarrow \exists s, q: send_p^s(m, q) \vee broadcast_p^s(m).$$

The destination $Dest(m)$ of a message m is the set of processes to which m is sent:

$$q \in Dest(m) \Leftrightarrow \exists s, p: send_p^s(m, q) \vee broadcast_p^s(m).$$

We assume that any message sent between two remote processes p and q has a transmission delay that

is at least δ_{min} :

$$td_q(m) \geq \delta_{min}.$$

The transmission delay of a message sent between two local processes can be smaller than δ_{min} . We also assume that the size of a message is not greater than some given upper bound. For example, the maximum message size for the *UDP* datagram service [20] is 64kBytes.

The datagram service does not ensure the existence of an upper bound for the transmission delay of messages. But since all services in our model are timed, we define a *one-way time-out delay* δ , chosen so that the actual message sent or broadcast are *likely* [5] to be delivered within δ . A message m whose transmission delay is at most δ , i.e. $td_q(m) \leq \delta$, is called *timely*. If m 's transmission delay is greater than δ , i.e. $td_q(m) > \delta$, we say that m suffers a *performance failure* (or is *late*).

We require that the safety invariants of a protocol designed for the timed asynchronous system model be always true independently of the choice of δ . The choice of the timeout delay is crucial only for protocol stability and progress (see examples of safety, stability and timeliness properties in [7].) The determination of a good δ that ensures likely stability and progress typically requires the measurement of protocol specific transmission delays [22].

3.2.1 Measurements

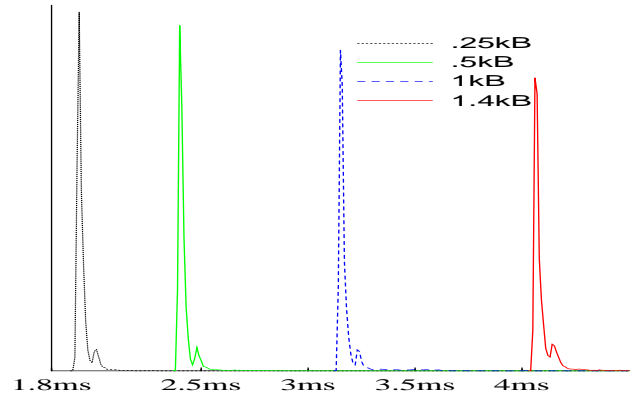


Figure 6: Measured distributions of round-trip times based on 20,000 round-trips for four different message sizes.

Message transmission delays increase with message size (see Figure 6) and depend on the message transmission pattern used by a protocol (see Figure 7). One solution to the first problem would be to make δ and also δ_{min} dependent on the size of the messages. For simplicity, we assume however that δ and δ_{min} are constant. This is a valid assumption since the message size is bounded from below and above.

To demonstrate that transmission delays can be very protocol dependent, we measured the transmission times experienced by a local leadership service [17]. This measurement involved one process p periodically broadcasting messages and five processes sending immediate replies to each broadcast message of p . After receiving a reply, p spends some time processing it before receiving the next reply. Hence, the transmission delays of all successive replies increase by the processing time of the preceding replies. The distribution of transmission delays therefore shows five peaks for the five replying processes (see Figure 7). Based on our measurements, we chose a timeout delay δ of $20ms$ for that service.

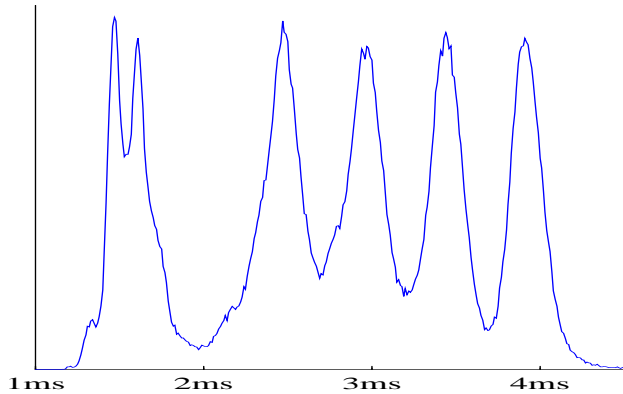


Figure 7: Distribution of the transmission delays of unicast messages sent by a local leader election protocol. This distribution is based on 500,000 replies.

In summary, the asynchronous datagram service is assumed to have an omission/performance failure semantics [5]: it can lose a message or it can fail to deliver a message in a timely manner, but one can neglect the probability that a message delivered by the service is corrupted. Broadcast messages allow asymmetric performance/omission failures in the sense that some processes might receive a broadcast message m in a timely manner, while other processes might receive m late or not at all. Since ρ and δ are such small quantities, we equate $(1 - \rho)\delta$ and $(1 + \rho)\delta$ with δ .

3.3 Process Management Service

A process p can be in one of the following three modes (see Figure 8):

- *up*: p is executing its ‘standard’ program code,
- *crashed*: a process stopped executing its code and has lost all its previous state, and
- *recovering*: p is executing its state ‘initialization’ code, (1) after its creation, or (2) when it restarts after a crash.

A process that is either crashed or ‘recovering’ is said to be *down*. The following events cause a process p to transition between the modes specified above (see Figure 8):

- *start*: when p is created, it starts in ‘recovering’ mode,
- *crash*: p can crash at any time, for example because the underlying operating system crashes.
- *ready*: p transitions to mode ‘up’ after it has finished initializing its state, and
- *recover*: when p restarts after a crash, it does so in ‘recovering’ mode.

We define the predicate $crashed_p^t$ to be true iff p is crashed at time t .

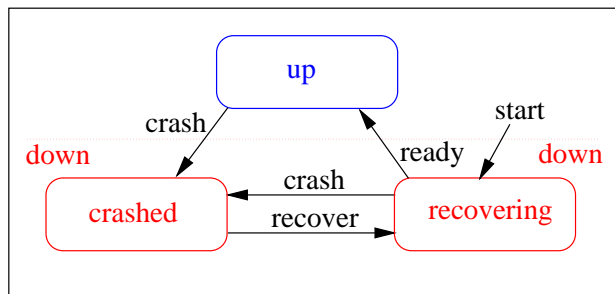


Figure 8: Process modes and transitions.

A process p can set an *alarm clock* to be awakened at some specified clock time. When p requests to be awakened at clock time T , the process management service will awake p when $H_p(t)$ shows a value of at least T . Note that an alarm clock can be used to implement a *timer* that allows to define an alarm time relative to the current time instead of specifying an absolute time. We use the following predicates to specify the behavior of alarm clocks:

- $SetAlarm_p^s(T)$: p requests at real-time s to be awakened at some future real-time u such that $H_p(u) \geq T$,
- $WakeUp_p^u(T)$: the process management service wakes up p at real-time u for some previously specified *alarm clock time* T .

When a process p crashes, the process management service forgets all alarm times p set before crashing. We require that a process p be awakened for a set alarm clock time T only when 1) its hardware clock shows at least T , 2) p has previously requested to be awakened at T , and 3) p has not crashed and has not been awakened for T since then:

$\forall p, u, T: WakeUp_p^u(T) \Rightarrow H_p(u) \geq T \wedge \exists s < u, \forall v \in [s, u):$
 $SetAlarm_p^s(T) \wedge \neg crashed_p^v \wedge \neg WakeUp_p^v(T).$

Let t be the earliest (or smallest) real-time for which $H_p(t) \geq T$. We call t the *real alarm time* specified by the $SetAlarm_p^s(T)$ event. Consider that the process management awakes process p for alarm time T at real-time u , i.e. $WakeUp_p^u(T)$ holds. The delay $u - t$ is called the *scheduling delay* experienced by the process p . The process management service does not ensure the existence of an upper bound on scheduling delays. However, being a timed service, like all services in the timed model, we define a scheduling timeout delay σ , so that actual scheduling delays are likely [5] to be smaller than σ .

We say that a process p suffers a *performance failure* when it is not awakened within σ time units of a specified real alarm time (see Figure 9). Otherwise, p is said to be *timely*. Thus, a process p suffers a performance failure at real-time u if there exists an alarm time T that should have caused a *WakeUp* event by u :

$pFail_p^u \triangleq \exists s \leq u, \exists T: SetAlarm_p^s(T) \wedge H_p(u - \sigma) \geq T$
 $\wedge \forall v \in [s, u]: \neg WakeUp_p^v(T) \wedge \neg crashed_p^v(T).$
 Formally, we define the predicate $timely_p^u$ to be true iff p is timely at u :

$$timely_p^u \triangleq \neg pFail_p^u \wedge \neg crashed_p^u.$$

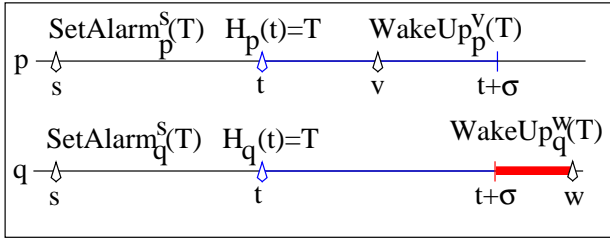


Figure 9: Process p is timely because it is awakened within σ real-time units of the real alarm time t corresponding to the specified alarm clock time T . Process q suffers a performance failure in $[t + \sigma, w)$ because it is awakened at real-time w after $t + \sigma$.

3.3.1 Measurements

To implement alarm clocks in the Unix family of operating systems, one can use the *select* system call. This call allows to specify a maximum interval for which a process waits for some specified I/O events in the kernel before it returns. Unix tries to awake the process *before* the specified time interval expires using an internal timer. This timer has typically a resolution of $10ms$. Thus, the scheduling delay is at least $10ms$. Figure 10 shows the distribution of scheduling delays experienced by a process executing a membership protocol [14]. Based on our mea-

surements, we selected a scheduling timeout delay of $30ms$ for this membership protocol.

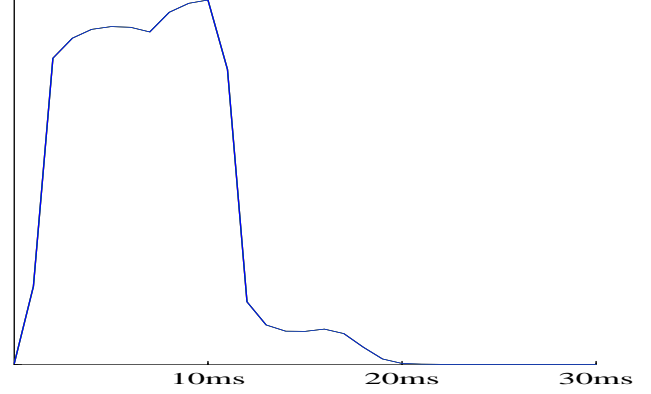


Figure 10: The distribution shows the difference between the time a process was awakened and the time it requested to be awakened. It is based on 350,000 measurements.

In summary, the timed model assumes that processes have crash/performance failure semantics [5] and that processes can recover from crashes. Since ρ and σ are such small quantities, we equate $(1 - \rho)\sigma$ and $(1 + \rho)\sigma$ with σ .

4 Extensions

The core of the timed asynchronous system model assumes the datagram service, the process management service and the local hardware clocks. We introduce two optional extensions of the model: *stable storage* and *progress assumptions*. Both extensions are reasonable for a network of workstations. However, not all systems might need to have or actually have access to stable storage. A progress assumption states that infinitely often a majority of processes will be stable for a bounded amount of time. While progress assumptions are valid for most local area network based systems, they are not necessarily valid for large scale systems connected by wide area networks. Moreover, most of our service specifications do not need a progress assumption to enable their implementation in timed asynchronous systems.

4.1 Stable Storage

Processes lose their memory state when they crash. To allow processes to store information between crashes, we introduce an extension of the timed asynchronous system model: a local stable storage service. This service provides the following two primitives to any local process p :

- $store(addr, val)$: p asks the value val to be stored at address $addr$, and
- $read(addr, val)$: p asks to read the most recent value it has stored at address $addr$. If p has not yet stored some value at $addr$, the undefined value \perp is returned.

The predicates that denote the invocation of the above primitives are: $store_p^t(addr, val)$ and $read_p^t(addr, val)$, respectively.

The stable storage service guarantees that for any address a that a process p reads, it returns the most recent value that p has stored at address a , if any:

$$\begin{aligned} read_p^t(a, val) \Rightarrow \\ \forall u < t, v: \neg store_p^u(a, v) \wedge val = \perp \\ \vee \exists s < t: store_p^s(a, val) \wedge \forall u \in (s, t], v: \neg store_p^u(a, v). \end{aligned}$$

A stable storage service can be implemented on top of Unix using the Unix file system. An implementation of such a service and its performance is described in [9].

4.2 Stability and Progress Assumptions

The timeliness requirements encountered in the specification of protocols designed for the timed asynchronous system model are often *conditional* in the sense that only when some “system stability” predicate is true, the system has to achieve “something good” (see e.g. [7]). Such conditional timeliness requirements express that when some set of processes SP is “stable”, that is, the failures affecting SP have a bounded rate of occurrence, the servers in SP have to guarantee progress within a bounded amount of time. We call a set SP a *stable partition* [17] iff

- all processes in SP are timely,
- all but a bounded number of messages sent between processes in SP per protocol round are delivered in a timely manner, and
- from any other partition either no message or only “late” messages arrive in SP .

The concept of a *stable partition* is formalized by a *stability predicate* that defines if a set of processes SP forms a stable partition in some given time interval $[s, t]$. There are multiple reasonable definitions for stability predicates: examples are the *stable* predicate in [10], or the *majority-stable* predicate in [12]. In this paper we formally define the stability predicate Δ -*F-partition* introduced informally in [15]. To

do that, we first formalize and generalize the notions of connectedness and disconnectedness introduced in [10].

Two processes are *F-connected* in the time interval $[s, t]$ iff (1) p and q are timely in $[s, t]$, and (2) all but at most F messages sent between the two processes in $[s, t]$ are delivered within at most δ time units. We denote the fact that p and q are *F-connected* in $[s, t]$ by the predicate $F\text{-connected}(p, q, s, t)$:

$$\begin{aligned} F\text{-connected}(p, q, s, t) \triangleq & \exists M \subseteq Msg : |M| \leq F \\ & \wedge \forall u \in [s, t] : timely_p^u \wedge timely_q^u \\ & \wedge \forall m \in Msg - M, \forall r \in \{p, q\} : st(m) \in [s, t] \wedge \\ & \quad Sender(m) \in \{p, q\} \wedge r \in Dest(m) \Rightarrow td_r(m) \leq \delta. \end{aligned}$$

A process p is Δ -*disconnected* from a process q in $[s, t]$ iff any message m that p receives in $[s, t]$ from q has a transmission delay of more than $\Delta > \delta$ time units. A common situation in which two processes are Δ -disconnected is when the network between them is overloaded or at least one of the processes is slow. One can use a fail-aware datagram service [16] to classify messages with a transmission delay greater than some $\Delta > \delta$ as “slow” and messages with a transmission delay of at most δ as “fast”. Messages with a transmission delay within $(\delta, \Delta]$ are either classified as “slow” or “fast”. We use the predicate Δ -*disconnected*(p, q, s, t) to denote that p is Δ -disconnected from q in $[s, t]$:

$$\begin{aligned} \Delta\text{-disconnected}(p, q, s, t) \triangleq & \forall m, \forall u \in [s, t] : \\ & deliver_p^u(m) \wedge sender(m) = q \Rightarrow td_p(m) > \Delta. \end{aligned}$$

We say that a non-empty set of processes S is a Δ -*F-partition* in an interval $[s, t]$ iff all processes in S are *F-connected* in $[s, t]$ and the processes in S are Δ -disconnected from all other processes:

$$\begin{aligned} \Delta\text{-F-partition}(S, s, t) \triangleq & S \neq \emptyset \\ & \wedge \forall p, q \in S : F\text{-connected}(p, q, s, t) \\ & \wedge \forall p \in S, \forall r \in \mathcal{P} - S : \Delta\text{-disconnected}(p, r, s, t). \end{aligned}$$

As an example of the utility of the above stability predicate, consider an atomic broadcast protocol designed to achieve *group agreement* semantics [7], where all messages that are possibly lost or late are re-sent up to $F + 1$ times. If a group of processes S forms a Δ -*F-partition* for sufficiently long time, that group can make progress in successfully broadcasting messages during that time.

4.2.1 Progress Assumptions

The lifetime of a distributed system based on a local area network is characterized by long periods in which there exists a majority of processes that are stable. These stability periods alternate with short instability periods. Based on this observation,

we introduced the concept of *progress assumptions* [12] to show that classical services, such as consensus, originally specified by using unconditional termination requirements, are implementable in the extended timed model. A progress assumption states that the system is infinitely often “stable”: there exists some constant η such that for any time s , there exists a $t \geq s$ and a majority of processes SP so that SP forms a stable partition in $[t, t + \eta]$.

4.2.2 Measurements

We measured the behavior of six processes each running on a SUN workstation in our Dependable Systems Lab over a period of a day under normal load conditions (see Figure 11). The set of all six processes were, on the average, Δ -1-stable for about 218s. The average distance between two Δ -1-stable periods was about 340ms. The typical behavior experienced during an “unstable” phase was that one of the six processes was slow. The membership service [14] allows the fast processes to continue to make progress by temporarily removing the slow process(es) from the membership. Note that a progress assumption only requires that a majority of the processes form a stable partition, i.e. it is sufficient that at least four of the processes be Δ -1-stable.

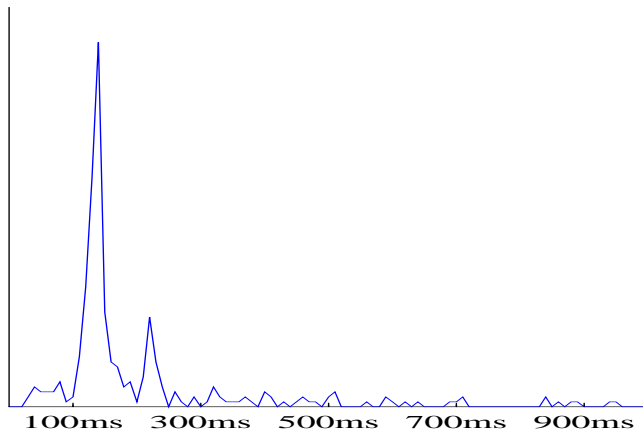


Figure 11: Distribution of the time between two Δ -1-stable periods for six processes, where $\Delta = 20ms$ and $\sigma = 30ms$. The typical failure behavior observed between consecutive stability periods was that one process was slow.

5 Communication By Time

In synchronous systems, communication by measuring the passage of time is certain: if a correct process p does not hear in time the ‘I-am-alive’ message of q , then q has failed. The communication uncertainty that characterizes timed asynchronous

systems makes such “communication by time” more difficult, but still possible. To illustrate how processes can communicate by (measuring the passage) of time in timed asynchronous system, we sketch how two processes p and q can ensure that at any time at most one of them is leader. We use a *locking mechanism* [15] that facilitates communication by time even when local clocks are not synchronized. The mechanism ensures that if p sends some information in a message m to a process q , p can determine by consulting only at its local hardware clock the real-time beyond which q will no longer make use of the information contained in m , since p knows that if q receives m , q uses the information only for a bounded duration after m was sent.

For concreteness, consider the pseudo-code of Figure 12. Process p sends a message m to q informing q that it can become leader for *Duration* clock time units if the transmission delay of m is at most Δ real-time units. Process q has to calculate an upper bound on the transmission delay of m to determine if it can use m . The fail-aware datagram service introduced in [16] calculates an upper bound on the transmission delay of messages (using round-trip messages). It delivers m as “fast” when its transmission delay is at most Δ . Process q uses m only when m is “fast” and it sets variable *ExpirationTime* so that q ’s leadership expires exactly *Duration* time units after the reception of m at local clock time *RT*.

Process p waits for $Duration(1+2\rho)+\Delta(1+\rho)$ clock time units before it becomes leader, where the factor $(1 + 2\rho)$ is necessary because p ’s and q ’s hardware clocks can drift apart by up to 2ρ . A process $r \in \{p, q\}$ is leader at t iff the function *Leader?* evaluates at t to true when called with the value of r ’s hardware clock at t as argument, i.e.

$$\text{leader}_r^t \triangleq \text{Leader?}_r^t(H_r(t)).$$

Process p and q are never leader at the same time since (1) q can only be leader when the transmission delay of m is at most Δ and it is leader for at most *Duration* local clock time units after receiving m , and (2) after p has sent m , it waits for at least $Duration(1+2\rho)+\Delta(1+\rho)$ local clock time units before becoming leader.

6 Possibility and Impossibility Issues

We address in this section the issue of why problems like election and consensus are implementable in actual distributed computing systems while they do not allow a deterministic solution in (1) the time-free model and (2) to some extent in the core timed

```

const   time Duration,  $\Delta$ ;
boolean Leader?(time now)
  if now < ExpirationTime then
    return true;
  return false;
process p begin
  time ExpirationTime = 0;
  fa-send("you are leader", q);
  T = H()+Duration(1+2* $\rho$ )+ $\Delta$ (1+ $\rho$ );
  SetAlarm(T);
  select event
    when WakeUp(T):
      ExpirationTime =  $\infty$ ;
  end select
end
process q begin
  time ExpirationTime = 0;
  select event
    when fa-deliver(m, p, fast, RT);
      if fast then
        ExpirationTime = RT+Duration;
      endif
  end select
end

```

Figure 12: This pseudo-code uses communication by time to enforce that p is eventually leader while ensuring that there is only one leader at a time.

model. To fix our ideas, we use the election problem to illustrate the issues.

Whether the leader problem has a deterministic solution or not depends on 1) the exact specification of the problem, 2) whether the underlying system model allows communication by time, and 3) on the use of progress assumptions.

6.1 Termination vs Conditional Timeliness Conditions

There is no commonly agreed-upon rigorous specification for the election problem. For example, [21] specifies the election problem for the time-free system model as follows:

- (S) at any real-time there exists at most one leader, and
- (TF) infinitely often there exists a leader, i.e. for any real-time s there exists a real-time $t \geq s$ and a process p so that p becomes leader at t .

Problems specified for timed systems do not use such strong unconditional *termination conditions* re-

quiring that “something good” eventually happens. Instead, we use *conditional timeliness condition*. These require that if a system stabilizes for an a priori known duration, “something good” will happen within a *bounded* time. With the introduction of the Δ -F-stable predicate earlier, we can generalize the specifications given in [10] and [17] for the election (or the highly available leadership) problem for timed asynchronous systems as follows:

- (S) at any real-time there exists at most one leader, and
- (TT) when a majority of processes are Δ -F-stable in a time interval $[s, s + \kappa]$, then there exists a process p that becomes leader in $[s, s + \kappa]$.

The specification (S, TF) is not implementable in time-free systems, even when only one process is allowed to crash [21], while (S, TT) is implementable in timed systems [10, 17]. To explain why this is so, consider a time-free system that contains at least two processes p and q . To implement (S, TF) , one has to solve the following problem: when a process p becomes leader at some real-time s and stays leader until it crashes at a later time $t > s$, the remaining processes have to detect that p has crashed to elect a new leader at some time $u > t$ to satisfy requirement (TF). Since processes can only communicate by messages, one can find a run that is indistinguishable for the remaining processes and in which p is not crashed and is still leader at u . In other words, one can find a run in which at least one of the two requirements (S, TF) is violated.

The implementability of (S, TT) in a timed asynchronous system can be explained as follows. First, to ensure property (S) processes do not have to decide if the current leader is crashed or just slow. A process is leader for a bounded amount of time before it is demoted (see Section 5). Processes can therefore just wait for a certain amount of time (without exchanging any messages) to make sure that the leader is demoted. In particular, processes do not have to be able to decide if a remote process is crashed (this is impossible in both the time-free and the timed asynchronous system models). Second, when the system is stable, a majority of timely processes can communicate with each other in a timely fashion. This is sufficient to elect one of these processes as leader in a bounded amount of time and ensure that the timeliness requirement is also satisfied [10, 17].

Note that the specification (S, TF) is not implementable in the core timed model even when only one process is allowed to crash. To explain this, consider a run R in which no process can communicate

with any other process (because the datagram service drops all messages). If at most one process l in R is leader, we can construct a run R' such that l is always crashed in R' and R' is indistinguishable from R for the remaining processes, therefore R' does not satisfy (S, TF) . Otherwise, if there exist at least two processes p and q in R that become leaders at times s and t , respectively, we can construct a run R'' that is indistinguishable from R for all processes and in which p and q are leaders at the same point in real-time ($s = t$), since p and q do not communicate with any other process.

6.2 Why Communication by Time is Important for Fault-Tolerance

One interesting question is if (S, TT) could be implemented in time-free systems. Since no notion of stability was defined for time-free systems, we sketch the following alternative result instead: (S, TT) is not implementable in a timed system from which hardware clocks are removed even if at most one process can crash and no omission failures can occur. Note that, if processes have no access to local hardware clocks, they cannot determine an upper bound on the transmission delay of messages nor can a leader enforce that it demotes itself within a bounded time that is also known to all other processes. In particular, the only means for interprocess communication is, like in the time-free model, explicit messages. Thus, the proof sketched above that (S, TF) is not implementable in the time-free model also applies for the timed system model without hardware clocks. It is thus essential to understand that it is the access to local clocks that run within a linear envelope of real-time, which *enable communication by time*, that allows us to circumvent in the timed model the impossibility result of [21] stated for the time-free model.

6.3 Progress Assumptions

Another observation is that while (S, TF) does not have a deterministic solution in the core timed model, it is implementable in a practical network of workstations. The reason is that while the timed asynchronous system model allows in principle runs in which the system is never stable, the actual well-tuned production systems that one encounters in practice make such behavior extremely unlikely because δ and σ are assumed to be well chosen. As mentioned earlier, such a system is very likely to alternate between long stability periods and relatively short instability periods. To describe such systems, it is therefore reasonable to use a progress assumption

(see Section 4.2.1), that is, assume the existence of an η such that the system is infinitely often stable for at least η time units. For $\eta \geq \kappa$, a progress assumption ensures that a solution of (S, TT) elects a leader infinitely often. Thus, the introduction of a progress assumption implies that a solution of (S, TT) is also a solution of (S, TF) .

In the service specifications we have defined for asynchronous services implementable in the timed model, we always use *conditional timeliness conditions* and we never use *termination conditions* like (TF) . In general we do not need progress assumptions to enable the implementation of services with conditional timeliness conditions in timed asynchronous systems, i.e. these services are implementable in the core timed system model. Furthermore, while progress assumptions are reasonable for local area systems, they are not necessarily valid for wide area systems that frequently partition for a long time. Thus, we have not included progress assumptions as a part of the core timed asynchronous system model.

7 Conclusion

We have given a rigorous definition of the timed asynchronous system model. Based on the measurements performed on the network of workstations in our Dependable Systems Laboratory, and on other unpublished measurements at other labs that we are aware of, we believe that the timed asynchronous system model is an accurate description of actual distributed computing systems. In particular, we believe that the set of problems solvable in the timed model extended by progress assumptions is a close approximation of the set of problems solvable in systems of workstations linked by reliable, possibly local area based, networks.

The timed asynchronous system model is not only interesting from a practical point of view, it also raises interesting theoretical questions. For example, it does not only allow to solve different problems than the time-free model [18], it seems that the minimum message complexity to solve a problem could be different in the two models because the former allows communication by time. There are also many open questions with respect to the relation between the timed model extended by a progress assumption, and the time-free model extended by a failure detector [3]. For example, one question is which failure detectors are implementable in timed asynchronous systems and which are not. So far, we have shown that a Perfect failure detector is not implementable in a timed system even when extended by a progress assumption [13].

References

- [1] Y. Amir and A. Wool. Evaluating quorum systems over the internet. In *Proc. 26th Int Symp on Fault-tolerant Computing*, pages 26–35, Sendai, Japan, June 1996.
- [2] T. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost. On the impossibility of group membership. In *Proceedings of the Fifteenth ACM Symposium on Principles of Distributed Computing*, pages 322–330, May 1996.
- [3] T. Chandra and S. Toueg. Unreliable failure detectors for asynchronous systems. In *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing*, pages 325–340, Aug 1991.
- [4] F. Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3:146–158, 1989. An earlier version IBM Research Report, San Jose, RJ 6432, 1988.
- [5] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of ACM*, 34(2):56–78, Feb 1991.
- [6] F. Cristian. Group, majority, and strict agreement in timed asynchronous distributed systems. In *Proc. of the 26th Int. Symposium on Fault-Tolerant Computing*, pages 178–187, Sendai, Japan., June 1996.
- [7] F. Cristian. Synchronous and asynchronous group communication. *Communications of the ACM*, 39(4):88–97, Apr 1996.
- [8] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to Byzantine agreement. *Information and Computation*, 118:158–179, April 1995. Early version: *FTCS15*, June 1985.
- [9] F. Cristian, S. Mishra, and Y. Hyun. Implementation and performance of a stable storage service for unix. In *Proceedings of the 15th Symposium on Reliable Distributed Systems*, pages 86–95, Niagara-on-the-Lake, Canada, Oct 1996.
- [10] F. Cristian and F. Schmuck. Agreeing on processor-group membership in asynchronous distributed systems. Technical Report CSE95-428, UCSD, 1995.
- [11] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, Jan 1987.
- [12] C. Fetzer and F. Cristian. On the possibility of consensus in asynchronous systems. In *Proceedings of the 1995 Pacific Rim Int'l Symp. on Fault-Tolerant Systems*, Newport Beach, CA, Dec 1995.
- [13] C. Fetzer and F. Cristian. Fail-aware failure detectors. In *Proceedings of the 15th Symposium on Reliable Distributed Systems*, Niagara-on-the-Lake, Canada, Oct 1996.
- [14] C. Fetzer and F. Cristian. Fail-aware membership services. Technical Report CSE96-XXX, UCSD, 1996.
- [15] C. Fetzer and F. Cristian. Fail-awareness: An approach to construct fail-safe applications. Technical Report CS96-501, UCSD, Nov 1996.
- [16] C. Fetzer and F. Cristian. Fail-awareness in timed asynchronous systems. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing*, Philadelphia, May 1996.
- [17] C. Fetzer and F. Cristian. A highly available local leader service. Technical Report CS96-499, UCSD, Nov 1996.
- [18] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, Apr 1985.
- [19] L. Lamport and N. Lynch. Distributed computing: Models and methods. In *Handbook of Theoretical Computer Science*, pages 1158–1199. Elsevier Science Publishers, 1990.
- [20] J. Postel. User datagram protocol. Technical Report RFC768, USC/Information Sciences Institute, 1980.
- [21] L. Sabel and K. Marzullo. Election vs. consensus in asynchronous systems. Technical Report TR95-1488, Cornell University, Feb 1995.
- [22] O. Suciuc and C. Fetzer. Determining the user-level transmission delay in networks of workstations. Technical Report CS96-511, UCSD, Dec. 1996.
- [23] P. Verissimo and C. Almeida. Quasi-synchronism: a step away from the traditional fault-tolerant real-time system models. *IEEE TCOS Bulletin*, 7(4), Dec 1995.
- [24] P. Verissimo, L. Rodrigues, and M. Baptista. Amp: A highly parallel atomic multicast protocol. In *SIGCOMM'89*, pages 83–93, Austin, TX, Sep 1989.