The Timed Asynchronous Distributed System Model

Flaviu Cristian and Christof Fetzer

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

Keywords— **timed model, system model, failure model, distributed systems, asynchronous systems, synchronous systems, measurements, communication by time.**

I. INTRODUCTION

Depending on whether the underlying communication and process management services provide "certain communication", distributed systems can be classified as either *synchronous* or *asynchronous* [7]. By *certain communication* we mean that 1) at any time there is a minimum number of correct processes, and 2) any message m sent by a correct process to a correct destination process is received and processed at the destination within a known amount of time, i.e. the probability that m is not received and processed in time is "negligible". The authors of [28], [5] explain what it means for failures to be negligible. A synchronous system guarantees certain communication. All other systems are asynchronous.

To achieve certain communication, one assumes that the frequency of failures that can occur in a system is bounded. This bounded failure frequency assumption allows system designers to use space [8] or time redundancy [32] to mask lower level communication failures and provide the abstraction of certain communication. However, for almost all distributed systems it is not reasonable to assume that the failure frequency is bounded.

Dependable systems are characterized by strict stochastic specifications [5]. Hence, even if one tries to fix the unpredictability of a system to achieve certain communication (e.g. through admission control, resource allocation, redundant communication channels, etc.), the probability of communication failures might still not be negligible. For many dependable systems it is therefore not necessarily reasonable to assume that communication is certain. In this paper we define an asynchronous system model that makes much simpler assumptions than a synchronous system model. Hence, the probability that one of these assumptions is violated is much smaller than the probability of a violation of the the assumptions of a synchronous system. Nevertheless, this asynchronous system model is still strong enough to serve as a foundation for the construction of dependable applications.

Most published research on asynchronous systems is based on the *time-free* model [21]. This model is characterized by the following properties: 1) services are time-free, i.e. 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 (some researchers relax this condition), i.e. any message sent between two non-crashed processes is eventually delivered to the destination process, 3) processes have crash failure semantics, i.e. processes can only fail by crashing, and 4) processes have no access to hardware clocks. In the time-free model a process cannot distinguish between a non-crashed (but very slow) and a crashed process. Most of the services that are of importance in practice, such as consensus, election or membership, are therefore not implementable [21], [2].

The *timed asynchronous distributed system model* (or, shorter the *timed model*) which we define formally in this paper 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: the only failures that messages can suffer are omission (message is dropped) and performance failures (message is delivered late, 3) processes have crash/performance failure semantics: the only failures a process can suffer are crash and performance failures, 4) processes have access to hardware clocks that proceed within a linear envelope of real-time, and 5) no bound exists on the frequency of communication and process failures that can occur in a system. We feel this model adequately describes existing distributed systems built from networked workstations. In contrast with the time-free model, the timed model allows practically needed distributed services such as clock synchronization, membership, consensus, election, and atomic broadcast to be implemented [4], [10], [14], [6], [13].

Since it does not assume the existence of hardware 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 - at the highest level: the human user - fixes a timeout to decide if they have failed. Therefore, from a practical point of view the requirements that services be timed

and processes have access to hardware clocks do not make the timed model less general than the time-free model.

In fact, the failure semantics of interprocess communication in the time-free model (as defined in [21]) 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, the timed model allows such partitioning to be naturally modeled as the occurrence of sufficiently many message omission or performance failures. This characteristic of the timed model reflects the situations in which communication partitions can be observed for hours, or even days 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 goals of this paper are to 1) propose a formal definition for the timed asynchronous distributed 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.

II. RELATED WORK

Distributed system models can be classified according to what they assume about *network topology*, *synchrony*, *failure model*, and *message buffering* [23]. 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 message omission failures.

The most important difference between the timed model and the time-free model [21] is the existence of local hardware clocks. Many distributed applications are specified using real-time constraints. For example, if a component fails, then within X time units the application has to perform some action. Hardware clocks allow one to implement application level "timeouts".

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 [21]. 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 frequency increases beyond the assumed threshold. In [14] we formalized this as *progress assumptions*. A progress assumption is an optional extension of the "core" timed asynchronous system model (see Section IV): a progress assumption states that after an unstable period there always exists a time interval of some given minimum length in which the system will be stable. Progress assumptions allow one to solve problems like consensus, that were originally specified by using unconditional termination conditions (defined in Section VI-A), as opposed to our use of conditional timeliness properties (see Section VI-A). One can view a progress assumption as a formal way to require the parameters of the timed model (the *one-way time-out delay* δ; Section III-B and the scheduling timeout delay σ; Section III-C.2) to be well chosen. Progress assumptions are similar 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. However, progress assumption only require that infinitely often there exists a majority set of processes that for a certain minimum amount of time are timely and can communicate with each other in a timely manner.

Progress assumptions have also a certain similarity with *failure detectors* [3], which are mechanisms to strengthen the time-free model: certain failure detector classes provide their desired behavior based on the observation that the system eventually stabilizes. The main differences between the model of [3] and the timed model are the following: 1) the timed model allows messages to be dropped and processes to recover after a crash, and 2) the timed model provides processes with access to hardware clocks while the model of [3] provides processes with access to a failure detector. Note that hardware clocks can be used to detect failures. 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.

We will sketch in Section VI that certain problems that are implementable in synchronous systems are not implementable in timed asynchronous systems. Previously, other authors addressed possibility issues. For example, [24] and [27] address the issue of what problems can be *simulated* in an asynchronous system. In Section VI we do however not address simulation issues: for example, we are concerned about how one can ensure that there are no two leaders at any point in *real-time* and we are not interested in solutions where there are no two leaders in *virtual* time. This difference is important for real-time systems that have to interact with external processes.

In [15] we introduced the notion of *fail-awareness* as a systematic means of transforming synchronous service specifications into *fail-aware* specifications that become implementable in timed asynchronous systems. The idea is that processes have to provide their "synchronous properties" as long as the failure frequency is below a given bound and whenever a property cannot be guaranteed anymore, this is detectable in a timely manner by all correct clients that depend on this property. Our claim is that the weakened fail-aware specification is still useful while implementable in a timed system. Fail-awareness depends on the timely detection of message performance failures. We introduced in [19] a mechanism that allows a receiver r of a message m to detect if m has suffered a performance failure: the basic idea is that 1) one can use local hardware clocks to measure the transmission delay of a message round-trip, and 2) one can use the duration of a round-trip that contains m as an upper bound for the tranmission delay of m. We introduced in [19] several optimizations to provide a better upper bound for m. [12] describes the use of the timed model and a fail-aware datagram service in a fully automated train control system.

The *quasi-synchronous model* [31] 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 incomplete *assumption coverage*, that is, a non-zero probability that the bound postulated by (Px) is violated at run-time [28]. 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.

III. THE MODEL

A timed asynchronous distributed system consists of a finite set of processes $\mathcal{P}$, which communicate via a datagram service. Processes run on the computer nodes of a network (see Figure 1). Lower level software in the nodes and the network implements the datagram service. Two processes are said to be *remote* if they run on separate 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.

A. Hardware Clocks

All processes that run on a node can access the node's *hardware clock*. The simplest hardware clock consists of an 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*. Other hardware clock implementations are described in [26]. *Correct clocks display strictly monotonically increasing values.*

We denote the set of real-time values with $\mathcal{RT}$ and the set of
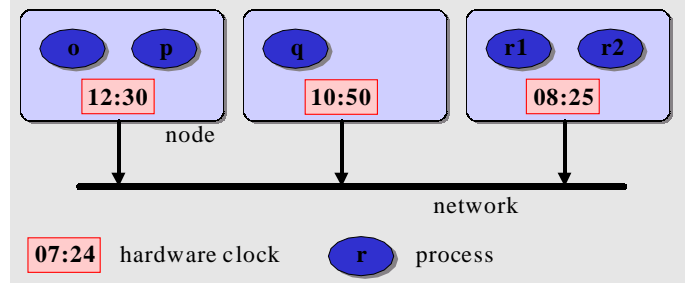

Fig. 1. *Processes in a timed asynchronous system have access to local hardware clocks and communicate via datagram messages across a network.*

clock values with $\mathcal{CT}$. The clock H_p of process p is represented by a function H_p from real-time to clock-time:

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

$H_p(t)$ 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. Thus, if processes p and q are running on the same node, $H_p = H_q$.

Due to the imprecision of the oscillator, temperature changes, and aging, a hardware clock drifts apart from real-time. Intuitively, the *drift rate* of a hardware clock indicates how many microseconds a hardware clock drifts apart from real-time per second. For example, a drift rate of $2\frac{\mu s}{s}$ means that a clock increases its value by $1sec + 2\mu s$ every second.

We assume the existence of a constant *maximum drift rate* $\rho \ll 1$ that bounds the absolute value of the drift rate of a correct clock. Thus, the drift rate of a correct clock is at least $-\rho$ and at most $+\rho$ (see Figure 2). The constant ρ is known to all processes. A correct clock measures the duration of a time interval $[s, t]$ with an error within $[-\rho(t - s) - G, \rho(t - s) + G]$. The term G accounts for the error due to the granularity of the clock and the factor ρ for the error due to the drift of the clock.

We define a predicate $correct_{H_p}^u$ that is true iff p's hardware clock H_p is correct at time u. The definition is based on the intuition that H_p has to measure the duration of any time interval $[s, t]$ before u with an absolute error of at most $\rho(t - s) + G$:

$$correct_{H_p}^u \triangleq \forall s,t:\ s<t\leq u \Rightarrow$$
$$(1-\rho)(t-s)-G \leq H_p(t)-H_p(s) \leq (1+\rho)(t-s)+G.$$

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

When one analyzes the drift error of a clock, it is possible to distinguish (1) a systematic drift error due to the imprecision of its oscillator, and (2) drift errors due to other reasons such as aging or changes in the environment. The speed of a *calibrated* hardware clock is changed by a constant factor c to reduce the systematic drift error. The relation between an uncalibrated clock H_p and its calibrated counterpart $H_p^{calibrated}$ can be expressed as follows:

$$H_p^{calibrated}(t) = cH_p(t).$$

Hardware clock calibration can be done automatically in systems that have Internet access or have local access to an external time provider such as a GPS receiver.

Clocks are *externally synchronized* if at any instant the deviation between any correct clock and real-time is bounded by a known constant. Clocks are *internally synchronized* if the deviation between any two correct clocks is bound by a known
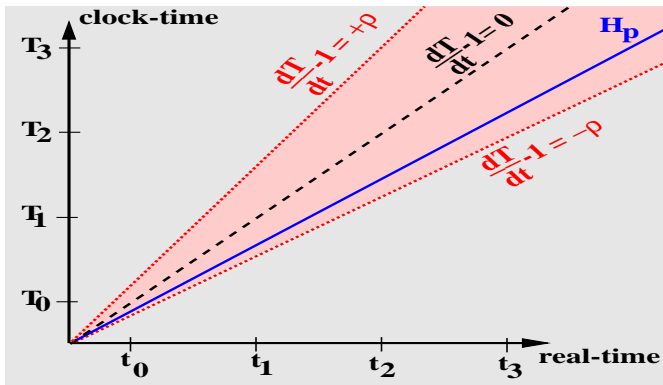
Fig. 2. *At any point in time the drift rate of a correct hardware clock H_p is within $[-\rho, +\rho]$. Note that the drift rate does not have to be constant since it can change over time and can assume any value within $[-\rho, +\rho]$.*

constant. If all correct clocks in a system are externally synchronized by some known ϵ, then the clocks are also internally synchronized by 2ϵ. Clock calibration can be done once during the lifetime of a system. However, to account for aging of a clock, it makes sense to recalibrate the clock occasionally. Internal and external clock synchronization needs to be performed periodically to account for the ongoing drift of all clocks.

The timed asynchronous system model does not require clocks to be calibrated, nor to be externally or internally synchronized. Only their drift rate has to be bounded by ρ. However, it is advantageous to calibrate hardware clocks since this allows the reduction of the maximum drift rate (see below).

A.1 Measurements

Common operating systems provide processes access to a "real-time clock". This real-time clock is more or less synchronized with real-time, e.g. UTC (universal time) or GPS time. In many Unix domains one tries to maintain a good synchronization with real-time using time services like NTP [25]. However, processes do not always know how good the synchronization with real-time is. There might not even exist an upper bound on the drift rate of a real-time clock because an operator can change the speed of the real-time clock [1].

More recent operating systems provide processes access to hardware clocks that are not subject to adjustments, i.e. neither the software nor the operator can change the speed of such a clock. For example, Solaris provides a C library function *gethrtime* (get high-resolution real-time) that returns a clock value expressed in nanoseconds. The high-resolution real-time clocks are an example of hardware clocks provided by an operating system. They are also ideally suited to implement *calibrated* hardware clocks.

For current workstation technology, the granularity of a hardware clock is in the order of $1ns$ to $1\mu s$, and the constant ρ is in the order of 10^{-4} to 10^{-6}. We measured the drift rate of the uncalibrated, unsynchronized hardware clocks of several SUN workstations running Solaris 2.5 over a period of several weeks. The Figure 3 shows the drift rate of four hardware clocks. We measured the drift rate using both a NIST timeserver and clocks externally synchronized via NTP. The average drift rate of all four hardware clocks stayed almost constant over the measured

period. The computers are all located in air-conditioned rooms. For computers that are subject to higher temperature changes one has to expect a higher variance in the clock drift. Note that we do not assume that the clock drift is constant ; we assume that clock drift is within some $[-\rho, +\rho]$. This interval has to be chosen large enough to account for unsteady environmental conditions.
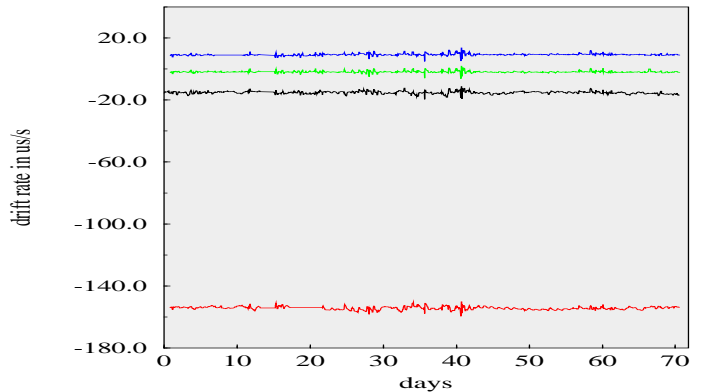

Fig. 3. *Measured drift rate (in $\frac{\mu s}{s}$) of four hardware clocks over a period of more than 70 days. The drift rate was determined every hour using externally synchronized clocks.*

We measured the relative drift rate of two calibrated hardware clocks for an interval of several days, i.e. we measured how many μs two calibrated hardware clocks drift apart from each other every second. The calibrated clocks were implemented on top of the high-resolution real-time clocks of Solaris. Variations in the message transmission delays introduce errors when reading remote clocks in a distributed system. We did read remote clocks using a fail-aware datagram service [19] that calculates an upper bound on the transmission delay of each message it delivers[1]. That allowed us to calculate lower and upper bounds on the remote clock reading errors. With these bounds we calculated lower and upper bounds on the drift rate. To minimize the measurement error, we measured every minute the average relative drift rate over the last 100 minutes. Figure 4 shows a lower and an upper bound for the relative drift rate of two calibrated hardware clocks over a period of more than 2 days.

In summary, 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 of one clock 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 maximum drift rate ρ of $2\frac{\mu s}{s}$ instead of a ρ of about $200\frac{\mu s}{s}$. 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 $(1+\rho)$. We also assume that the clock granularity G is negligible.

A.2 Clock Failure Assumption

We assume that each non-crashed process has access to a correct hardware clock, i.e. has access to a hardware clock with a

---

[1]The service requires an upper bound on the drift rate of a clock. We chose that constant based on measurements of the (absolute) drift rate of calibrated clocks using an externally synchronized clock. Hence, there is no "circularity" in this measurement.
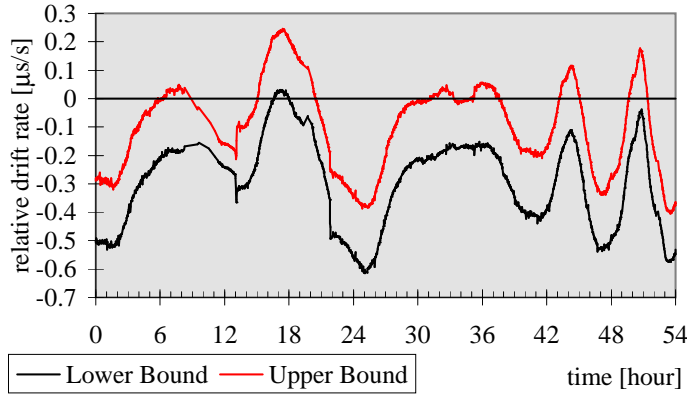
Fig. 4. *The relative drift rate between two calibrated hardware clocks stayed within about $1\frac{\mu s}{s}$ during our measurements.*

drift rate of at most ρ. This assumption simplifies applications since they have to deal with crash failures anyhow but they do *not* have to deal with faulty clocks like "fast" or "slow" clocks. Let predicate $crashed_p^t$ be true iff process p is crashed at real-time t. Formally, we can express this clock assumption (CA) as follows:

$$(CA)\forall p, \forall t : \neg crashed_p^t \Rightarrow correct_{H_p}^t.$$

In practice, one can actually weaken this assumption in the following sense. If a hardware clock H_p fails at time s and a process p tries to read the clock at $t \geq s$, p crashes at t before an incorrect clock value is returned to p. Since p does not read any incorrect clock information, this relaxed assumption is actually equivalent to (CA). In particular, no process can determine that H_p failed. One can implement this relaxed assumption by detecting clock failures at lower protocol levels (transparent to application processes) and transform them into process crash failures.

There are two basic real-time clock implementations in operating systems: 1) an oscillator increments a long hardware counter (typically, 64 bit long) and the value of the real-time clock is the current value of the hardware counter, and 2) a periodic timer is used to increment a software counter (= value of real-time clock). In the first case, the properties of the clock are determined by the physical properties of the oscillator. In the second case, interrupt priorities might affect the properties of the real-time clock. In most systems, the timer interrupt has the highest priority and these systems do not loose timer interrupts. However, there exist a few systems in which other interrupt sources (e.g. the serial line) have a higher priority than the timer interrupt. These systems might loose timer interrupts, i.e. these clocks can go slower if there are too many interrupts of higher priority.

For most systems one can find a reasonable ρ such the probability that a hardware clock fails (i.e. its drift rate is not bounded by ρ) is very low. Whether this probability can be classified as negligible depends on the stochastic requirements of the application (see [28], [5] for an explanation of what failures can be neglected). If this probability is negligible, one does not even have to detect clock failures. However, if the requirements of an application are too stringent to neglect the probability that a single hardware clock fails, one can use redundant hardware clocks

to make sure that the clock failure assumption is valid.

We showed in [20] how one can use commercial off the shelf components to build fault-tolerant clocks to make clock failures negligible. For example, one can use two redundant hardware clocks (two 64 bit counters connected to separate oscillators ; available as PC cards) to detect a single hardware clock failure. The detection (or even the masking of clock failures when at least three hardware clocks are used) can be localized in one *clock reading procedure* so that it becomes transparent to higher level processes. Whenever a process wants to read its hardware clock, the process calls the clock reading procedure and this procedure reads the two redundant clocks. The procedure uses the two values to determine if the relative drift rate of the two clocks is within an acceptable range. If the failures of hardware clocks are independent, one can detect the failure of a clock with a very high probability. Thus, when stochastic application requirements are stringent, redundant clocks allow the detection of clock failures so that the probability that a process reads a faulty clock becomes negligible.

B. Datagram Service

The datagram service provides primitives for transmitting unicast (see Figure 5) and broadcast messages (see Figure 6). 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.
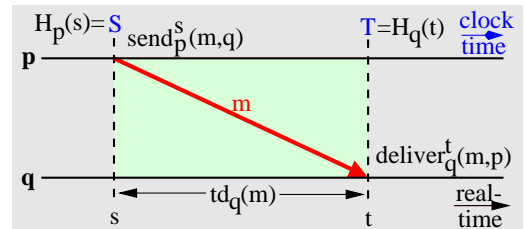

Fig. 5. *Process p sends unicast message m to q at real-time s and q receives m at real-time t. The transmission delay of m is $td_q(m) = t - s$.*

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.
- $send_p^t(m,q)$: 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)*.

Let m be a message that p sends (see Figure 5) or broadcasts (see Figure 6) 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,
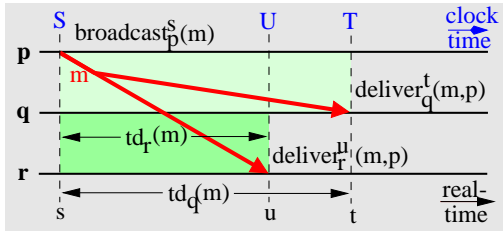
Fig. 6. *Process p sends broadcast message m at s and q receives m at t, while r receives m at u. The transmission delays of m are $td_q(m) = t - s$ and $td_r(m) = u - s$.*

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

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

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

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

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

The requirements for the datagram service (*Validity, No-duplication*, and *Min-Delay*) are defined as follows:

• *Validity:* 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 $s < t$:

$$\forall p, q, m, t: deliver_p^t(m,q)$$
$$\Rightarrow \exists s < t: send_q^s(m,p) \lor broadcast_q^s(m).$$

• *No-duplication*: each message has a unique sender and is delivered at a destination process at most once.

$$\forall p, q, r, m, t, u: deliver_p^t(m,q) \land deliver_p^u(m,r)$$
$$\Rightarrow q = r \land t = u.$$

• *Min-Delay:* We assume that any message m 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 *Min-Delay* requirement does *not* restrict the minimum transmission delay of a message n sent between two *local* processes: the transmission delay of n can be smaller than δ_{min}. The intuition of δ_{min} is that when knowing the minimum message size and the maximum network bandwidth, one knows a lower bound on the message transmission delay. One can use δ_{min} to improve the calculated a posteriori upper bound on the transmission delay of remote messages: the tighter δ_{min} is to the "real" minimum transmission delay, the tighter the a posteriori upper bound gets (see [19] for details). However, if δ_{min} is chosen too big (i.e. some remote messages have transmission delays of less than δ_{min}), one might calculate a bound that is too small. Since the network configuration of a system might change during its lifetime, the safest choice is to assume that $\delta_{min} = 0$.

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 messages sent or broadcasted are *likely* [5] to be delivered within δ. A message with a transmission delay of less than δ_{min} is called *early* (see Figure 7). In the timed model we assume that there are *no* early messages, i.e. δ_{min} is well chosen (see *Min-Delay* requirement). A message m whose transmission delay is at most δ, i.e. $\delta_{min} \leq td_q(m) \leq \delta$, 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, m is *late*). If a message is never delivered, we say that m suffered an *omission failure* (or, m is *dropped*).
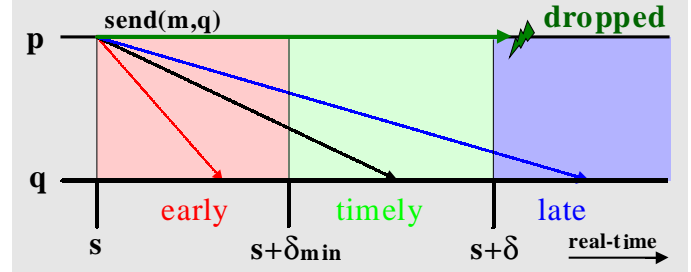

Fig. 7. *In the timed model a message can either be timely, late or* dropped. *The timed model assumes that no message is* early.

B.1 Measurements

The timed model assumes the existence of a one-way time-out delay δ that is used to define message performance failures. The choice of δ determines the frequency of message performance failures. The timed model does not put any upper bound on the frequency of failures and hence, the choice of δ does not affect the correctness of protocols designed for the timed model. We described in [17] several techniques that allow the timely detection of message and process performance failures to be able to detect when certain application properties do not hold anymore. In [19] we describe a mechanism that allows a receiver to detect if a message is timely.

A "good" selection of δ might be system and application dependent. First, for some applications the choice of δ can be naturally derived from the application requirements. For example, 1) an application might have to achieve "something good" within D time units, and 2) the protocol used to implement the application can achieve something good within, say, $k\delta$ time units (in case the failure frequency is within some given bound). In this case, it makes sense to define δ by $\delta \triangleq \frac{D}{k}$.

Second, other applications might not constrain δ. From a practical point of view for these applications a good choice of δ is crucial for protocol stability and speed: 1) choosing a too small δ will increase the frequency of message performance failures and hence, the quality of service might degrade more often, and 2) choosing a too large δ might increase the response time of a service since service time-outs take longer. The choice of a good δ is not always easy since message transmission delays increase with message size (see Figure 8) and with network load (see Figure 9), and also depend on the message transmission pattern used by a protocol (see Figure 10). The determination of a good δ that ensures likely stability and progress might require the measurement of protocol specific transmission delays.

We performed all our measurements on a cluster of 9 Sun IPX workstations connected by a 10Mbit Ethernet in our Dependable Systems Laboratory at UCSD. Seven of these computers run SunOS 4.1.2 while 2 machines run Solaris 2.5. The measurement programs use different services provided by the FORTRESS toolkit [18]. FORTRESS uses UDP for interprocess communication.
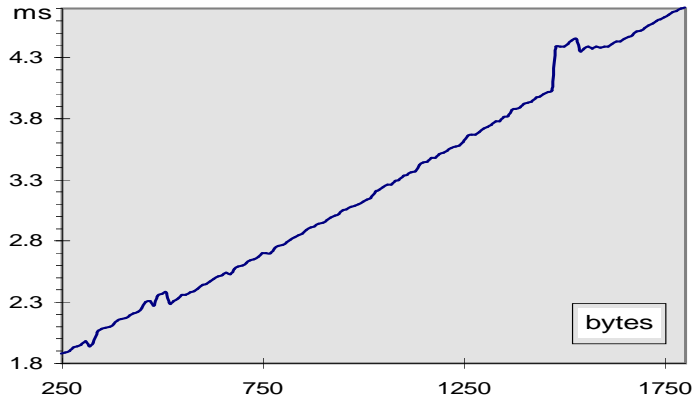
Fig. 8. *Measured minimum delay of round-trip message pairs for different message sizes. We used 20,000 round-trips for each of the 156 measured message sizes.*

To model the dependence of message transmission times on the message size (see Figure 8), we could replace constants δ and δ_{min} by two functions that increase with the size of a message. We actually use in [19] such a function for δ_{min}, i.e. δ_{min} increases with the size of a message. This tighter lower bound allowed a receiver to calculate a better upper bound for the transmission delay of a received message. For simplicity, we however assume in the timed model that δ and δ_{min} are constant.
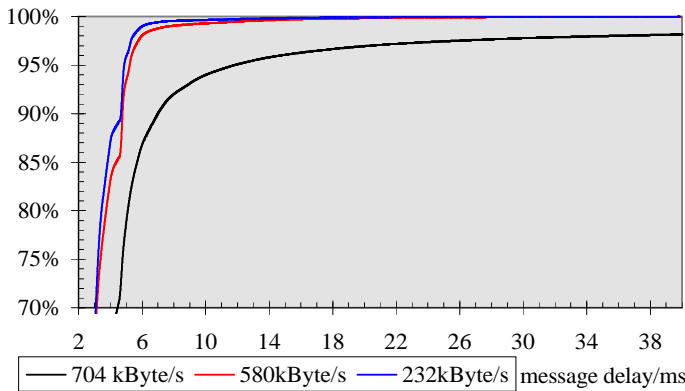

Fig. 9. *The transmission delay of messages increases with the network load. For a network load of 232 kByte/sec, 99% of the messages were delivered within 6.0ms and for 580 kByte/sec within 7.4 ms. For a network load of 704 kByte/sec, less than 99% of the messages were delivered.*

We measured the distribution of message transmission times for different network loads. During these measurements we used 8 Sun IPX workstations connected by a 10Mbit Ethernet. The workstations were grouped into 4 pairs and the two processes of a pair sent each other ping-pong messages of size 1448bytes (without UDP header). We estimated the network load to be the average number of bytes the 8 workstations sent per second. As expected, the likelihood that a message is delivered within some given time decreases with the network load (see Figure 9). Our measurement showed that the minimum experienced message transmission delay can slightly *decrease* with an increase in network load. This can be explained by a decrease of cache misses for the network protocol code with an increase in network traffic.

To demonstrate that transmission delays can be very protocol dependent, we measured the transmission times experienced by a local leadership service [13]. This measurement involved one process p periodically broadcasting messages and five processes sending immediate replies to each message of p. After receiving a reply, p spends some time processing it before receiving the next reply. Hence, the transmission delays of 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 10).
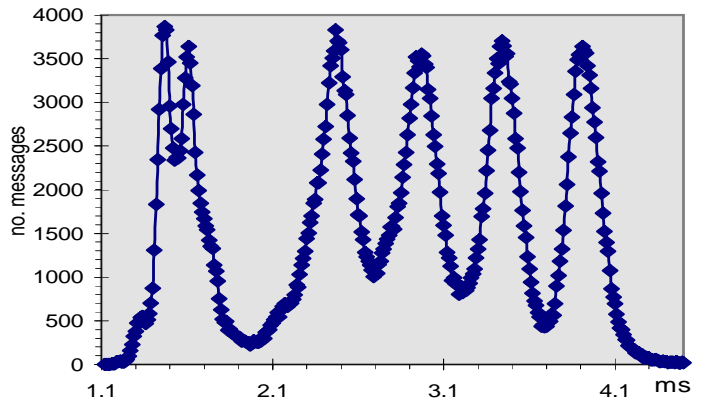

Fig. 10. *Distribution of the transmission delays of unicast messages sent by a local leader election protocol. This distribution is based on 500,000 replies.*
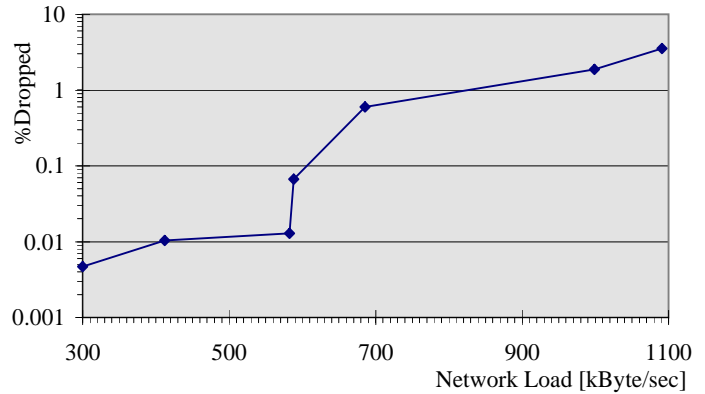

Fig. 11. *Likelihood that a message is dropped with respect to the network load.*

The network and the operating system can drop messages. For example, on Ethernet based systems a message can be dropped because of a message collision on the cable or because of a checksum error. However, it is more likely that the operating system drops messages because of buffer overruns, which occur when it cannot deliver the messages fast enough to the receiving processes. We measured the likelihood that a UDP packet is dropped for different network loads. We used the same setup as for the network load measurements: 8 Sun IPX workstations grouped into 4 pairs. During a measurement the processes of each pair send each other ping-pong unicast messages with a fixed scheduled wait time between the sending of messages. We changed the network load by changing the scheduled wait time. With a network load of about 1090 kByte/sec, approximately 3.5% of the messages were dropped, while for a load of about 300 kByte/sec this decreased to about 0.003% (see Figure 11).
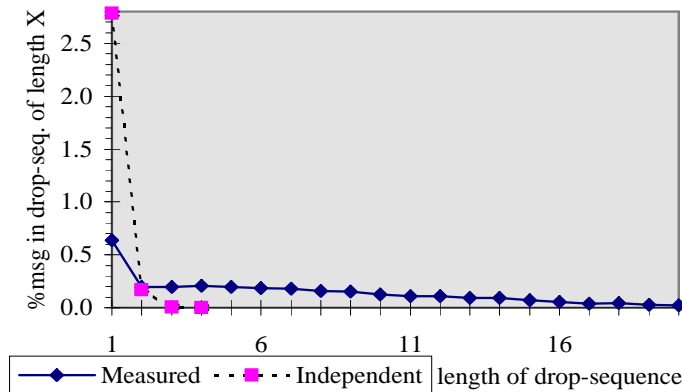
Fig. 12. *Message omission failures are not independent.*

We also performed experiments to test if communication omission failures are "independent". We used again 8 computers in the same setup as before. Note that if omission failures were independent, the probability that two consecutive messages are dropped is the square of the probability that a message suffers an omission failure. We sent 3,000,000 messages between 8 processes to have a high network load (about 1000 kByte/sec). About 2.9% of these messages suffered omission failures. During this measurement, one process experienced the drop of 53 consecutive messages that were sent by some other process. Figure 12 shows the measured likelihood that a message is part of a sequence of $X \in \{1, .., 20\}$ consecutively dropped messages. We also plotted how the curve would look if omission failures were independent ($X \in \{1, .., 4\}$). This shows that message omission failures are not independent.

B.2 Datagram Failure Assumption

We did various experiments to test if our system detects message corruption. So far we have sent more than 10^8 messages with a known "random" contents. The system has not delivered any corrupted messages. We also tested if there were any message duplications. None of the 10^8 messages were delivered more than once. The probability of undetected message corruption and duplication might not be negligible in all systems. However, one can use an additional software layer to reduce this probability to the degree that it becomes negligible. This software layer can be transparent to the processes, i.e. the processes do not need to know of its existence.

Source address spoofing occurs when a process p sends a message m to some process q and makes q believe that a different process $r \neq p$ has sent m. The *validity assumption* implies that we assume that the probability of source address spoofing is negligible. When one cannot neglect this probability, one can use message authentication [30] to reduce this probability so that it becomes negligible. This can be done in a manner transparent to the processes. Note that message authentication can increase the transmission time substantially if there is no special hardware assistance.

In summary, the asynchronous datagram service is assumed to have an omission/performance failure semantics [5]: it can drop messages and it can fail to deliver messages in a timely manner, but one can neglect the probability of source address spoofing

and that a message delivered by the system is corrupted or is delivered multiple times. 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, δ_{min} and δ are such small quantities, we equate $(1-\rho)\delta$ and $(1+\rho)\delta$ with δ and $(1-\rho)\delta_{min}$ and $(1+\rho)\delta_{min}$ with δ_{min}.

C. Process Management Service

C.1 Process Modes

A process p can be in one of the following three modes (see Figure 13):
- *up*: p is executing its 'standard' program code,
- *crashed*: a process stopped executing its code, i.e. does not take the next step of its algorithm 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 13):
- *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. While p is crashed, it cannot execute any step of its algorithm. We define the predicate *recovering*$_p^t$ to be true iff p is recovering at time t. A process can be recovering only as a consequence of a start or recover event occurrence.
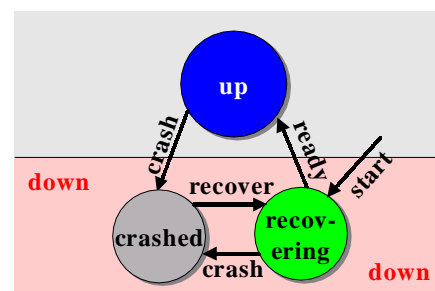

Fig. 13. *Process modes and transitions.*

C.2 Alarm Clocks

A process p can set an *alarm clock* to be awakened at some specified *future* clock time[2]. 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. We call T an alarm time. Process p will not take a step after it sets its alarm clock to T unless 1) it is awakened for alarm time T, or 2) it receives

---

[2]We assume that a process is signaled an error in case it requests to be awakened at a time that has already passed. To avoid this, a process can define a positive alarm time relative to the current time instead of specifying an absolute alarm time.

a message before it is awakened for alarm time T. We assume that if a process p sets its alarm clock before it was awakened for its previous alarm time T (this can only happen if it receives a message before it is awakened for T), alarm time T is overwritten, i.e. p will not be awakened for alarm time T. In other words, at any time a process p can have at most one *active* alarm time, i.e. an alarm time that has not been overwritten and for which p has not been awakened. Note that a process can maintain multiple alarm times based on the alarm clock provided by the timed asynchronous system model.

We use the following predicates to specify the behavior of an alarm clock:

• $SetAlarm_p^s(T)$: p requests at real-time s to be awakened at some future real-time u such that $H_p(u) \geq T$, i.e. p wants to take its next step when its hardware clock shows at least value T unless it receives a message before T, and

• $WakeUp_p^u(T)$: the process management service wakes up p at real-time u for the *alarm clock time T*.

When a process p crashes, the process management service forgets any active alarm time p has set before crashing. If p never crashes, we assume that it will eventually be awakened for all active alarm times. The behavior of an alarm clock is constrained by the following requirement (AC = alarm clock): a process p is awakened for an alarm clock time T at real-time u only if 1) its hardware clock shows at least T at u, 2) p has requested at some previous time $s < u$ to be awakened at T, and 3) within time interval $(s, u]$, process p has not crashed, has not overwritten the alarm time and has not been awakened for T since s. Formally, the (AC) requirement is expressed as follows:

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

Let t be the earliest real-time (i.e. smallest value) 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 process p. The process management service does not ensure the existence of an upper bound on scheduling delays. However, being a timed service, we define a scheduling timeout delay σ, so that actual scheduling delays are likely [5] to be smaller than σ. Since ρ and σ are such small quantities, we equate $(1 - \rho)\sigma$ and $(1 + \rho)\sigma$ with σ.

We say that a non-crashed process p suffers a *performance failure* when it is not awakened within σ of the last alarm time T it has specified (see Figure 14), i.e. if it is awakened when its local hardware clock H_p shows already a value greater than $T + \sigma$. In this case, we say that p is late. Otherwise, if p is awakened when its hardware clock shows a value in $[T, T + \sigma]$, it is said to be *timely*. If p is awakened for T before H_p shows T, p is said to be *early*. Since it is easy to avoid early timing failures (by checking that $H_p \geq T$ and going to sleep again if $H_p < T$), the timed model assumes that processes do not suffer early timing failures.

Formally, 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:

$pFail_p^u \triangleq \exists s \leq u, \exists T: SetAlarm_p^s(T) \wedge H_p(u) > T + \sigma \wedge \forall v \in (s,u]:$

$\neg WakeUp_p^v(T) \wedge \neg crashed_p^v \wedge \forall S: \neg SetAlarm_p^v(S)$.

We define the predicate $timely_p^u$ to be true iff p is timely at u:

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

We extend the notion of a process p being timely to a time interval I as follows:

$timely_p^I \triangleq \forall t \in I : timely_p^t$.

Note that we do not include the processing time of messages in the definition of a timely process. The reason for that is that – conceptually – our protocols for the timed model add the processing time of a message m to the transmission delay of the messages sent during the processing of m (see [17] for a more detailed description): a too slow processing of messages is therefore transformed into message performance failures.
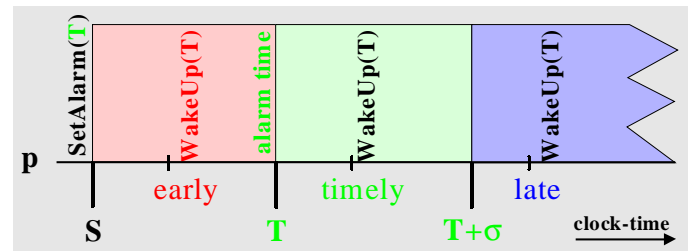

Fig. 14. *Process p is timely if it is awakened within σ ticks of alarm time T. Process p suffers a performance failure (or, is late) if it is awakened after $T + \sigma$. The timed model excludes early timing failures, i.e. a process is never awakened before time T.*

C.3 Measurements

To implement alarm clocks in the Unix family of operating systems, one can use the *select* system call. This call allows the specification of 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. In SunOs this timer has a resolution of $10ms$. Thus, the scheduling delay timeout σ should be chosen to be at least $10ms$. Figure 15 shows the distribution of scheduling delays experienced by a process executing a membership protocol [16]. These measurements where performed during normal daytime use of the system (low load).
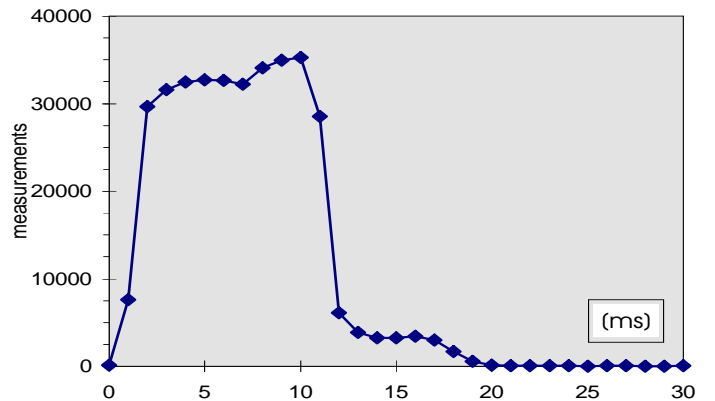

Fig. 15. *The distribution shows the difference between the time a process was awakened and the time it requested to be awakened. It is based on 350,000 measurements.*

C.4 Process Failure Asssumption

The timed model assumes that processes have crash/performance failure semantics [5]. However, the execution of a process might stop prematurely (crash failure) or a process might not be awakened within σ time units of an active alarm time (performance failure). Processes can recover from crashes.

In most applications, the probability that a *processor* executes the program of a process incorrectly is negligible. In systems in which that probability cannot be neglected, one can use redundancy at lower protocol levels to guarantee the crash/performance process failure semantics. For example, consider that *processors* can suffer independent failures. In this case one can use a processor pair to execute the program of a process in lock step. If the two processors disagree about the result of some instruction, both processors stop executing. In this way, processor failures can be transformed into crash failures. This dual processor approach is transparent to the processes. Thus, the assumption that processes have crash/performance failure semantics is reasonable.

IV. 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" (i.e. behave like a synchronous system) 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. We sometimes use the terms "core model" and "extended models" to distinguish between models that include assumptions about stable storage or progress assumptions in addition to the "core" assumptions about datagram service, the process management service, and the local hardware clocks.

A. 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 $addr$, value $\perp$ (undefined) is returned.

The predicates that denote the invocation of the above primitives at some real-time t 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:

$read_p^t(a,val) \Rightarrow$
$\quad \forall u \leq t, v: \neg store_p^u(a,v) \wedge val = \perp$
$\quad \vee \exists s < t: store_p^s(a,val) \wedge \forall u \in (s,t], v: \neg store_p^u(a,v).$

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].

B. 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 $SP \subseteq \mathcal{P}$ is "stable" (i.e. "behaves like a synchronous system"), that is, the failures affecting SP and the communication between them have a bounded frequency of occurrence, the servers in SP have to guarantee progress within a bounded amount of time. We call a set SP a *stable partition* [13] iff

- all processes in SP are timely,
- all but a bounded number of messages sent between processes in SP are delivered timely, 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 [14]. In this paper we formally define the stability predicate Δ-*F-partition* introduced informally in [17]. To do that, we first formalize and generalize the notions of connectedness and disconnectedness introduced in [10].

B.1 Δ-F-Partitions

Two processes p and q 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-connected(p,q,s,t)*:

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

A process p is Δ-*disconnected* from a process q in $[s,t]$ iff any message m that is delivered to p during $[s,t]$ 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 [19] to detect all message that have transmission delay of more than Δ while guaranteeing that no message with a transmission delay of at most δ is wrongly suspected to have a transmission delay of more than Δ. We use the predicate Δ-*disconnected(p,q,s,t)* to denote that p is Δ-disconnected from q in $[s,t]$:

$\Delta\text{-}disconnected(p,q,s,t) \stackrel{\Delta}{=} \forall m, \forall u \in [s,t] :$

$deliver_p^u(m) \wedge sender(m) = q \Rightarrow td_p(m) > \Delta.$

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:

Δ-*partition(S, s, t)* $\overset{\Delta}{=} S \neq \emptyset$
  $\wedge \forall p, q \in S : $F-connected(p,q,s,t)
  $\wedge \forall p \in S, \forall r \in \mathcal{P}\text{-}S: \Delta$-disconnected(p,r,s,t).

As an example of the utility of the above stability predicate, consider an atomic broadcast protocol designed to achieve *group agreement* semantics [6], 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.

B.2 Progress Assumptions

The lifetime of most distributed systems 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. This can be explained by the bursty behavior of the network traffic which can cause temporary instabilities. For example, traffic bursts can be caused by occasional core dumps or file transfers via the network. Based on this observation, we introduced the concept of *progress assumptions* [14] 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 $[t, t + \eta]$.
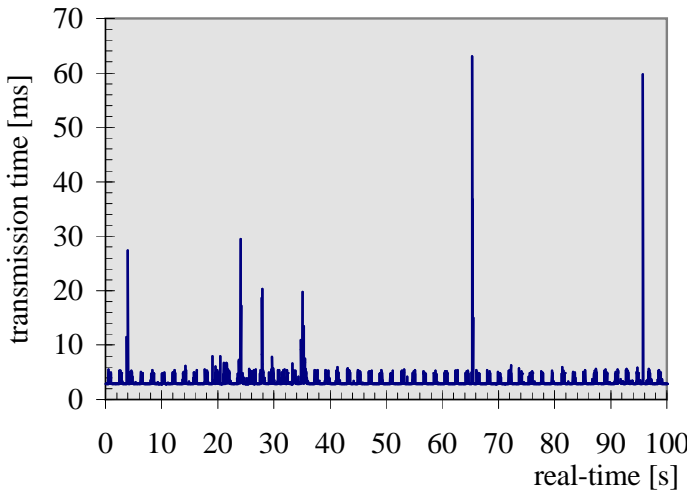

Fig. 16. *The graph shows the transmission delay of messages received by one process over a period of 100 seconds. The transmission delay of messages stayed for long periods of time well below 10ms. However, sporadically the delay increased well above 10ms.*

B.3 Measurements

The first measurement shows how transmission delays are distributed over time (see Figure 16). We used in this experiment 4 processes and each process receives and sends about 36

UDP messages per second. Hence, we sent about 144 messages every second on the Ethernet. Each message contains a payload of 1448 bytes, i.e. we induced a network load of more than 208 KByte/sec.

We also measured the behavior of six processes $\{p_1, ..., p_6\}$ each running on a SUN workstation in our Dependable Systems Lab over a period of a day under normal load conditions (see Figure 17). The set of all six processes were, on the average, "Δ-1-stable" for about $218s$, i.e. the six processes formed a Δ-F-partition with $F = 1$, $\Delta = 20ms$ and $\sigma = 30ms$. 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. For this measurement, we used a modified membership service [16]: whenever a process declared that it cannot keep its membership up-to-date or not all six processes stayed in the membership, we knew that $\{p_1, ..., p_6\}$ is not Δ-1-stable. From a theoretical point of view, one cannot determine perfectly if the system is Δ-1-stable. However, one can determine if the system looks to the processes like it is Δ-1-stable – which, from a practical point of view, is equivalent to the system being Δ-1-stable. Note that the membership service allows the fast processes to continue to make progress even if the system is not Δ-1-stable because it can temporarily remove the slow process(es) from the membership. In other words, system instabilities might result in the removal of slow or disconnected processes but in our experience in almost all cases the remaining processes can still provide their safety and timeliness properties.
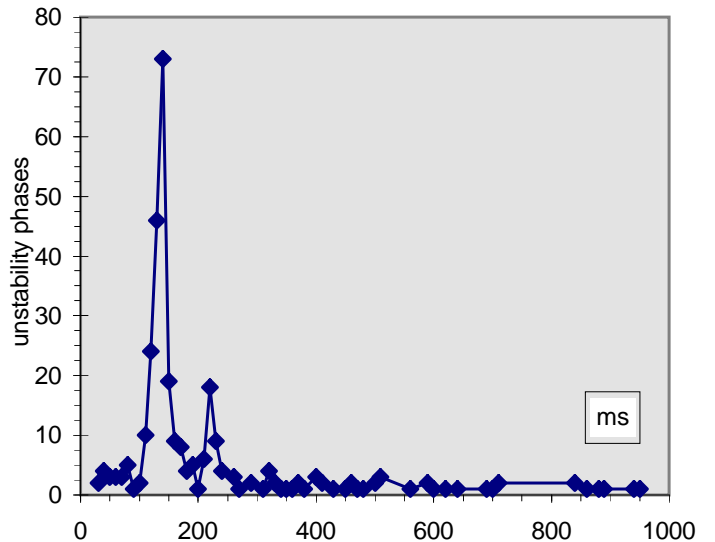
Fig. 17. *Observed time between two Δ-1-stable periods for six processes, where $\Delta = 20ms$ and $\sigma = 30ms$ during a period of 24 hours. The typical failure behavior observed between consecutive stability periods was that one process was slow.*

V. COMMUNICATION BY TIME

In synchronous systems, the communication by time (i.e. communication of information achieved by measuring the passage of time) is very important. For example, if a correct process p does not hear in time the 'I-am-alive' message of q, then p knows that q has crashed. The communication uncertainty

that characterizes timed asynchronous systems makes "communication by time" more difficult, but it is (in a more restricted form) still possible. For example, in the timed model if p does not hear from q in time the 'I-am-alive' message of q, p *does not* know that q has crashed. However, p knows that q or the 'I-am-alive' has suffered a failure. In many applications this is sufficient since p only cares about if or if not it can communicate with q in a timely manner. For example, in a leader election protocol, process p might only support the election of q as long as p can communicate with q in a timely manner and otherwise, it might try to support the election of another process (with which it can communicate in a timely manner). However, in this leader election example we have to make sure that there is at most one leader at a time. In synchronous systems enforcing this property is straight forward since processes can detect perfectly if the current leader has crashed and hence, when to replace it by a new leader. In asynchronous systems enforcing this property is not that easy, since one cannot decide if the current leader l has crash, or is slow, or the communication to l is slow.

We illustrate how two processes p and q can use communication by time to ensure that at any time at most one of them is leader. We use a *locking mechanism* [17], which can be viewed as a leases mechanism [22] for systems without synchronized clocks. This mechanism enables communication by time even when local clocks are not synchronized. This mechanism works as follows:

• p sends some information in a message m to a process q and p says that this information is only valid for a certain amount of time,

• if q receives m, it calculates an upper bound on the transmission delay of m to determine for how long it can use m, and

• p can determine by consulting only its local hardware clock the time beyond which q will no longer make use of the information contained in m.

For concreteness, consider the pseudo-code of Figure 18. In this example, we use communication by time to enforce one correct process p is eventually leader while ensuring that there is only one leader at a time even though q might be leader for a bounded amount of time. To concentrate on the main aspect, i.e. how p can detect that q is not leader anymore, q gets only one chance to become leader by p sending q a message saying that q is leader for a certain amount of time. A complete leader election protocol using a time locking mechanism can be found in [13].

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 [19] calculates an upper bound on the transmission delay of 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+2ρ)+Δ(1+ρ)* clock time units before it becomes leader, where 1) the factor $(1 + 2\rho)$ is necessary because p's and q's hardware clocks can drift apart by up to

2ρ, and 2) the factor $(1+\rho)$ since p's clock can drift apart from real-time by up to $\Delta\rho$ during the maximum transmission delay Δ of a "fast" message m. 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.

```
const      time Duration, Δ;

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*ρ)+Δ(1+ρ);
    SetAlarm(T);
    select event
        when WakeUp(T):
                ExpirationTime = ∞;
    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
```

Fig. 18. This pseudo-code uses communication by time to enforce that a correct p is eventually leader while ensuring that there is only one leader at a time even though q might be leader for a bounded amount of time.

Note that a leader is implicitly demoted by the advancement of its hardware clock. Since a process might be delayed immediately after checking if it is the leader, a demoted process might not immediately detect that it was demoted. However, when *Leader?* is used in a proper way, other processes can detect messages from a demoted leader in the following way:

• A process r first reads its hardware clock and if H_r shows a value T, then

• r determines if it is leader at T by querying function *Leader?* for time T,

• if r is leader at T, it does some processing and then sends some message n and it sets the send time-stamp of n to T, and

• a process receiving n will calculate the transmission time of n based on T, i.e. delays of r are added to the transmission time of n.

For example, if r is swapped out after reading *Leader?* and be-

fore sending n, the delay of this swap is added to the transmission delay of n and receiver(s) of n can reject n if the transmission delay of n is too slow. In summary, we typically transform delays of a demoted leader into message performance failures that can be detected by the receivers of the messages.

VI. 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 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) on the use of progress assumptions, and 3) whether the underlying system model allows communication by time. The main intuition of 1) is that one can weaken a problem such that one has only to solve the problem when the system is "well behaved", or 2) one can require instead that the system be "well behaved" from time to time and hence, one can solve the problem while the system is well behaved, and 3) if one can use communication by time to circumvent the impossibility that one cannot decide perfectly if a remote process is crashed, e.g. if one can use a local hardware clock to decide if a time quantum of a remote process has expired.

A. Termination Vs Conditional Timeliness Conditions

There is no commonly agreed-upon rigorous specification for the election problem. For example, [29] 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.
Typically, problems specified for timed systems do not use such strong unconditional *termination conditions* (like (TF)) requiring that "something good" eventually happens. Instead, we use *conditional timeliness conditions*. 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 [13] 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 [29], while (S, TT) is implementable in timed systems [10], [13]. 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 V). 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 processes is timely and 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 [13].

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 cannot communicate with any other process (in R).

B. 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 (S, TT) in 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 *enables communication by time* between processes, that allows us to circumvent in the timed model the impossibility result of [29] stated for the time-free model.

C. 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 systems that one encounters in practice make such behavior extremely unlikely when δ and σ are 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 IV-B.2), 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.

VII. CONCLUSION

We have given a rigorous definition of the timed asynchronous system model. Based on the measurements reported previously, performed on the network of workstations in our Dependable Systems Laboratory, and on other unpublished measurements at other labs that we are aware off, 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.

Most real-world applications have soft real-time constraints. Hence, such applications need a notion of time. Neither the original time-free model [21] nor its extension with failure detectors [3] provides that. These models are therefore not necessarily an adequate foundation for the construction of applications with soft real-time constraints. The timed model instead provides these applications with a sufficiently strong notion of time. The timed model is also a good foundation for the construction of fail-safe hard real-time applications (see [17], [12]).

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.

[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. In *Proceedings of the Fifteenth International Symposium on Fault-Tolerant Computing*, pages 200–206, Ann Arbor, MI, Jun 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, Dept of Computer Science and Engineering, University of California, San Diego, La Jolla, CA, 1995.

[11] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, Feb 1988.

[12] D. Essame, J. Arlat, and D. Powell. Padre: A protocol for asymmetric duplex redundancy. In *Proceedings of the Seventh IFIP International Working Conference on Dependable Computing for Critical Applications*, San Jose, USA, Jan 1999.

[13] C. Fetzer and F. Cristian. A highly available local leader service. *IEEE Transactions on Software Engineering*. To appear in 1999. http://www.cs.ucsd.edu/~cfetzer/HALL.

[14] 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. http://www.cs.ucsd.edu/~cfetzer/CONS.

[15] C. Fetzer and F. Cristian. Fail-awareness in timed asynchronous systems. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing*, pages 314–321a, Philadelphia, May 1996. http://www.cs.ucsd.edu/~cfetzer/FA.

[16] C. Fetzer and F. Cristian. A fail-aware membership service. In *Proceedings of the 16th Symposium on Reliable Distributed Systems*, Oct 1997. http://www.cs.ucsd.edu/~cfetzer/FAMS.

[17] C. Fetzer and F. Cristian. Fail-awareness: An approach to construct fail-safe applications. In *Proceedings of the 27th Annual International Symposium on Fault-Tolerant Computing*, Seattle, Jun 1997. http://www.cs.ucsd.edu/~cfetzer/FAPS.

[18] C. Fetzer and F. Cristian. Fortress: A system to support fail-aware real-time applications. In *IEEE Workshop on Middleware for Distributed Real-Time Systems and Services*, San Francisco, Dec 1997. http://www.cs.ucsd.edu/~cfetzer/FORTRESS.

[19] C. Fetzer and F. Cristian. A fail-aware datagram service. In D. Avaresky and D. Kaeli, editors, *Fault-Tolerant Parallel And Distributed Systems*, chapter 3, pages 55–69. Kluwer Academic Publishers, 1998. http://www.cs.ucsd.edu/~cfetzer/FADS.

[20] C. Fetzer and F. Cristian. Building fault-tolerant hardware clocks. In *Proceedings of the Seventh IFIP International Working Conference on Dependable Computing for Critical Applications*, San Jose, USA, Jan 1999. http://www.cs.ucsd.edu/~cfetzer/HWC.

[21] 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.

[22] C. G. Gray and D. R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 202–210, Dec 1989.

[23] L. Lamport and N. Lynch. Distributed computing: Models and methods. In *Handbook of Theoretical Computer Science*, pages 1158–1199. Elsevier Science Publishers, 1990.

[24] J. Lundelius Welch. Simulating synchronous processors. *Information and Computation*, 74(2):159–71, Aug. 1987.

[25] D. L. Mills. Internet time synchronization: the network time protocol. *IEEE Trans. Communications*, 39(10):1482–1493, Oct 1991.

[26] D. L. Mills. Modelling and analysis of computer network clocks. Technical Report 92-5-2, University of Delaware, Electrical Engineering Department, May 1992.

[27] G. Neiger and S. Toueg. Simulating synchronized clocks and common knowledge in distributed systems. *Journal of the Association for Computing Machinery*, 40(2):334–67, Apr. 1993.

[28] D. Powell. Failure mode assumptions and assumption coverage. In *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing Systems*, pages 386–395, 1992.

[29] L. Sabel and K. Marzullo. Election vs. consensus in asynchronous systems. Technical Report TR95-1488, Cornell University, Feb 1995.

[30] B. Schneier. *Applied Cryptography*. John Wiley, 1996.

[31] 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.

[32] P. Verissimo, L. Rodrigues, and M. Baptista. Amp: A highly parallel atomic multicast protocol. In *SIGCOMM'89*, pages 83–93, Austin, TX, Sep 1989.

Flaviu Cristian is Professor of Computer Science at the UC San Diego. He received his PhD from the University of Grenoble, France, in 1979. He joined IBM Research in 1982. While at IBM, he worked in the area of fault-tolerant distributed systems and protocols. After joining UCSD in 1991, he and his collaborators have been designing and building support services for providing high availability in distributed systems. Dr. Cristian has published over 100 papers in international journals and conferences in the field of dependable systems.

Christof Fetzer received his diploma in computer science from the University of Kaiserlautern, Germany (12/92) and his Ph.D. from UC San Diego (3/97). He is currently a research scientist at UC San Diego and he will join AT&T Labs in 1999. He received a two-year scholarship from the DAAD and two best student paper awards. He was a finalist of the 1998 Council of Graduate Schools/UMI distinguished dissertation award. Dr. Fetzer has published over 25 research papers in the field of distributed systems.