Fail-Awareness in Timed Asynchronous Systems

Christof Fetzer and Flaviu Cristian
Department of Computer Science & Engineering
University of California, San Diego
La Jolla, CA 92093−0114*
e-mail: {cfetzer,flaviu}@cs.ucsd.edu
http://www-cse.ucsd.edu/users/{cfetzer,flaviu}

CS95-453

May 23, 1996

Abstract

We address the problem of the impossibility of implementing synchronous fault-tolerant service specifications in asynchronous distributed systems. We introduce a method for weakening a synchronous service specification so that it becomes implementable in "timed" asynchronous systems, that is, asynchronous systems in which processes have access to local hardware clocks. The method (1) adds to a service interface an exception indicator so that a client knows at any time if a server is currently providing its standard "synchronous" semantics or some other specified exceptional semantics, (2) the standard behavior provided when the exception indicator does not signal an exception is "similar" to the original synchronous service behavior, and (3) a server has to provide its standard semantics whenever the underlying communication and process services exhibit "synchronous behavior". To illustrate our method, we show how the specification of a synchronous datagram service and an internal clock synchronization service can be transformed into a *fail-aware* service specification. Further illustrations of the usefulness of fail-aware services are provided by describing a railway crossing service and a fail-aware weak group membership service.

*This research was partially supported by a grant from the Air Force Office of Scientific Research. An earlier version of this technical report was published in the *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing, May 1996, Philadelphia.*

1 Introduction

Synchronous fault-tolerant services are services defined for synchronous systems: they are always required to meet their deadlines, provided no more than a bounded number of failures occur [9]. Examples are internal clock synchronization [18] and synchronous atomic broadcast [7], and membership services [8]. The implementation of such services depends critically on the existence of an upper bound for message transmission and process scheduling delays. Synchronous fault-tolerant services cannot be implemented in asynchronous systems which are characterized by unbounded communication and scheduling delays. In this paper we address the problem of how to change the specification of a synchronous service so that it becomes implementable in asynchronous distributed systems in which processes have access to local hardware clocks.

Hardware clocks and the notion of "performance failures" are essential for our approach. This work is therefore based on the *timed asynchronous system model* [11] instead of the *time-free* model [17]. The timed asynchronous system model (see section 4) is asynchronous in the sense that it does not require the existence of upper bounds for message transmission and scheduling delays. Processes have however access to local hardware clocks and the model defines time-outs for message transmission and scheduling delays to allow the definition of *performance failures*. A performance failure occurs when an experienced delay is greater than the associated time-out delay.

This paper proposes a design concept, which we call *fail-awareness*, for weakening the specification S of a synchronous service to a new specification FS, so that FS becomes implementable in timed asynchronous systems. The basic idea is to make the FS service *fail-aware*: when FS cannot satisfy the original synchronous specification S because of the occurrence of "too many" performance failures, it will send an exception signal to affected clients. *Fail-awareness* is therefore a design concept that weakens a specification such that it becomes implementable in timed asynchronous systems by allowing performance failures to be translated into exception occurrences. The transformation of a synchronous specification S into a fail-aware specification FS is done in three steps: (1) the interface of FS is augmented with an exception indicator, (2) a FS server is required to provide its standard semantics which is "similar" (see section 8) to the synchronous semantics S whenever its indicator does not signal an exception, and (3) whenever the underlying communication and process services of a server exhibit "synchronous behavior" (see section 6.1), the indicator of this server does not signal an exception.

Fail-aware services are implementable in timed asynchronous systems because they are only required to provide their standard semantics when the underlying communication and process services exhibit enough "synchronous behavior" to allow this. The intuition behind the implementation of the exception indicator is that a server can only succeed to switch its exception indicator off when the underlying communication and process services exhibit a sufficient amount of "synchronism" and this amount of "synchronism" is enough to provide the specified standard semantics. Fail-awareness is a general concept which can be applied to various synchronous services such as synchronous datagram, internal clock synchronization and synchronous broadcast and membership services. A mechanical transformation of a synchronous service specification into a fail-aware service specification does not seem to be desirable nor possible because sometimes it is advantageous that the standard semantics be only "similar" to the semantics of the synchronous service (see section 8).

We illustrate the concept of fail-awareness by demonstrating how to specify and construct a fail-aware internal clock synchronization and a fail-aware datagram service. The specification of internal clock synchronization is transformed by (1) adding a "synchronized" indicator, and requiring that (2) the deviation between two clocks be bounded whenever their indicators are true, and (3) any clock which can communicate with a majority of clocks in a "timely manner" has an indicator which is true. Fail-awareness is not restricted to the transformation of synchronous services into fail-aware services, e.g. we applied it to define *fail-aware* failure detectors [16].

Most fault-tolerant applications are *timed* in the sense that they are required to react to inputs by certain deadlines. We are interested in supporting the implementation of timed, fault-tolerant applications in asynchronous systems by providing fail-aware services. In particular, we are targeting systems consisting of local networks of computers running soft real-time operating systems. Such systems do not guarantee that services will meet their deadlines "all the time", but promise that deadlines will be met "most of the time". However, missing a deadline can result not only in the violation of timeliness properties, but also in the violation of safety properties. A fail-aware server *knows* when it cannot provide its standard semantics and signals such events to the affected clients by an exception indicator. In this way, the clients can take appropriate recovery actions at their level of abstraction, e.g. in a manner similar to the technique of hierarchical exception masking described in [9]. Using a simple railway crossing example, we show how fail-aware services can be used to enforce the safety properties of an application.

2 Related Work

Many problems such as consensus [17] and weak membership [3] cannot be solved in asynchronous systems. Several approaches to overcome this problem have been proposed: (a) the usage of randomization [2], (b) the introduction of partially synchronous models [12, 14], failure detectors [4], and progress assumptions [15], and (c) the investigation of weaker problems [13, 1]. Fail-awareness is a method for transforming problems into weaker problems such that they become implementable in timed asynchronous systems. However, fail-awareness can be combined with progress assumptions to strengthen the properties of fail-aware services without changing their protocols (see section 8).

In this work we use the timed asynchronous system model [11] instead of the time-free asynchronous system model [17]. The timed model assumes that each process has access to a local hardware clock and defines time-outs for message transmission and process scheduling delays. The time-outs are the foundation for specifying what it means for a system to exhibit "synchronous behavior". In particular, they allow the introduction of performance failures and to give the a priori informal terms such as "timely process" and "timely message" a precise meaning. The timed model is asynchronous in the sense that scheduling and message transmission delays are unbounded. It is stronger than the time-free model with respect to the introduction of local hardware clocks. However, all existing distributed systems we are aware of provide local hardware clocks, so this is not a practical restriction. The timed model is even weaker than the time-free model with respect to the assumed communication service. It assumes a datagram service with omission/performance failures instead of a reliable unicast service. Such a datagram service allows that in certain runs correct processes are partioned forever, while a reliable unicast service excludes such runs. Every system which can be described in the time-free model and which has local hardware clocks can therefore be modeled as a timed system by defining "appropriate" time-out delays.

The protocol for a fail-aware datagram service is based on an idea which underlies probabilistic remote clock reading [6], namely that the measurement of round trip message delays allows the calculation of an upper bound of the transmission delay of messages. In probabilistic clock reading, knowledge of the round trip delay allows a slave clock to calculate an upper bound for the deviation between its clock and the clock of the master, and hence, allows the slave clock to *know* if it is synchronized or not [6]. Probabilistic clock synchronization can therefore be viewed as an early example of a fail-aware distributed service. An example of the usage of an indicator in distributed services are the membership services of [8, 11] each of which has a "joined" indicator telling the service clients whether they are joined to a group or not.

Most internal clock synchronization protocols, e.g. [18, 19], mask arbitrary failures, in the sense that correct clocks are synchronized when less than a third of the clocks suffer failures. Our proposed fail-aware clock synchronization service handles performance, omission, and crash failures but does not tolerate arbitrary failures. An extension to mask a bounded number of arbitrary failures could be done in a way similar to the protocols described in [10]. Performance failures are in a certain sense "more difficult" than arbitrary failures because they are not restricted to some subset of processes, i.e. two processes can fail to read different sets of clocks within some a priori given reading error. In particular, not every clock synchronization algorithm which masks F arbitrary clock/process failures can handle F independent clock reading errors per process. However, performance failures can be detected and can therefore be "more easily" masked than arbitrary failures. The fail-aware clock synchronization protocol is based on the protocols introduced in [10].

3 Example: Railway Crossing

One possible application domain of fail-aware services is the implementation of timed, fault-tolerant applications. Obviously, not all timed, fault-tolerant applications can be implemented in timed asynchronous systems. However, many can be partitioned into a soft real-time subsystem (SRTS) and a hard real-time kernel (HRTK) such that SRTS can be implemented on a timed asynchronous system and HRTK on a synchronous system. This partitioning can in particular be used for timed applications which have a "safe" or "degraded" mode such that the HRTK can switch to this mode whenever the SRTS suffers "too many performance failures". To illustrate this idea, we consider a gated railway crossing which is controlled by a distributed system consisting of several sensor and gate actuator units connected to several controllers via a network (see figure 1). The crucial safety property of this application is that the gates must be down whenever a train is in the railway crossing. Its main liveness property is that the gates have to be open when no train is in the crossing and the system exhibits "synchronous behavior". We show how this system can be partitioned: (1) the sensors and controllers make the SRTS, and (2) the gate units form the HRTK. When the HRTK switches to its safe mode, it closes the gates. The HRTK switches to its safe mode iff the information the SRTS presents to the HRTK could be out of date because the SRTS has suffered too many performance failures.

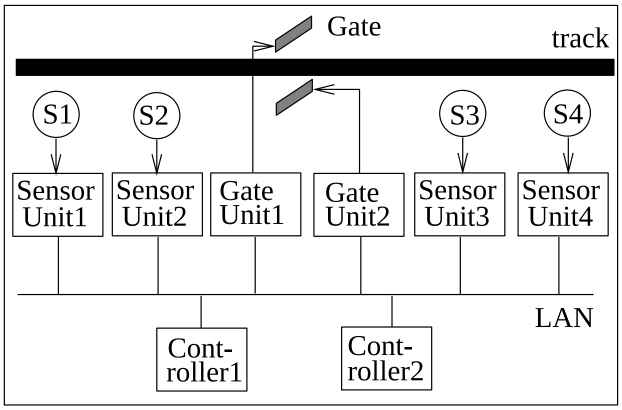

Figure 1: Architecture of railway crossing application.

The sensor units periodically broadcast their sensor state readings to the controllers. The controllers use these broadcasts to update their approximation of the state of the sensors. A controller periodically scans this approximation to open or close gates: it sends a "Gate Down" command message when at least S_{down} sensors tell that a train is approaching the railway crossing, otherwise it sends a "Gate Up" command when at least S_{up} sensors tell that no train is approaching. A gate unit closes its gate whenever it receives an up-to-date "Gate Down" command and opens its gate on the reception of an up-to-date "Gate Up" command. A gate unit switches to its safe mode whenever it has not received an up-to-date command for, say, TO time units.

A major safety hazard is that out-of-date information caused by slow messages and slow sensor or controller processes could result in situations where the gates are open when a train is in the crossing. For example, the usage of a slow sensor message saying that no train is approaching could result in the wrong decision of a controller to open the gates. To prevent such wrong decisions, it is sufficient that a controller ensures that it only uses information contained in *standard* messages, that is, messages delivered at most some Δ time units after the sensor was read. We introduce in this paper a *fail-aware datagram service* which allows the receiver of a message m to determine if m's transmission delay is at most Δ time units. Controllers and gate units can use the fail-aware datagram service to reject all messages with a transmission delay of more than Δ time units.

To mask controller performance failures and slow command messages from a controller to a gate unit, our design uses multiple controllers to send commands to the gate units in an "interleaved manner". A simple way to coordinate the actions of the controllers would be the usage of internally synchronized clocks to define a global time grid. This time grid could ensure that at any point in time there is at most one controller responsible for sending commands to the gate units. Internal clock synchronization is not implementable in timed asynchronous systems. However, we can use *fail-aware internal clock synchronization* to define a global time grid in the following way: each controller is assigned a sequence of unique time slots and a controller p can send commands to the gate units whenever its clock says that it is synchronized and its value is within one of p's time slots.

4 Timed Asynchronous Systems

A timed asynchronous system [11] is a distributed system consisting of a finite set of processes linked by an asynchronous datagram service. A one-way time-out delay δ is defined for the datagram service. Although there is no guarantee that a message will be delivered within δ time units, a message is *likely* to be delivered within δ. We say that a process receives a message m in a *timely manner* if the transmission delay of m is not greater than δ. When the transmission delay of m is greater than δ, we say that m has suffered a performance failure. The asynchronous datagram service has omission/performance failure semantics [9]: it can lose a message or it can fail to deliver a message in a timely manner. More precisely, the *asynchronous datagram service* satisfies the following requirements:

- Only "real" messages are delivered, i.e. when a process p receives a message m from q at some time t, then q has sent m to p at some time $s < t$, and
- A message is delivered at most once, i.e. when a process q receives a message m at time s, then there exists no other time $t \neq s$ such that q receives m at t.

The process management service defines a maximum scheduling delay σ, meaning that a process is likely to react to any trigger event (such as a timer event) within σ time units. If a process p takes more than σ time units to react to a trigger event, it suffers a performance failure. We assume that processes have crash/performance failure semantics. A crashed process can recover. Each process p has access to a local hardware clock H_p. The drift rate of a correct hardware clock is bounded by an a priori given constant ρ. Hardware clocks are not synchronized: the deviation between two correct hardware clocks can be arbitrarily large. For most quartz clocks available in modern computers, the maximum hardware clock drift rate ρ is of the order of 10^{-4} to 10^{-6}. Since ρ is such a small quantity, in what follows we neglect terms of the order of ρ^2 or higher. We assume that hardware clocks have *crash* failure semantics and that a non-crashed process has a correct hardware clock. A hardware clock does not recover from a crash.

We use $\mathcal{P}$ to denote the finite set of all processes and N to denote the number of all processes. A set of processes is a *minority* iff it contains at most $F \triangleq \lfloor \frac{N-1}{2} \rfloor$ processes. A set S of processes is a *majority* iff $|S| > \frac{N}{2}$. A process p is *timely* in an interval $[s,t]$, when p is not crashed and it does not suffer any performance failure in $[s,t]$, that is, p does not miss any deadline in $[s,t]$. Two processes are *connected* in $[s,t]$ if they are timely during $[s,t]$ and each message sent by one process to the other in $[s, t-\delta]$ is received in a timely manner [11].

5 Fail-Aware Datagram

A datagram service provides operations to send and to deliver messages. A synchronous datagram service has, by definition, omission failure semantics: the communication system may lose messages but all delivered messages are timely. More precisely, a *synchronous datagram service* satisfies the following properties:

- Only "real" messages are delivered,
- A message is delivered at most once, and
- Only timely messages are delivered, i.e. when process p receives a message m at time t from q, then q has sent m at some time $s \geq t - \delta$.

The asynchronous datagram service has omission/performance failure semantics. One potential problem with this failure semantics is that slow messages can contain out-of-date information and hence, such messages often need to be detected as being late and rejected by the receiver. We address this problem by defining a *fail-aware datagram service*. This service allows a receiver to detect when the transmission time of a message was greater than an a priori given constant Δ. It is then up to the receiver to handle such an "exceptional" message in a manner that makes sense at its level of abstraction [9]. For example, the receiver can reject all exceptional messages to get omission failure semantics.

To simplify the description of the requirements for the fail-aware datagram service, we assume that each fail-aware datagram message (or fa-message) is unique. In other words, two fa-messages are different even if they are sent by the same process (at two different points in time) and have the same "contents". When the service delivers an fa-message m to a process p, it provides the message m, the identification s of the sender, and an exception indicator e. When this delivery happens at time t, it is denoted by $Deliver_p(m,s,e)$ at t. The event of process p sending a message m to a process q at time t is denoted by $Send_p(m,q)$ at t. By analogy with exception handling terminology [5], we call the delivery of a message with a false exception indicator a *standard* delivery, and the message is then called *standard*. The delivery of a message with a true exception indicator is called an *exceptional* delivery, and the message is accordingly called *exceptional*.

To explain the requirements of a fail-aware datagram service, let us consider that each fa-message m is sent as a message of the underlying asynchronous datagram service (an a-message). Ideally, we would like that an fa-message sent by (1) a timely a-message be delivered as a standard fa-message, and (2) a late a-message be delivered as an exceptional fa-message. The problem is that we cannot calculate the transmission time perfectly and hence, cannot implement exactly this goal. However, it is possible to define a constant $\Delta > \delta$ such that (1) all a-messages with a transmission delay greater than Δ result in an exceptional delivery, (2) all a-messages with a transmission delay smaller than or equal to δ result in a standard delivery, and (3) all a-messages with a transmission delay between δ and Δ result in either a standard or an exceptional delivery.

We will relax condition (2) by allowing that after becoming connected, two processes deliver for a bounded amount of time only exceptional messages. This relaxation allows us to design more efficient protocols with a tight Δ. We thus require a fail-aware datagram service to satisfy the following properties:

- *Only "real" messages are delivered.* When a message m is delivered to process p at t identifying q as the sender, then q has sent m at some time $s < t$: $\forall p, q, m, t$: $Deliver_p(m, q, \_)$ at $t \rightarrow \exists s < t : Send_q(m, p)$ at s.
- *Messages are delivered at most once:* $\forall p, m, s, t$: $Deliver_p(m, \_, \_)$ at $t \wedge Deliver_p(m, \_, \_)$ at $s \rightarrow s = t$.
- *Fail-awareness.* When a process p receives a standard message m at time t from q, then q has sent m at or after time $t - \Delta$: $\forall p, q, m, t$: $Deliver_p(m, q, false)$ at $t \rightarrow \exists s \geq t - \Delta$: $Send_q(m, p)$ at s.
- *Timeliness.* When processes p, q are connected in $[s - d, s + \Delta]$, where d is a protocol specific constant, and q sends a message m to p at time s, then p receives m as a standard message by $s + \Delta$: $\forall p, q, s, m$: $connected(p, q, s - d, s + \Delta) \wedge Send_q(m, p)$ at $s \rightarrow \exists t \leq s + \Delta : Deliver_p(m, q, false)$ at t.

5.1 Protocol Overview

The first two requirements for a fail-aware datagram service are already satisfied by the underlying asynchronous datagram service. We therefore restrict ourselves to describe the implementation of the fail-awareness and the timeliness requirement. All fa-messages are sent as a-messages. The fail-awareness requirement is implemented using an approach similar to that of the probabilistic remote clock reading [6, 10]: a process p computes for each a-message m an upper bound on the transmission time by measuring the duration of an asynchronous datagram round-trip which includes m. Any fa-message sent as an a-message with a calculated upper bound greater than Δ is delivered as exceptional; otherwise it is delivered as standard.

More precisely, each a-message is time stamped with a send time and a receive time. All time stamps are determined by the local hardware clocks. Each process p stores for each process q the receive and send time stamp of the message m_q with the highest send time stamp that p has received from q so far. Process p includes the send and receive time stamp of m_q in every a-message m process p sends to q. When q receives m from p, it calculates an upper bound for the transmission delay of m using the time stamps of m_q and m (see figure 2). In case p has not yet received a message from q, the upper bound for m is set to be infinite. Let $m_q.R$,$m.R$ denote the receive stamps and $m_q.S$,$m.S$ the send stamps of m_q and m, respectively. We assume the existence of a lower bound for the transmission delay of a-messages which we denote δ_{min}. Note that such a bound always exists because the transmission delay of a message is always positive.

The duration between the time p receives message m_q and the time p sends m is at least $(m.S - m_q.R)(1 - \rho)$. The factor $-\rho$ accounts for the possible drift of p's hardware clock. The duration between q sending m_q and q receiving m is at most $(m.R - m_q.S)(1 + \rho)$, where factor $+\rho$ accounts for the drift of q's hardware clock. Because the transmission time of m_q takes at least δ_{min} time units, the transmission delay of m is at most:

$$(\text{m.R-m}_q\text{.S})(1+\rho) - (\text{m.S-m}_q\text{.R})(1-\rho) - \delta_{min}.$$

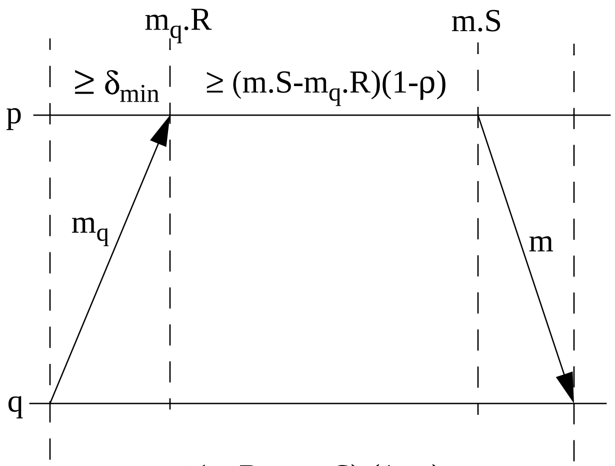

Figure 2: Calculating an upper bound for the transmission delay of message m using m_q.

The timeliness property is enforced by (1) ensuring that any two connected processes p, q receive at least every d time units an a-message from each other to allow p and q to calculate an upper bound after at most d time units after they have become connected, and (2) defining Δ such that whenever a process is able to calculate an upper bound for an a-message received from a connected process, this bound is at most Δ. Each process p has a watchdog which monitors the outgoing messages. When p has not sent a message to a process q for up to $d(1-\rho)-\delta$ clock time units and p is timely, the watchdog will send a "helper" a-message to q, where factor $-\rho$ takes care of the hardware clock drift rate and term δ of the transmission delay of the helper message. A timely process p succeeds therefore that each connected process receives at least every d time units an a-message from p. Because clock synchronization and group membership protocols send messages periodically to all other processes, the fail-aware datagram service has typically not to send helper messages.

The constant Δ has to be defined such that when an a-message m is sent from process p to q and these two processes has been connected for at least d time units when sending m, then the calculated upper bound is at most Δ. The calculated upper bound is maximum when (1) the term $(m.R-m_q.S)(1+\rho)$ is maximum, and (2) the term $(m.S-m_q.R)(1-\rho)$ is minimum with respect to (1). This is the case when

- q's hardware clock drift rate is $+\rho$,
- the transmission delay of m_q and m is δ,
- q has sent m_q exactly $\delta+(d-\delta)=d$ time units before p sending m, and
- p's hardware clock drift rate is $-\rho$.

Therefore, (1) p measures the time between receiving m_q and sending m as $(d-\delta)(1-\rho)$, and (2) q measures the time between sending m_q and receiving m as $(d+\delta)(1+\rho)$. Thus, we define Δ by:

$$\Delta \triangleq (\mathrm{d}+\delta)(1+2\rho)\text{-}(\mathrm{d}\text{-}\delta)(1\text{-}2\rho)\text{-}\delta_{min}.$$

For example, measurements on a network of SUN IPX workstations showed that 99% of the messages sent were delivered within 1.1 ms and all delays were greater than .85ms. Let us assume that a process p ensures that any connected process receives at least every $d=.5s$ a message from p. By defining $\delta \triangleq 1.1ms$ and $\delta_{min} \triangleq$ .85ms and $\rho \triangleq 10^{-5}$, Δ evaluates to $\approx 1.4ms$.

6 Fail-Aware Clock Synchronization

Deterministic internal clock synchronization for synchronous systems [18] requires that

- at any point in time t and for any two correct processes p and q, the deviation between the clocks of p and q be bounded by a constant *maximum deviation* ψ, and
- the drift rate of a clock of a correct process be bounded by a constant *maximum drift rate* $\rho_v \ll 1$.

Deterministic internal clock synchronization cannot be implemented in timed or time-free asynchronous systems due to the presence of unbounded message transmission delays. We propose a new fault-tolerant internal clock synchronization specification, which is implementable in any asynchronous system having local hardware clocks.

6.1 Δ-Stability

A fail-aware service has to provide its standard semantics whenever the system exhibits "synchronous behavior". The exact meaning of a system "exhibiting synchronous behavior" is defined by a *stability predicate* [11, 15]. For fail-aware clock synchronization we define a predicate Δ-*stable* and we will require that whenever a process has been Δ-stable for a sufficiently long time its clock has to be synchronized. The intuition is that the system is Δ-stable whenever a "global observer" could identify a minority set U of processes such that the remaining processes are timely and can communicate with each other in a timely manner. Moreover, the processes in $\mathcal{P}-U$ can detect all messages from processes in U as slow and can therefore reject these messages even without exactly knowing which processes are in U. This idea is generalized to allow a system to be continuously Δ-stable even when a bounded number of processes in $\mathcal{P}-U$ recover or crash during I. There exists weaker and perhaps "more intuitive" predicates like the *majority-stable* predicate of [15] which could be used for fail-aware clock synchronization. Even though we have designed a protocol for majority-stability, we will describe a protocol for Δ-stability to simplify the protocol description.

We say that the system is Δ-*stable* in an interval $I = [s, t]$ iff $s < t - \delta$ and the set of processes can be partitioned into four sets S, C, R, and U such that (1) S is a majority and all processes in S are timely in I, (2) each process in C crashes at some time u in I, is timely in $[s, u)$ and crashed during $[u, t]$, (3) each process in R recovers at some time u in I, is crashed in $[s, u)$, and is timely during $[u, t]$, (4) U contains all other processes, (5) each message sent by a process in U and received by a process in $\mathcal{P} - U$ during I has a transmission delay of more than Δ real-time units, and (6) each message sent between two processes in $\mathcal{P} - U$ at a time $u \in [s, t - \delta]$ has either a transmission delay of at most δ or is dropped because the receiver was crashed at some time in $[u, u + \delta]$. We call the set $\mathcal{P} - U$ a Δ-stable majority. A process p is Δ-*stable* in an interval $[s, t]$, when there exists a Δ-stable majority M in $[s, t]$ such that $p \in M$ and p is timely during $[s, t]$. By requiring that p is timely in $[s, t]$ we ensure that p neither crashes nor recovers during $[s, t]$.

6.2 Requirements

The idea behind a *fail-aware* clock synchronization service is that (1) each process p has a "synchronization" indicator S_p which tells the process if its clock is currently internally synchronized, and (2) indicator S_p is true whenever p has been Δ-stable for at least, say, ι time units. Condition (1) lets clients ensure safety properties which depend on the property that the deviation between their clocks is bounded since clients can detect when their clocks are out of synch. The second condition rules out trivial solutions and allows Δ-stable processes to bound the time it takes to become synchronized despite of a bounded number of crash, performance, and omission failures. We denote by $C_p(t)$ the value of process p's "fail-aware" clock C_p at real-time t. Fail-aware clock synchronization requires that the following conditions be satisfied:

- *bounded deviation:* at any time t and for any two processes p and q which are non-crashed at t,
  $$S_p(t) \wedge S_q(t) \rightarrow |C_p(t) - C_q(t)| \leq \psi.$$
  Constant ψ is called the *maximum deviation.*
- *bounded integration time:* when a process p is Δ-stable in interval $[t, t + \iota]$, then $S_p(t + \iota)$ is true. Constant ι is called the *maximum integration time.*
- *bounded drift:* for any two points in real-time $s < t$ and any process p whose clock has been synchronized between s and t, i.e. $\forall r : s \leq r \leq t : S_p(r)$, the drift of p's clock is bounded by
  $$(t\text{-}s)(1\text{-}\rho_v)\text{-}G \leq C_p(t)\text{-}C_p(s) \leq (t\text{-}s)(1\text{+}\rho_v)\text{+}G,$$
  where the *maximum discontinuity* G accounts for the adjustments of the clocks and the maximum drift rate $\rho_v \ll 1$ for the drift of the hardware clocks (and adjustments).

6.3 Protocol Overview

Our goal is to illustrate that fail-aware clock synchronization is possible in timed asynchronous systems. The intuition behind the sketched protocol is that (1) when none of the clocks are synchronized, a leader is elected and the remaining processes try to synchronize their clocks to the clock of the leader, and (2) synchronized clocks periodically read each others clocks and a process can keep its clock synchronized when it succeeds to read a majority of synchronized clocks within some a priori given error. The synchronization indicator S_p of process p is implemented using a boolean flag and an expiration time. The expiration time ensures that the indicator is switched off before p's clock can drift out of synch even when p suffers performance failures.

The synchronization protocol uses a *fail-aware remote clock reading method.* The goal of this method is (1) to read remote clocks within some *maximum acceptable error* Λ, and (2) to provide a flag which indicates if the remote clock is synchronized. The method is fail-aware because (a) it has an exception indicator and when the indicator is false, the executing process has read the specified remote clock within Λ, and (b) when two processes p and q are connected for at least some d time units, p will *succeed* to read q's clock, i.e. the exception indicator is false. The implementation of this method is straightforward when using a fail-aware datagram service because a process q can send its current clock value in an fa-message m and when m is delivered to the receiver p as a standard message, p knows that the transmission delay of m is at most Δ. An appropriate definition of Λ ensures that all processes which are connected for at least d time units will succeed to read each others clocks. Note that all processes which have been Δ-stable for at least d time units will succeed to read each others clocks.

A clock C_p or a process p is called *synchronized* iff its indicator S_p is true. To *synchronize* a clock C_p or process p means therefore to set its indicator S_p to true. When we want to emphasize that the value of a clock indicator turns from false to true, we talk about *integrating* the clock (to the set of synchronized clocks). Because of the bounded deviation requirement, to synchronize or integrate clocks means to bring them within ψ.

A process p has to resynchronize its clock at least every, say, r time units because otherwise it could drift out of synch with the other clocks. When p does not succeed to resynchronize its clock within r time units, its indicator S_p has to be switched off. Because p can suffer performance failures, p cannot perform this switch itself. The indicator S_p is therefore represented by (1) a boolean flag, and (2) an expiration time which is a hardware clock time-stamp that says how long the contents of the boolean flag is valid. When a client of the fail-aware clock service reads p's clock, it also evaluates S_p: S_p is true iff the flag is true and the validity of the flag has not expired, i.e. the expiration time is greater than the current value of p's hardware clock. Even if performance failures occur during the execution of the synchronization protocol, clients will nevertheless correctly evaluate S_p because in this case the protocol also fails to update the expiration time.

The synchronization protocol is divided into three sub-protocols: integration, join, and maintenance protocol. The cooperation between the three sub-protocols is based on the following idea:

- When the synchronized Δ-stable processes form a majority, then the maintenance protocol will keep these synchronized and the join protocol will integrate the remaining Δ-stable processes.
- When the system is Δ-stable and the synchronized Δ-stable processes form a minority (this can only happen when the system just became Δ-stable), then the maintenance protocol will fail to keep these clocks synchronized. Hence, after a bounded amount of time no clock will be synchronized. After such a loss of synchrony, the integration protocol will succeed to integrate all Δ-stable processes.

The *maintenance protocol* adjusts all synchronized clocks in a round based fashion. Only processes with synchronized clocks participate in the maintenance protocol. It uses the synchronized clocks to ensure that the adjustments of Δ-stable processes take place in a small real-time interval with a maximum size of, say, β time units. A process p can only adjust its clock when it succeeds to read the *synchronized* clocks of a majority of processes. Process p sets its clock to the midpoint of the synchronized clocks it has read successfully. Why this is sufficient to keep p's clock synchronized can be explained as follows: when two processes p and q can keep their clocks synchronized, there exists a synchronized clock C such that p and q have successfully read C, and C ensures that the distance between p's and q's midpoint is bounded by some constant. When a process p does not succeed to read a majority of synchronized clocks, it cannot keep its clock synchronized and has to reset its synchronization indicator S_p.

When the system is non Δ-stable, clocks can nevertheless be synchronized. Hence, when the system becomes Δ-stable some of the Δ-stable processes might be synchronized and others might not be synchronized. In case the synchronized Δ-stable processes form a majority, the maintenance protocol will keep all these clocks synchronized because they succeed to read each others clocks. The join protocol will integrate the remaining Δ-stable processes. When the system becomes Δ-stable and the synchronized Δ-stable processes form a minority, a Δ-stable process p can at most succeed to read the synchronized clocks of a minority of processes. Hence, after a bounded amount of time all clocks will have lost their synchronization. The integration protocol will then succeed to integrate all Δ-stable processes.

The *join protocol* satisfies the property that when the synchronized Δ-stable processes form a majority, it integrates all other Δ-stable processes. Only processes with clocks which are not synchronized participate in the join protocol. The join protocol uses messages sent by the maintenance protocol. When a non synchronized process p succeeds to read the synchronized clocks of a majority of processes (using clock reading messages sent by the maintenance protocol), it adjusts its clock to the *midpoint* of these values and sets its indicator to true.

The goal of the *integration protocol* is to integrate all Δ-stable processes iff none of the clocks are synchronized. The idea is that when the system is Δ-stable, the protocol succeeds to elect a Δ-stable process as leader in a bounded amount of time. This leader l then sends a "start" fa-message to all processes to request that all processes synchronize their clocks to l's clock. When a process receives a standard "start" message, it synchronizes its clock to the leader. All Δ-stable processes succeed in synchronizing their clocks to l's clock because they can read each others clocks successfully. All synchronized clocks are then at most 2Λ apart from each other. Leader l also schedules the next round of the maintenance protocol such that all processes which have synchronized their clocks to l, have to resynchronize their clocks after at most r time units after l requested to become the leader.

To ensure that a leader l is elected only when none of the clocks are synchronized, we require that (1) at least a majority of processes have to support the election of l as the new leader, and (2) all these supporting processes have not participated in any of the three synchronization protocols for at least $r + \beta$ real-time units. Therefore, none of the clocks are synchronized when l is elected because (1) the three synchronization protocols can only synchronize a process when a majority of processes participate in these protocols, and (2) a process has to synchronize its clock at least every r time units and the interleaving between successful adjustments of the same round is at most β.

We assume that there is a total order on process identifiers. Each process which participates in the integration protocol sends an "election" message in a round based fashion. Each process p stores the smallest sender identification L_p of the standard "election" messages it has received in the last two election rounds. All Δ-stable processes implicitly agree on the Δ-stable process with the smallest identification and hence, all Δ-stable processes support the election of the same process. When a process p receives a standard "election" message from L_p and p has not participated in the maintenance / join protocol and has not supported the election of a leader in the last $r + \beta$ time units, it replies with a "support" message to L_p. When a process l receives standard "support" messages from a majority of processes within a certain amount of time after sending an "election" message to all processes, it becomes the leader for a bounded amount of time.

7 Example: Fail-Aware Weak Group Membership

Most previous attempts to specify a primary partition asynchronous group membership service have failed to achieve their goal in being implementable in asynchronous systems [3]. The reason for this impossibility is that a service weaker than the services specified in those attempts is not implementable in asynchronous systems. This service is called the *weak group membership* (WGM) service. Consider a team G of at least 4 processes p_1, p_2, p_3, p_4, and assume that all these processes are in the first group g to be created by the service. The WGM service is required to satisfy the following properties [3]: *(1) If p_1 or p_2 (or both) wish to leave g, then at least one process eventually joins a newly created group g', and no process joins a different group, and (2) for every $i \in$ {1,2} there is at least one execution in which p_i is the only process that wishes to leave g and also the only one to be removed from a newly created group g".*

In this section we show how the WGM service specification can be transformed into a fail-aware service specification that is implementable in timed asynchronous systems. The intuition behind this transformation is that many timed applications have only to make progress when the system exhibits synchronous behavior, and hence, allow requirement (1) to be relaxed. For instance, in the railway example (see section 3) controllers could use a membership service to keep track of the correct sensors. A sensor leaves the current group g of correct sensors when a periodical self-check fails. Consider that the railway crossing application has to provide its standard semantics whenever the system is Δ-stable, and otherwise it is allowed to switch to its safe mode. When a sensor p_i fails, it is sufficient that a new group g', which does not have p_i as a member, be created as soon as the system becomes Δ-stable. In particular, if the system never becomes Δ-stable it is acceptable that p_i be never removed "from the group", i.e. no new group is created from which p_i is removed. We say that the current group of a process q is *out of date* at time t if a process p

leaves at time $s < t - \tau$, where τ is an a priori given constant, and the group q has joined at t does not reflect p's departure. We define for each process p a membership indicator $M_p(t)$ so that when $M_p(t)$ is true, p's current group is up to date (i.e. it is not out of date).

We define a fail-aware WGM service which is implementable in timed asynchronous systems as follows: (1) if p_1 or p_2 (or both) wish to leave g at s or t, respectively, and there exists a process p_j and a time $u > min\{s,t\} + \tau$ so that $M_{p_j}(u)$ is true, then p_j eventually joins a new group g', and no process joins a different group, (2) for every i $\in$ {1,2} there is at least one execution in which p_i is the only process that wishes to leave g and also the only one to be removed from the newly created group g', and (3) when p_j is Δ-stable in $[u - \tau, u]$, then $M_p(u)$ is true. Because of space restrictions, we do not give a protocol for the fail-aware WGM service or a full-fledged fail-aware group membership service specification. The intuition why this problem becomes solvable is that all Δ-stable processes can agree on the processes to be removed. This can be done in a manner similar to the consensus protocol given in [15] which achieves such an agreement under the weaker assumption that processes are majority-stable (instead of Δ-stable).

8 Fail-Awareness

Fail-awareness is a design concept which allows to transform a synchronous service specification into a fail-aware service specification implementable in timed asynchronous systems. We have used this method to transform several synchronous services into fail-aware services. Our experience indicates that a mechanical transformation does not seem to be possible nor desirable because (1) the exact meaning of the "system exhibits synchronous behavior" has to be selected carefully so that the fail-aware service has to provide its standard semantics "most of the time" while still being implementable, and (2) the standard semantics of a fail-aware service has only to be "similar" to the semantics of the synchronous service to accommodate the possibility of performance failures.

By "similar" we mean that the standard semantics implies the synchronous semantics under the condition that all processes and messages would always be timely. We shall explain this weak kind of similarity using the following example. In a synchronous membership service [8] the current group membership v can satisfy the properties: (1) v does not contain any process which crashed or left the group more than τ time units ago, and (2) v contains all processes which have not been crashed for at least τ time units. In a synchronous system condition (2) is equivalent to the condition that v contains all processes which have been showing "synchronous behavior" for at least τ time units. For a fail-aware asynchronous group membership service, we could therefore require that an up-to-date membership v satisfies the properties (1), and (2') v contains all processes which are Δ-stable for at least τ time units.

The exact meaning of "the system exhibits synchronous behavior" is determined by a *stability predicate* such as *stable* [11], Δ-stable, or *majority-stable* [15]. The weaker the stability predicate is, the more often the system exhibits synchronous behavior and the more often a fail-aware service has to provide its standard semantics. However, any fail-aware services requires that the system exhibits some "minimal synchronism" before it can provide its standard semantics. This minimal synchronism depends on the standard semantics and can therefore be different for different services. Furthermore, for most services it seems that the closer one chooses the stability predicate to the minimal synchronism needed, the more complex the protocol to implement the service becomes. Therefore, defining a fail-aware service requires to make a trade-off between having a less complex protocol and a service being able to provide its standard semantics more often.

8.1 Progress Assumptions

The standard semantics of a fail-aware service often implies the original synchronous specification under less stringent conditions. For example, the properties of the fail-aware WGM imply the properties of the WGM under the following assumption (PA): the system always eventually becomes Δ-stable for a "sufficiently long time". We call condition (PA) a *progress assumption* [15]. A progress assumption is introduced to guarantee that a system will exhibit synchronous behavior, that is, they are a way to introduce additional synchronism into the timed asynchronous system model. In particular, progress assumptions can be used to strengthen the properties of fail-aware services without changing the protocol implementing the service. For example, the properties of a fail-aware WGM service are strengthened with the progress assumption (PA) so that they imply the properties of WGM.

Most timed applications do not require progress assumptions because they are only required to to provide its standard semantics when the system exhibits synchronous behavior. We note however that progress assumptions can also be useful to solve time-free problems (which do not have specified deadlines). These problems typically require that a solution be eventually found. Fail-awareness applied to time-free problems means that a solution is found under the condition that the system exhibits synchronous behavior for a sufficiently long time. The reason for this is that a fail-aware service is required to provide its standard semantics whenever the system shows synchronous behavior. Because an appropriate progress assumption guarantees that the system will show synchronous behavior for a sufficiently long time, it is guaranteed that a solution will eventually be found.

9 Conclusion

Many services defined for synchronous systems are not implementable in asynchronous systems. This paper has discussed a concept for modifying synchronous service specifications so that they become implementable in timed asynchronous systems. The main idea is to introduce an indicator which allows a client to detect if the service is currently providing its standard semantics or an exceptional semantics, and to ensure that the modified service provides its standard semantics whenever the system exhibits synchronous behavior, e.g. when the system is Δ-stable. We have shown how the specification of a synchronous datagram service and an internal clock synchronization service can be transformed into a fail-aware service specification. Fail-awareness is applicable to various other synchronous services such as a group membership service and an atomic broadcast service. Moreover, fail-awareness can be applied to time-free problems like the weak group membership problem (see section 7) and failure detectors [16]. The combination of fail-awareness and progress assumptions allows the solution of time-free problems which are otherwise not solvable in asynchronous systems. The usage of fail-aware services to ensure safety properties of timed applications is illustrated by a railway crossing example.

References

[1] H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. R. Renaming in an asynchronous environment. *Journal of the ACM*, 37(3):524–548, 1990.

[2] C. B. and C. Dwork. Randomization in byzantine agreement. *Advances in Computer Research*, 5:443–497, 1989.

[3] T. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost. Impossibility of group membership in asynchronous systems. Technical Report 95-1533, Computer Science Department, Cornell University, Ithaca, 1995.

[4] 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.

[5] F. Cristian. Correct and robust programs. *IEEE Transactions on Software Engineering*, SE-10(2):163–174, Mar 1984.

[6] F. Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3:146–158, 1989.

[7] F. Cristian. Synchronous atomic broadcast for redundant broadcast channels. *The Journal of Real Time Systems*, 2:195–212, 1990.

[8] F. Cristian. Reaching agreement on processor-group membership in synchronous distributed systems. *Distributed Computing*, 4:175–187, 1991.

[9] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of ACM*, 34(2):56–78, Feb 1991.

[10] F. Cristian and C. Fetzer. Fault-tolerant internal clock synchronization. In *Proceedings of the Thirteenth Symposium on Reliable Distributed Systems*, Dana Point, Ca., Oct 1994.

[11] F. Cristian and F. Schmuck. Agreeing on processor-group membership in aynchronous distributed systems. Technical Report CSE95-428, Dept of Computer Science and Engineering, University of California, San Diego, La Jolla, CA, 1995.

[12] 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.

[13] D. Dolev, N. Lynch, S. Pinter, E. Stark, and W. Weihl. Reaching approximate agreement in the presence of faults. *Journal of the ACM*, 33(3):499–516, 1986.

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

[15] 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.

[16] C. Fetzer and F. Cristian. Fail-aware failure detectors. Technical Report CSE96-475, UCSD, 1996. Available via anonymous ftp at cs.ucsd.edu as /pub/team/failAwareFD.ps.Z.

[17] 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.

[18] L. Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(1):52–78, Jan 1985.

[19] J. Lundelius-Welch and N. Lynch. A new fault-tolerant algorithm for clock synchronization. *Information and Computation*, 77(1):1–36, 1988.