Derivation of Fail-Aware Membership Service Specifications

Christof Fetzer, Flaviu Cristian

Department of Computer Science & Engineering
University of California, San Diego
La Jolla, CA 92093–0114
http://www-cse.ucsd.edu/users/cfetzer

Abstract. We derive the specification of a primary partition and a partitionable fail-aware node membership service in a top-down fashion. The derived specifications are fail-aware in the sense that each client of a membership server can learn if the server currently provides its standard semantics or an exception semantics because too many failures have occurred. We first propose the specification of an *ideal membership* service and then transform this ideal specification step by step to derive the two fail-aware specifications that are implementable in timed asynchronous systems. In each step we address an implementation problem or a change in the system/failure model.

1 Introduction

Roughly speaking, a membership service is supposed to provide to each process p the names of all the processes that are in the same network partition as p and all processes in p's network are supposed to agree on what processes are in their partition. It is actually quite difficult to specify precisely a membership service that makes sense in practice and that is still implementable in existing distributed systems like a network of workstation [1]. In particular, [2] showed that no useful membership service is implementable in *time-free* asynchronous systems [2, 14], i.e. distributed systems without any notion of time and without any failure detectors. The explanation for this impossibility result is that processes cannot decide if another process is crashed or just slow and such a decision is necessary to achieve an agreement on the current membership.

One possible specification is to require that the set of process names the membership service maintains in a network partition be the set of all non crashed processes that are in this partition. However, it is not always possible to decide if a remote process is crashed or just slow. In practice one actually does not necessarily want to know which processes are crashed and which are not. Instead one is typically interested in knowing which processes can communicate with each other in a timely manner, i.e. one wants to exclude processes from the membership of a process p that are slow or that cannot communicate with p in a timely fashion. The idea behind our derivation of a membership service specification is actually to identify maximum sets of processes in the system that behave like a

synchronous system, i.e. the communication delays and process scheduling delays experienced within each set can be bounded by some given constants. We will introduce the notion of a *stable partition* to refer to such a subset of processes with synchronous behavior. Since each such stable partition behaves like a synchronous system, the processes in a stable partition can actually achieve an agreement on what processes are in their stable partition.

In our work we use instead of the time-free [14], the *timed* asynchronous system model [5] which has a notion of performance failures and assumes that each process has access to an *unsynchronized* local hardware clock with a bounded drift rate. A performance failure occurs when the delay experienced by a process or a message is greater than some a priori given constant. The system model is asynchronous in the sense that there is no upper bound on the performance failure frequency and hence, there actually does not exist any upper bound on process and messages delays. As in time-free systems, in timed asynchronous systems processes cannot necessarily decide if another process is crashed or just slow. However, unlike for time-free systems [2], it is possible to specify and implement practically relevant membership services in timed asynchronous systems. The fundamental reason why reasonable membership services can be implemented in timed systems is that processes can distinguish between "timely" and "slow" processes and messages using the local hardware clocks.

The specification of a membership services is not trivial [1]. In particular, the specifications are often quite complex and sometimes the intuition of a specification is hard to capture. We address this problem by giving a top down derivation of two membership services for timed asynchronous systems: a primary partition and a partitionable membership service. We start our derivation by proposing the specification of an *ideal membership* service and then transform this ideal specification step by step to derive two "fail-aware" specifications. In each step we address an implementation problem or a change in the system/failure model. The derived specifications are implementable in timed asynchronous systems.

To reduce the complexity of the specification, we concentrate on a node membership service instead of deriving a specification for a process membership service: a node membership service keeps track of the computer nodes in a system while a process membership service keeps track of all processes in the system. A node membership service can be extended to a complete process membership service along the lines described in [3]. We will assume that each computer node is represented by exactly one membership server process: the node membership service keeps track of these (server) processes.

2 Timed Systems and Partitions

In timed asynchronous systems message transmission delays and process scheduling delays are associated with time-out delays (δ and σ). These time-out delays allow us to introduce performance failures, which give the terms "slow" and "timely" message / process a precise meaning. The timed asynchronous system model assumes the following failure model: processes have crash/performance

failure semantics and messages have omission/performance failure semantics. Unlike in synchronous systems where the maximum frequency of performance, omission and crash failures is a priori bounded, in timed asynchronous systems there exist no such a priori given upper bound. The timed asynchronous system model is an accurate description of networks of workstations [5].

Many distributed systems can partition into disjoint subsets of processes due to network failures or excessive message and process performance failures. Each such subset is informally referred to as a *communication partition*. Note that a (communication) partition is not a partition in the mathematical sense. However, such a partition can be viewed as one component of a mathematical partition of the set of processes. A partition SP is called *stable* if the failure frequency experienced by the processes and messages in SP is below a given threshold [13]. Processes of a stable partition are themselves called *stable*. At the membership level, communication partitions are abstracted as *logical partitions*, that is, "well-behaved" partitions that could be viewed as synchronous subsystems in the sense that a membership service provides in each logical partition properties similar to those of a synchronous membership service.

The two main goals of the partitionable membership service that we derive are (1) to provide the processes of a stable partition SP with an approximation of the processes in SP, and (2) to guarantee that these processes agree on their individual approximations of SP. The specification that we derive will require that the partitionable membership service creates for each stable partition SP a *logical partition* and each process in SP periodically updates its *member-set*, i.e. its approximation of the set of stable processes in SP. The primary partition service ensures that at any point in time there exists at most one logical partition.

We call the services *fail-aware* because a process p and its clients know if p's current member-set is "up-to-date", i.e. (1) p agrees with the other processes in its logical partition on the membership, (2) p's local membership information contains all processes in p's stable partition, and (3) it does not contain any crashed process and any process from another stable partition. Each stable process is required to have an up-to-date member-set.

3 Related Work

The membership service was first specified for synchronous systems in [3]. It has also been specified and implemented for asynchronous systems by various authors, e.g. see [19, 18, 15]. In [2] the authors show that it is impossible to derive a useful specification for a membership service that is implementable in time-free asynchronous systems without introducing extensions to the model like failure detectors. The reason for that impossibility result is that a subproblem common to any useful membership service is impossible to solve in time-free systems. The authors of [1] discovered multiple problems in several specifications of membership services for time-free systems [14]. The reason for some of these problems, for example the possibility of having "capricious" membership changes is that time-free systems have no notion of performance failures. By contrast,

in a timed asynchronous system we can specify that a membership service is only allowed to remove "slow" or crashed processes. In time-free systems it is not possible to differentiate between a "slow" and a "timely" process. Therefore, it is difficult to specify a membership service that reflects the behavior of the processes during an execution. For example, Neiger defines a membership service that is implementable in time-free systems [20]. However, there are only weak restrictions on what processes can be removed from the membership.

For timed asynchronous systems, [16] specified two membership services and showed how they can be implemented. Cristian and Schmuck specified in [6] a suite of five membership protocols for timed asynchronous systems, described five protocols that implement the specifications and proved their correctness. All five protocols are timer driven. The specifications we derive in this paper are for clock driven protocols, i.e. all processes in a stable partition agree on a common time base and change their member-set at the same clock times. The goal of this paper is to derive a specification of a membership service for timed asynchronous systems that is as strong and as similar as possible to that of clock driven protocols for synchronous systems. The derived specification is in certain respects even stronger than some synchronous membership specifications, e.g. that of [3], because we will require that processes agree at any point in clock time on the current membership while [3] does only require that processes agree on the order in which they remove or include processes. We show in [11] how the fail-aware specifications that we derive in this paper can efficiently be implemented in timed asynchronous systems.

We introduced in [9] the notion of *fail-awareness* as a general method for extending properties of a fault-tolerant synchronous service by an *exception indicator* so that the new, extended service becomes implementable in timed asynchronous systems. The idea is that each server uses its indicator to tell its clients whether the server currently provides its standard synchronous semantics or if it provides an exception semantics as a consequence of the occurrence of "too many performance failures". We showed in [12] how to extend fail-awareness for partitionable systems: (1) a service has to create for each stable partition a "logical partition" that is an approximation of the partition, and (2) the indicator of a server does not only show if the server provides its standard semantics, it also shows the logical partition of the server.

4 Derivation of Specification

To derive the specification of a primary partition and a partitionable fail-aware membership service, we first propose the specification of an *ideal node membership* service. Since we assume that there is exactly one membership server process per computer node, when we talk about processes, we mean membership server processes. Since a non-crashed node always runs a node membership server, it is sufficient that the servers actually keep track of the other node membership server processes.

4.1 An Ideal Membership Service

Typically, clients of a membership service use the information provided by the service to perform some form of load distribution or to achieve some distributed coordination like managing the availability of distributed services [4]. A membership service should therefore provide an accurate approximation of the set of nodes which are currently stable, i.e. nodes with a bounded performance failure frequency and that can communicate with each other in a timely manner. Each membership server maintains a set (we will call this set the *member-set*) that contains the ids of the node membership server processes that it believes to be in the same network partition.

Let us first assume that the underlying system is completely synchronous, i.e. the only failures that can occur are crash failures: (1) processes are either *crashed* or *non-crashed* and all non-crashed processes are timely, and (2) the message transmissions between two non-crashed processes are failure free, i.e. all messages sent are delivered within some δ time units.

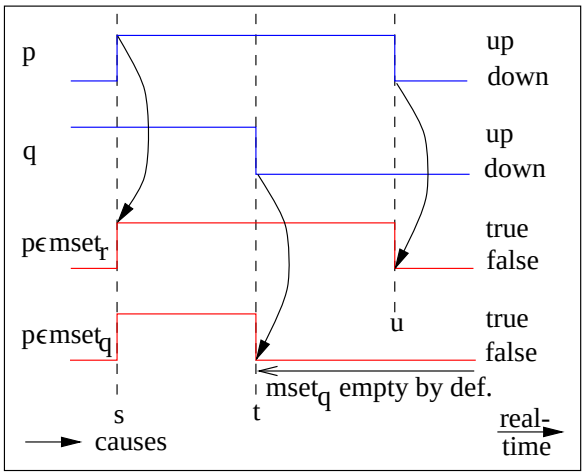

Fig. 1. At any time t an ideal membership service provides each process r that is up at t with a member-set that consists of the set of processes that are up at t. The specification defines the member-set of an any process completely by requiring that the member-set of a process is empty whenever the process is not up.

In this system model all non-crashed processes are by definition stable. An *ideal* instantaneous membership service provides at any real-time t all non-crashed process with the set of all non-crashed processes at t. Let us call a crashed process "down" and a non-crashed process "up" (see Section 4.2 for a precise definition of "up"). The approximation of the set of up processes provided by a membership server is called a *member-set*.

We can specify such an ideal instantaneous membership service by requiring

that for any process p that is up at real-time t, p's member-set be the set of up processes at t (see Figure 1). A crashed process cannot update its member-set and we therefore assume that the member-set of a down process is empty to make sure that the member-set of any process is always completely defined. In other words, we assume that when a process crashes its member-set becomes empty by definition.

We use a notation similar to that suggested in [17] to write the requirements. Let $\mathcal{P}$ denote the set of all processes, RT the set of all real-time values, $up_p(t)$ a predicate that is true iff p is up at real-time t, and $mset_p(t)$ process p's member-set at time t. Formally, we define the requirement (IM = ideal membership) of the ideal membership service as follows:

(IM) $\forall p,r \in \mathcal{P},\ \forall t \in RT:$

$$\begin{array}{ll} \wedge & up_r(t) \Rightarrow (p \in mset_r(t) \iff up_p(t)) \\ \wedge\neg & up_r(t) \Rightarrow mset_r(t) = \{\}. \end{array}$$

4.2 Detection Delay

Even in synchronous systems an ideal membership service (IM) is not implementable: since processes must communicate by exchanging messages and the message delay is always greater than zero, the detection of the crash or recovery of a process takes at least as long as the minimum time to transmit a message. In our first transformation of (IM) we introduce a delay for the detection of crash failures and recoveries of a process: it can take an up process r up to, say, d time units before it detects the crash or recovery of a process p. We call constant d the *detection time* and it has to be defined by any implementation of the refined membership service specification.

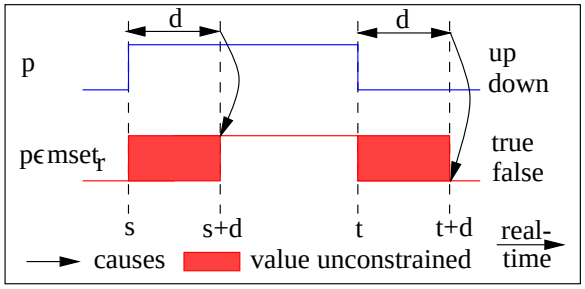

Fig. 2. An up process r can take up to d time units to detect the crash or recovery of another process p. For d time units after p has crashed or recovered, requirement (D) does not constrain the value of "$p \in mset_r$".

Our first transformation of (IM) results in the specification (D = detection delay) that requires for any process r and any time t (see Figure 2),

- if r is up at t, r's member-set at t contains any process p that has been up for at least d time units,

– if r is up at t, r's member-set at t does not contain any process p that has not been up for at least d time units, and
– if r is not up at t, r's member-set is empty.

Formally, we define the detection delay condition (D) as follows:

(D) $\forall p,r \in \mathcal{P},\ \forall t \in RT:$

$$\begin{array}{lll} \wedge & up_r(t) \Rightarrow & \wedge\ (\forall s \in [t\text{-}d,t]:\ up_p(s)) \Rightarrow\ p \in mset_r(t) \\ & & \wedge\ (\forall s \in [t\text{-}d,t]: \neg\, up_p(s)) \Rightarrow\ p \notin mset_r(t) \\ \wedge \neg & up_r(t) \Rightarrow & mset_r(t) = \{\}. \end{array}$$

An implication of the above specification (D) is that when a process p transitions between modes down and up with a frequency greater than $1/d$, p may or may not be in the member-set of the up processes (see Figure 3). We assume that a crashed process stays down for at least d time units. In this way, the crash of a process p always results in p's removal from the member-set of all up processes. Note that this assumption is reasonable because the restart of a process takes a certain amount of time (which is typically already longer than d and otherwise, can be extended to at least d). On the other hand, one cannot exclude that a process p that recovers does not crash again within d time units. In this case, the up processes do not necessarily include p in their member-sets because they might not detect that p has recovered before p crashes again.

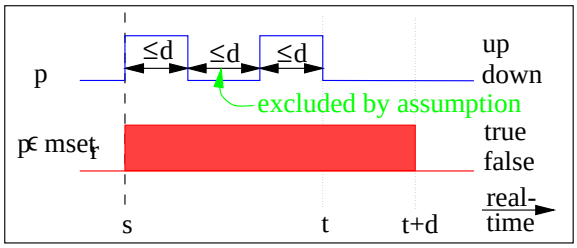

Fig. 3. Since (D) does not constrain if p is in or out of a member-set for d time units after p has crashed or recovered, an up process r may not include a process p in its member-set when p is up for less than d time units. Each crash of p is actually detected because we assume that it takes a process more than d time units to recover from a crash.

In the next subsections we address several problems of specification (D) (agreement, oscillations, and initialization) and solve these problems by extending (D) by two additional requirements (A) and (S).

Agreement Let p and q be two processes that both recover in interval $[s, s+d]$ and that both crash in interval $[t, t+d]$ and let r and o be two processes that are up. If a process recovers or crashes, an up process can take up to d time units to update its member-set. Hence, r and o do not necessarily agree if p or q first

recovered/crashed. Reaching agreement on the order of removals and inclusions of processes is essential for some clients such as availability managers [4].

We address this problem by extending specification (D) by another requirement (A = agreement) that enforces that all up processes agree on the order in which they remove or add processes to their member-sets. Formally, we require that at any time t, all processes that are up at t have the same member-set at t. Since the member-set of any process that is not up has to be empty, two processes also agree on the member-set when both are not up. We can therefore express the agreement condition (A) as follows:

(A) $\forall r,q \in \mathcal{P}$, $\forall t \in RT$: $up_r(t) = up_q(t)$
$\Rightarrow mset_r(t) = mset_q(t)$.

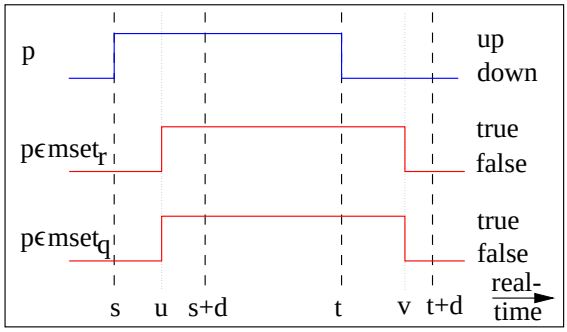

Fig. 4. Condition (D) requires that the any two up processes r and q detect the transitions of any process p within d time units. Condition (A) requires that r and q change their member-sets at the same time.

The specification (D,A) consisting of conditions (D) and (A) requires that all up processes detect the crash or recovery of a process within d time units and that the resulting membership changes be performed at the same time at all membership servers (see Figure 4). Note that even though the processes have to agree on the order of removals and inclusions of processes in their member-sets, this order is not necessarily the real order in which the processes actually have crashed or recovered. However, the processes can at most disagree with the real order of two events when these events occurred within d time units of each other.

Oscillations Each change of the member-set can increases the workload of the clients of the membership service. For example, a member-set change can require an availability manager to start some services that were running on a crashed node [4]. Hence, one wants to avoid capricious membership changes to avoid unnecessary work. The above specification (D,A) allows the up processes to capriciously include and remove a crashed or recovered process p for up to d time units after the process crashed or recovered, respectively (see Figure 5).

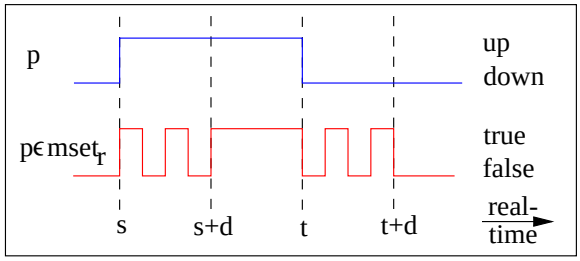

Fig. 5. Specification (D,A) allows that a process p that has recovered or has crashed can be capriciously removed from and included in the membership for up to d time units.

Ideally, we would like to strengthen the specification (D,A) so that each transition of a process q results in exactly one membership state change in the sense that (1) when q transitions to up at real-time s, this results in exactly one inclusion of q in the membership before $s + d$, and (2) when q crashes at t, this results in exactly one removal of q before $t + d$. However, a membership service can only achieve exactly one membership change per transition of q when the frequency of q's transitions is bounded. Hence, we can only require that there be at most one membership change per process transition. We achieve this by requiring that there exist a constant nc ("no change") such that when a process q is removed from (or included in) the membership at some time t, process q stays out (or in) the membership for at least nc time units. We require that nc is more than $\frac{1}{2}$ of the detection time d, i.e.:

$$\exists f > \tfrac{1}{2} : nc \geq f\ d.$$

This implies that $nc > \frac{d}{2}$ for $d > 0$ and removes the possibility of oscillations (see Figure 6) since any capricous membership change takes $2nc > d$ time units. Note that for $d = 0$ specification (D, A) does not allow capricous membership changes (even though nc might be only $\frac{d}{2}$) because the membership has to change instantaneously whenever a process crashes or recovers – leaving no time for capricous membership changes.

We extend the specification (D,A) by the following stability condition: for any process p that is up at some time t and any process q, there exists an interval $[s, s + nc]$ such that t is in that interval and for any time u in that interval, (A) if q was in the member-set of p at t, then q has to be in p's member-set at u unless p is not up at u, and (B) if q was *not* in the member-set of p at t, then q must not be in p's member-set at u (unless p is not up at u but in this case (D) requires the member-set of p to be empty),

(S) $\forall p,q{\in}\mathcal{P},\ \forall t{\in}RT{:}\ up_p(t)$
$\Rightarrow\ \exists s{:}\ s{\leq}t{\leq}s{+}nc,\ \forall u{\in}[s,s{+}nc]$
$\wedge\ q \in mset_p(t) \wedge up_p(u) \Rightarrow q \in mset_p(u)$
$\wedge\ q \notin mset_p(t) \wedge up_p(u) \Rightarrow q \notin mset_p(u).$

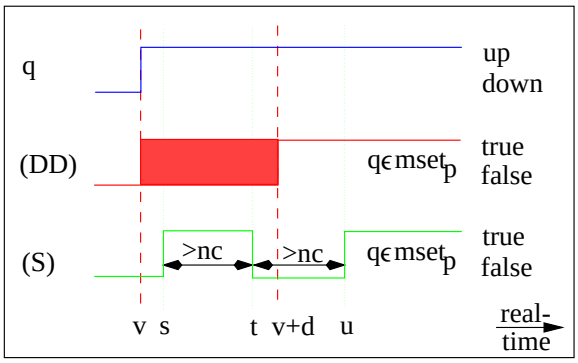

Fig. 6. For $nc > \frac{d}{2}$ requirement (S) guarantees that for each transition of a process q there is at most one transition in the membership because any capricious change requires $2nc > d$ time units.

Comparison of Specifications The ideal membership specification (IM) can be viewed as a special case of the relaxed specification (D,A,S). In particular, (IM) obviously implies the detection delay condition (D) for any detection time $d \geq 0$:

$$IM \Rightarrow \forall d \geq 0 : D.$$

Condition (IM) implies that for any time t and any processes r and q that are up at t, the member-sets $mset_r(t)$ and $mset_q(t)$ contain exactly all processes that are up at time t, i.e. processes r and q implicitly agree at any time on the current membership: $mset_r(t) = mset_q(t)$. Therefore, (IM) also implies the agreement condition (A).

On one hand, the ideal membership specification requires that a crash of a process results in its instantaneous removal and a recovery of a process in its instantaneous inclusion into the membership. On the other hand, the stability condition (S) requires that a process has to stay for at least $nc \geq \frac{d}{2}$ time units in the membership before it is removed from the membership. Hence, if a process can crash in less than nc time units after it has recovered, (IM) does not imply (S). However, (IM) implies the stability condition (S) for $d = 0$ because in this case nc can be set to zero:

$$IM \wedge d = 0 \Rightarrow \forall f > \tfrac{1}{2}, \exists nc \geq fd : S.$$

Note that if an implementation were satisfying specification (D,A,S) for a zero detection time, i.e. $d = 0$, this implementation would also satisfy condition (IM) since

$$D \wedge d = 0 \Rightarrow IM.$$

In summary, the relationship between the two specifications (IM) and (D,A,S) can be expressed as follows:

$$IM \Leftrightarrow \exists d : d = 0 \wedge D \wedge A \wedge S.$$

In other words, the ideal membership specification (IM) can be seen as a special case of specification (D,A,S) that requires the detection time to be zero.

Initialization The next problem we address is that a process p that recovers from a crash, cannot have instantaneously a member-set that contains exactly all up processes. We have to allow that p can take up to, say, x time units before its member-set is up to date (see Figure 7). Moreover, p should not be included in the membership before its member-set is up to date.

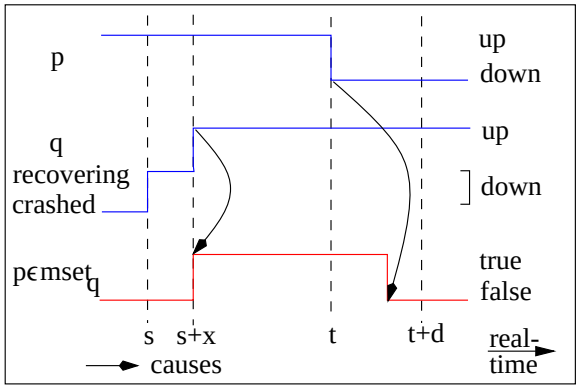

Fig. 7. A recovering process needs up to x time units to update its member-set. Condition (D) requires the member-set of a recovering process q to be empty, i.e. even when some other process p is up for more than d time units while q recovers, q has to set is member-set to empty.

To solve this problem, we assume that a process is in one of the following three modes: *crashed*, *recovering*, and *up* (see Figure 8). A process p that recovers from a crash, transitions to mode "recovering". Process p has to update its member-set and then to switch to mode "up" within x time units (unless p crashes within these x time units). When a process p is in mode "crashed" or "recovering", we say that p is *down*.

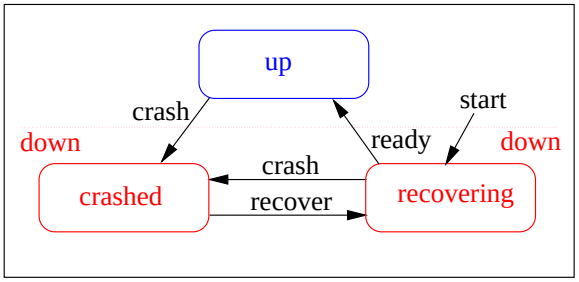

Fig. 8. Modes and mode transitions of a process.

The introduction of a recovering mode solves the above initialization problem in the following way. When process p crashes, it is removed from the membership within d time units. Since we assume that a process is crashed for at least d time units, when p recovers from its crash, it is already removed from the membership. As long as p is in mode "recovering" it just sets its member-set to the empty set as required by (D) and p has up to x time units to update its member-set. Before p can be included in the membership again, condition (D) also requires that p has to transition to mode "up" first. In other words, p can only be included in the membership after it has updated its member-set and switched to mode up.

4.3 Internal Time Base

A major implementation problem with the above specification (D,A,S) is that even in synchronous systems one cannot enforce that the processes change their member-sets at the same time [7]. The best one can guarantee is that two processes change their member-sets within, say, β time units of each other (see Figure 9). Thus, we relax the requirement (A) by replacing real-time by "clock time". We assume the existence of an internal time base provided by an internal clock synchronization service and require that all up processes change their member-sets at identical clock times instead of identical real-times. In other terms, we use the internal time base to guarantee that processes change their member-set within some β real-time units of each other. While for synchronous systems the internal time base is based on deterministic internal clock synchronization, for asynchronous systems we base it on fail-aware internal clock synchronization (specification given below) [9].

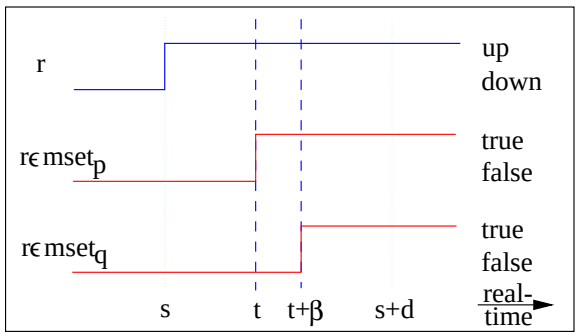

Fig. 9. The best temporal agreement two up processes can achieve is β: their member-set changes can be up to β apart from each other.

Each process p has a clock C_p. We denote the set of clock time values by CT and represent process p's clock by a function $C_p : RT \rightarrow CT$. Internal clock synchronization is specified by two requirements: the bounded deviation (BD) and bounded drift rate (BR) requirement. At any time t, the deviation between

the clocks of two up processes has to be bounded by an a priori given constant Ψ:

(BD) $\forall p,q \in \mathcal{P},\ \forall t \in RT:\ up_p(t) = up_p(t) \neq false$
$\Rightarrow\ |\ C_p(t) - C_q(t)\ | \leq \Psi.$

The drift from real-time of the clock of an up process is at most ρ, where $\rho \ll 1$ is an a priori given constant:

(BR) $\forall p \in \mathcal{P},\ \forall s,t \in RT:\ s < t \wedge up_p(s) = up_q(t) \neq false$
$\Rightarrow (t\text{-}s)(1\text{-}\rho) \leq C_p(t) - C_p(s) \leq (t\text{-}s)(1+\rho).$

Requirement (BR) implies that the time base is strictly monotonic increasing, i.e.

$\forall p \in \mathcal{P},\ \forall s,t \in RT:\ s < t \wedge up_p(s) = up_p(t) \neq false$
$\Rightarrow C_p(s) < C_p(t).$

This can be achieved by using continuous clock amortization in our internal clock synchronization protocol [21]. In what follows, we use S, T, U, V to denote clock times while we use s, t, u, v to denote real-times.

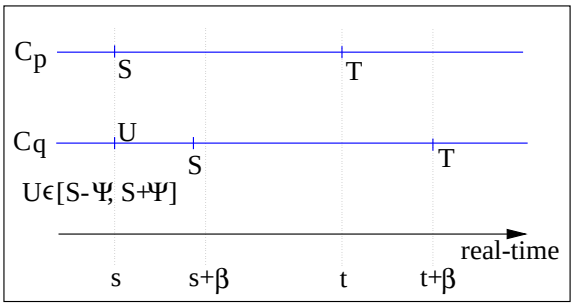

Fig. 10. The specification of the internal time base guarantees that two processes show the same value S within $\beta = (1+\rho)\Psi$.

Using an internal time base, two processes p and q can achieve a temporal agreement of $\beta \stackrel{\Delta}{=} (1+\rho)\Psi$ (see Figure 10): when two processes p and q change their member-set at clock time S, these changes are at most β real-time units apart because (1) for any time s such that $C_p(s) = S$, (BD) implies that $C_q(s) \in [S \Leftrightarrow \Psi, S + \Psi]$, and (2) since the drift rate of C_q is at least $\Leftrightarrow \rho$, C_q shows S within $\Psi(1+\rho)$ real-time units.

4.4 Synchronous Membership Service

We can use the concept of an internal time base to transform requirement (A) so that it becomes implementable in completely synchronous systems, i.e. systems in which processes can crash but neither processes nor messages suffer performance failures. This transformation replaces real-time by clock time (see Figure 11).

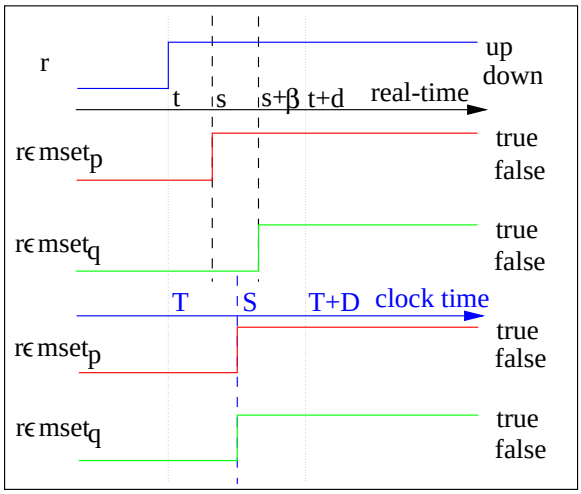

Fig. 11. We make the specification implementable by using clock time instead of real-time in (A): processes change their member-set at the same clock time instead of real-time.

The specification of a synchronous membership service consists of the two conditions (D,S) and the (SA = synchronous agreement) condition that is defined by:

(SA) $\forall s,t \in RT,\ \forall p,q \in \mathcal{P}:$
$\quad \wedge\ C_p(s) = C_q(t)$
$\quad \wedge\ up_p(s) = up_q(t)$
$\qquad \Rightarrow\ mset_p(s) = mset_q(t).$

This specification (D,SA,S) is implementable in completely synchronous systems: the protocol proposed in [11] guarantees (D,SA,S) as long as the underlying system is completely synchronous. The specification implies that all up processes agree on the order in which they remove and include processes in their member-sets and that two up processes change their member-set within β real-time units of each other. Figure 12 shows a possible behavior of a service that satisfies (D,SA,S).

Comparison of Specifications We want to compare conditions (SA) and (A). If clocks were perfectly synchronized with real-time, the following condition (PC = perfect clock) would hold:

$$\textbf{(PC)}\ \forall p : C_p(t) = t.$$

Condition (PC) implies the bounded deviation (BD) and bounded drift condition (BR) for any maximum deviation $\Psi \geq 0$ and any maximum drift rate $\rho \geq 0$.

The idea behind specification (D,A,S) is that processes have perfectly synchronized clocks that allow processes to change their member-sets at the same

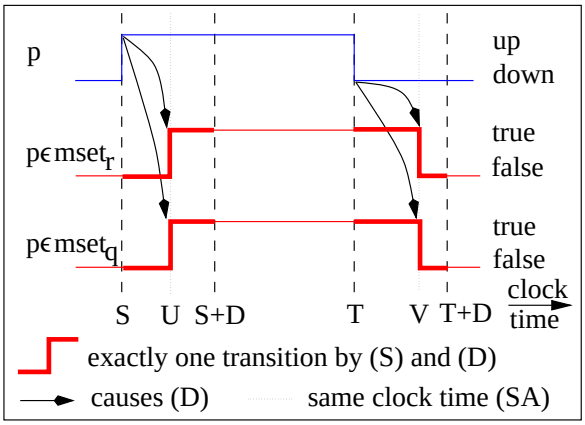

Fig. 12. A synchronous membership service guarantees that all up processes see the same membership changes at the same time and that up processes remove a crashed or include a recovered process within $D \triangleq d(1+\rho)$ clock time units.

time. The relation between the the two specifications (D,A,S) and (D,SA,S) can therefore be expressed as follows:

$$D \wedge A \wedge S \Leftrightarrow \exists C : PC \wedge D \wedge SA \wedge S.$$

Specification (D,A,S) can be seen as a special case of specification (D,AS,S) that requires each process to have a perfect clock.

4.5 Asynchronous Systems

An implementation of the above synchronous specification (D,SA,S) requires that up processes be timely and that they can communicate with each other in a timely manner. For example, the detection delay condition (D) requires a timely communication between processes to make sure that a crash of a process is detected within d time units while no up process is removed from the membership. In asynchronous systems however it cannot be assumed that up processes are always timely or can always communicate with each other in a timely manner. We have to relax the specification so that it becomes implementable in *timed* asynchronous systems [5], i.e. systems in which (1) all processes have a local hardware clock with a bounded drift rate, and (2) message transmissions and process scheduling delays are associated with time-out delays to define performance failures. We transform the synchronous specification (D,SA,S) into a *fail-aware* asynchronous specification which becomes implementable in timed asynchronous systems.

Fail-awareness [9] is a general method for transforming a synchronous service specification into a specification that is implementable in a timed asynchronous system (see [9]). We apply this method to transform the specification (D,SA,S) into a fail-aware specification (AD,AA,AS,AT): instead of requiring that the specification be satisfied by all *up* processes, we will require that (1) it be satisfied

by each process p that has an indicator M_p that is true, and (2) each "stable" process must have an indicator that is true. A "stable" process is up, meets all its deadlines, and can communicate with "some set of processes" in a timely manner. Note that a crashed process is never stable. A process that is neither stable nor crashed is called *unstable*.

We assume that the implementation defines a predicate $stablePartition(Q, s, t)$ that evaluates to true if and only if the set of processes Q forms a stable partition during interval $[s, t]$. Formally, a process p is *stable* in some given interval $[s, t]$ if and only if p is in some stable partition Q during $[s, t]$:

$$p \text{ stable in } [s,t] \Leftrightarrow \exists Q : p \in Q \wedge stablePartition(Q, s, t).$$

We assume that whenever a process p is stable in some interval $[s, t]$, then p has also to be up in $[s, t]$. In other words, whenever p is not up, p cannot be stable. This takes care of the general idea that a crashed process cannot be stable. In the literature there exists different examples of *stability predicates* [6, 8, 9]. For example, Δ-*stable* [9] defines that a set SP of processes form a stable partition iff

- SP contains a majority of processes, and during I
- all processes in SP are timely,
- they can communicate with each other in a timely manner, and
- all messages sent to processes in SP from processes outside of SP are detectably late, i.e. can be detected by the processes in SP as being sent by processes outside of SP because their transmission delay is greater than some $\Delta > \delta$ (see [10]).

Our derivation of the specification of a fail-aware membership service is independent of the stability predicate used by an implementation of the service. We designed and implemented a membership protocol for predicate Δ-*stable* [11].

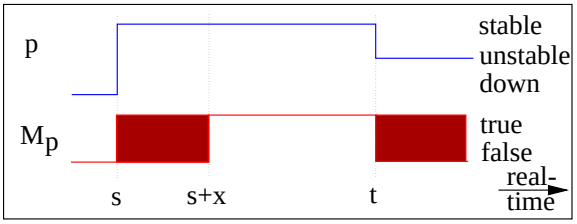

Fig. 13. The indicator of a crashed process is by definition false. A stable process has to set its indicator to true within x time units. An unstable process may have a true or a false indicator.

Synchronous internal clock synchronization as defined by the specification (BD,BR) is not implementable in timed asynchronous systems. However, the specification can be transformed into a fail-aware specification (FD,FR) that is implementable in timed asynchronous systems [9]. The idea is that the clock of

a process p has only to be synchronized at some time t when p's indicator $M_p(t)$ is true at t

(FD) $\forall p,q \in \mathcal{P},\ \forall t \in RT:\ M_p(t) = M_q(t) \neq false$
$\Rightarrow\ |\ C_p(t) - C_q(t)\ | \leq \Psi.$

The drift between real-time and the clock of a process with an up-to-date indicator is at most ρ, where $\rho \ll 1$ is an a priori given constant:

(FR) $\forall p \in \mathcal{P},\ \forall s,t \in RT:\ s < t \wedge M_p(s) = M_p(t) \neq false$
$\Rightarrow\ (t\text{-}s)(1\text{-}\rho) \leq C_p(t) - C_p(s) \leq (t\text{-}s)(1+\rho).$

The specification of a fail-aware primary partition membership service requires that

- each process p maintains an indicator M_p showing if p's member-set is currently up-to-date (M_p is true) or out-of-date (M_p is false),
- the indicator of any process that is stable for at least x time units is true and the indicator of a process that is not up is assumed to be false (see Figure 13),
- any processes that has a true indicator includes in its member-set all processes that have a true indicator for at least d time units and its member-set does not contain any process that has a false indicator for d or more time units,
- two processes that have a true indicator at clock time T, have the same member-set at T, and
- when a process p includes a process q in its member-set, then p keeps q in its member-set for at least nc time units (unless p cannot keep its member-set up-to-date), and when p removes q from its member-set, then p keeps q out of its member-set for at least nc time units.

Formally, we can transform specification (D, SA, S) into a new specification (AD, AA, AS, AT) be replacing predicate up with an indicator M and adding timeliness condition (AT). The specification of a fail-aware primary partition membership service (AD,AA,AS,AT) requires the following conditions to be satisfied:

(AD) $\forall p,r \in \mathcal{P},\ \forall t \in RT:$

$\wedge\quad M_r(t) \Rightarrow \wedge\ (\forall s \in [t\text{-}d,t]:\ M_p(s)) \Rightarrow p \in mset_r(t)$
$\wedge\ (\forall s \in [t\text{-}d,t]: \neg M_p(s)) \Rightarrow p \notin mset_r(t)$

$\wedge \neg\ M_r(t) \Rightarrow\ mset_r(t) = \{\}.$

(AA) $\forall s,t \in RT,\ \forall p,q \in \mathcal{P}:$

$\wedge\ C_p(s) = C_q(t)$
$\wedge\ M_p(s) = M_q(t)$
$\Rightarrow\ mset_p(s) = mset_q(t).$

(AS) $\forall p,q \in \mathcal{P},\ \forall t \in RT:\ M_p(t)$

$\Rightarrow\ \exists s:\ s \leq t \leq s+nc,\ \forall u \in [s,s+nc]$
$\wedge\ q \in mset_p(t) \wedge M_p(u) \Rightarrow q \in mset_p(u)$
$\wedge\ q \notin mset_p(t) \wedge M_p(u) \Rightarrow q \notin mset_p(u).$

The asynchronous membership specification introduces a new condition (AT = asynchronous timeliness) that excludes trivial solutions like setting the indicators of all processes to false. Any process p that is in a stable partition for at least x time units has to set its indicator M_p to true and whenever p is not up, its indicator has to be false.

(AT) $\forall p \in \mathcal{P}$, $\forall t \in RT$, $\forall Q \subseteq \mathcal{P}$:

$\wedge\ \neg up_p(t) \Rightarrow \neg M_p(t)$

$\wedge\ stablePartition(Q,t\text{-}x,t) \wedge p \in Q \Rightarrow M_p(t)$.

Note that this condition requires that whenever a process p is in a stable partition, it has to be up. Otherwise, this condition cannot be satisfied because p's indicator would have to be true and false at the same time. This condition also requires that one has to define that the indicator of a crashed process is assumed to be false.

Processes have to exchange messages to keep their member-sets up-to-date. Thus, processes in different communication partitions cannot agree on a common member-set because they cannot communicate with each other. Since at any clock time T all processes that have an up-to-date member-set agree on their member-set by requirement (AA), the specification defines a "primary partition" membership service in the sense that processes in at most one partition can have an up-to-date member-set. The partition in which the processes can keep their member-sets up-to-date is called the *primary partition*.

4.6 Partitionable Systems

Due to space restrictions, we cannot include in this paper the derivation step for partitionable systems. Please, get the complete paper via the internet:

http://www-cse.ucsd.edu/users/cfetzer/DFAMS/dfams.html

5 Conclusion

We derived the specifications of two fail-aware membership services that are implementable in timed asynchronous systems. These specifications are very similar to the specification of a synchronous membership service. All processes that are stable ("timely") have to keep their member-sets up-to-date. Slow processes that cannot keep their member-set up-to-date, have to signal this to their clients so that the clients know that they cannot use the information provided by their membership server. For example, a process with an out-of-date membership is not allowed to participate in an election protocol that is based on the membership.

The derivation consists of four major steps. We first defined an ideal instantaneous membership service that provides processes with a perfect view of what processes are currently up. We transformed that ideal specification to derive a specification that is implementable in timed asynchronous systems. The first derivation step introduces non-zero detection times: processes have up to d time units to detect the crash or the recovery of another process. The second

step introduces approximately synchronized clocks: instead of requiring perfect clocks, processes are now allowed to use clocks that are up to Ψ apart from each other. The third step introduces the possibility that processes cannot keep up with the other processes due to performance failures: a process that cannot keep up, is allowed to signal this condition to its clients to let them know that they cannot depend on the information provided by the process. The final derivation step makes sure that processes in different network partitions can make progress independently of each other: processes are allowed to name their local partition and have only to agree on the set of processes that are in their local partition.

We showed in [11] that the two derived fail-aware specifications allow an efficient implementation in timed asynchronous systems. Like time-free systems, timed asynchronous systems do not allow processes to decide if another process is crashed or just slow. The reason why timed asynchronous systems allow the implementation of a useful membership service while time-free system (without being extended by a failure detector) do not [2] is mainly due to the availability of hardware clocks and the use of *conditional timeliness requirements*: in time-free systems one requires that when a process q crashes, at least one process will eventually install a new view [2] while in timed systems we require that each *stable* process p will remove q from its member-set in a bounded amount of time. Since stable processes have the same "synchronicity" as the processes of a synchronous systems, they can achieve an agreement that q has to be removed from the membership within a bounded amount of time.

References

1. E. Anceaume, B. Charron-Bost, P. Minet, and S. Toueg. On the formal specification of group membership services. Technical Report 95-1534, Computer Science Department, Cornell University, Ithaca, 1995.
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. F. Cristian. Reaching agreement on processor-group membership in synchronous distributed systems. *Distributed Computing*, 4:175–187, 1991.
4. F. Cristian. Automatic reconfiguration in the presence of failures. *Software Engineering Journal*, pages 53–60, Mar 1993.
5. F. Cristian and C. Fetzer. The timed asynchronous system model. Technical Report CS97-519, UCSD, Jan 1997. Available at http://www-cse.ucsd.edu/users/-cfetzer/MODEL/model.html.
6. 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.
7. D. Dolev, J. Y. Halpern, and R. Strong. On the possibility and impossibility of achieving clock synchronization. *Journal of Computer and System Science*, 32(2):230–250, 1986.
8. 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. Available at http://www-cse.ucsd.edu/users/-cfetzer/CONS/cons.html.
9. 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. Available at http://www-cse.ucsd.edu/-users/cfetzer/FA/fa.html.
10. C. Fetzer and F. Cristian. A fail-aware datagram service. In *Proceedings of the 2nd Annual Workshop on Fault-Tolerant Parallel and Distributed Systems*, Geneva, Switzerland, Apr 1997. Available at http://www-cse.ucsd.edu/users/-cfetzer/FADS/fads.html.
11. C. Fetzer and F. Cristian. A fail-aware membership service. In *Proceedings of the 16th Symposium on Reliable Distributed Systems*, Oct 1997. Available at http://-www-cse.ucsd.edu/users/cfetzer/FAMS/fams.html.
12. 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. Available at http://www-cse.ucsd.edu/-users/cfetzer/FAPS/faps.html.
13. C. Fetzer and F. Cristian. A highly available local leader service. In *Proceedings of the Sixth IFIP International Working Conference on Dependable Computing for Critical Applications*, Grainau, Germany, Mar 1997. Available at http://www-cse.-ucsd.edu/users/cfetzer/HALL/hall.html.
14. 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.
15. M. Hiltunen and R. Schlichting. Properties of membership services. In *Proc. 2d Int. Symp. on Autonomous Decentralized Systems*, Phoeniz, AZ, Apr 1995.
16. F. Jahanian, S. Fakhouri, and R. Rajkumar. Processor group membership protocols: Specification, design and implementation. In *Proceedings of the 12th Symposium on Reliable Distributed Systems*, pages 2–11, Princeton, NJ, Oct 1993.
17. L. Lamport. How to write a long formula. Technical Report SRC119, System Research Center of Digital Equipment, Palo Alto, Ca, Dec 1993.
18. P. M. Melliar-Smith, L. E. Moser, and V. Agrawala. Processor membership in asynchronous distributed systems. *IEEE Trans. on Parallel and Distributed Systems*, 5(5):459–473, May 1994.
19. S. Mishra, L. Peterson, and R. Schlichting. A membership protocol based on partial order. In J. F. Meyer and R. D. Schlichting, editors, *Dependable Computing for Critical Applications 2*, pages 309–331. Springer-Verlag, Wien, 1992.
20. G. Neiger. A new look at membership services. In *Proceedings of the Fifteenth Annual Symposium on Principles of Distributed Computing*, pages 331–340, Philadelphia, May 1996.
21. F. Schmuck and F. Cristian. Continuous clock amortization need not affect the precision of a clock synchronization algorithm. In *Proceedings of Ninth Annual ACM Symposium on Distributed Computing*, 1990.

This article was processed using the LaTeX macro package with LLNCS style