Fail-Awareness: An Approach to Construct Fail-Safe Applications

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

Abstract

We present a framework for building fail-safe hard real-time applications on top of an asynchronous distributed system subject to communication partitions, i.e. using processors and communication facilities whose real-time delays cannot be guaranteed. The basic assumption behind our approach is that each processor has a local hardware clock that proceeds within a linear envelope of real-time. This allows to compute an upper bound on the actual delays incurred by a particular processing sequence or message transmission. Services and applications can use these computed bounds to detect when they cannot guarantee all their properties because of excessive delays. This allows an application to detect when to switch to a fail-safe mode.

1 Introduction

In recent years there has been a trend to use commercial-off-the-shelf (COTS) products such as real-time Unix and main-stream hardware platforms to build hard real-time systems. A system is *hard real-time* if the consequence of a non-masked performance failure can be catastrophic [13]. Hard real-time systems can coarsely be classified into *fail-safe* and *fail-operational* systems. Fail-safe systems have at least one safe state and the system has to transit to such a state when a non-maskable component failure occurs. The motivation for using COTS is to cut costs while still using the latest technologies. From a technical point of view COTS usage is quite challenging with respect to the construction of distributed hard real-time systems. To explain this, note that several recently built distributed hard real-time systems [11, 20, 12, 1] rely on the *guaranteed response paradigm* [13]. This paradigm depends on the assumption that the maximum number of failures per time unit is a priori known to guarantee that the real-time system reacts to events occurring in the controlled object within an a priori known time bound. However, if this *failure assumption* can be violated at run-time, the real-time system can be subject to unpredictable behavior. To bound the number of performance failures per time unit, one has to know an upper bound on the processor and network load. However, the usage of COTS software and hardware does not necessarily allow to bound the peak processor and network load a priori because the load induced by these products is not known and can only be estimated using measurements. For example, due to interrupts, caching, and bus arbitration it is very difficult to determine the worst case execution times for main-stream hardware platforms [17]. Moreover, the load induced by the application and middleware products depends also on the point of operation of the system and for many systems the envelope of operation is not well known. The occurrence of non-maskable performance failures is therefore difficult to avoid by design. We present an approach to address the problem of non-maskable performance failures.

Due to the problems posed by the guaranteed response paradigm, many practical systems are based on the *best effort paradigm* that does not guarantee that a real-time system always responds within the required time bounds. However, the system has to show in empirical tests that it statistically responds in a timely fashion. We describe a best effort approach to constructing fail-safe distributed hard real-time applications for partitionable systems: *fail-awareness.* In a *partitionable system* the set of processes can split into disjoint subsets due to network failures or excessive performance failures. Each such subset is informally referred to as a *(communication) partition.* The general goal of our approach is as follows: as long as the number of failures per time unit experienced by the underlying communication and process services remains below a given bound, all services provide their *standard* synchronous (i.e. hard real-time) semantics and each server knows this fact; when the number of failures per time unit rises above that bound, a server is allowed to switch to a specified *exception* semantics. Clients can learn if a server provides its standard or exception semantics by examining an *exception indicator* provided by each server. An application can use the indicators of the servers it depends upon to switch the system to a safe state when the occurrence of non-maskable performance failures causes some underlying services to switch to their exception semantics.

The novelty of our approach is that instead of aiming for real-time support in an asynchronous, partitionable setting only by providing high throughput (see Transis [3] and Totem [15]), we specify the standard semantics of services using real-time deadlines and provide mechanisms to detect when an applica-

---

*This research was partially supported by a grant from the Air Force Office of Scientific Research.

0731-3071/97 $10.00 © 1997 IEEE

tion cannot depend upon the standard semantics of services due to unmasked performance (or omission or crash) failures. This detection is essential for fail-safe applications that have to switch to a fail-safe mode whenever they cannot guarantee their standard synchronous properties. Our approach allows the servers in each partition to "make progress" independently of the servers in other partitions, i.e. these servers can provide their standard semantics and hence, increase the availability of the system. In this paper we give an overview of several fail-aware partitionable services that we have designed such as clock synchronization, membership, and atomic broadcast to illustrate our approach.

2 Model Rationale

The guaranteed response paradigm is based on the assumption that the classes of likely failures and their maximum number per time unit is a priori known. The fail-awareness paradigm also assumes knowledge of the classes of likely failures, but does not assume aything about their maximum frequency of occurrence (since the number of performance failures per time unit cannot be bounded due to the use of COTS products). That difference in the underlying assumptions results in the use of different system models for the guaranteed response paradigm (i.e. use of a synchronous system model) and the fail-awareness paradigm (i.e. use of the timed asynchronous system model [2]). We review in this section the basic differences between these two models.

In a *completely synchronous system* the real-time delays of all processes and all messages are within a priori known bounds. One can generalize the notion of a synchronous system by allowing a bounded number of "performance failures" per time unit. To define performance failures, one first introduces thresholds for process scheduling (σ) and message transmission delays (δ). When the transmission delay of a message m is greater than the maximum assumed message delay δ, one says that m suffers a performance failure. Otherwise, m is said to be *timely*. Similarly, when the scheduling delay of a process p is greater than the maximum assumed scheduling delay σ, p suffers a performance failure. A process that does not suffer any performance failures in a given interval is said to be *timely*. A *failure model* specifies what kind of failures have to be considered in the design of a system, i.e. the probability that any other kind of failure occurs is negligible. A typical failure model used in synchronous systems assumes that processes have a crash/performance failure semantics and messages have an omission/performance failure semantics. A *failure assumption* states the maximum number of (performance, crash, and omission) failures that can occur per time unit, i.e. the probability that more failures occur is negligible. Since most distributed protocols are "round based", a failure assumption typically states the maximum number of failures per round, i.e. a "time unit" is the maximum length of a round.

A *synchronous system* requires that the classes of failures that can occur and the maximum number of these failures per time unit be a priori known. Knowing *what classes* of failures can occur (stated in the failure model) and knowing the *maximum number of these failures* per time unit (stated in the failure assumption), one can use a sufficient amount of redundancy to mask all failures that can occur by hypothesis. Thus, when the failure model and assumptions are correct, one can exclude the occurrence of non-maskable failures, that is, system failures, by design. In other terms, the probability that the system masks all failures is at least as high as the probability that the failure model and the failure assumption are valid [16].

To bound the maximum number of performance failures per time unit, one has to bound the peak processor and network load. Using commercial software packages with unknown peak load therefore increases the difficulty of deriving a well founded maximum number of performance failures per time unit. For example, using a network of workstations with a standard operating system like Unix does in many cases not allow a reasonable failure assumption to be made in the sense that the probability that the failure assumption can be violated will be negligible. We use therefore the *timed asynchronous system model* as the foundation of our work since it does not put any bound on the number of failures per time unit. For a detailed description and comparison with other models like the quasi-synchronous model of [19] see [2].

The timed asynchronous model assumes that processes have access to a local *unsynchronized* hardware clock with a bounded drift rate, i.e. they proceed within a linear envelope of real-time. It uses the following failure model: processes have crash/performance failure semantics and messages have omission/performance failure semantics. In what follows, when we use the generic term "failure" we mean a failure that belongs to one of *these* classes of failures. This model does not define any bound on the maximum number of failures per time unit, i.e. it has no failure assumption. It is an accurate description of existing distributed systems like a network of workstations running Unix or Windows NT. The model also allows the system to split into partitions when all messages sent between processes (in different partitions) suffer omission or performance failures. Unlike in the synchronous system model, the timed asynchronous model does not necessarily allow to mask all failures that occur since any amount of redundancy used to mask failures can be exceeded by the actual number of failures that occur. The fail-aware protocols we have been designing have nevertheless some resemblance to synchronous protocols because they behave like synchronous protocols as long as the number of failures per time unit is within some give bound. However, the fail-aware protocols have to deal with situations when processes become partitioned, partitions merge, or the number of failures per round becomes that high that servers have to switch to their exception semantics.

3 Fail-Awareness

Since our main interest is in fail-safe applications, we base our approach on the following idea: instead of depending on the assumption that the number of failures per time unit never rises above a given bound, we require that

- services mask all failures as long as the the number of failures per time unit is within some given threshold, and
- a server detects when it cannot provide its standard (synchronous) semantics anymore (because the number of failures is above the threshold) and signals that condition to its clients using an *(exception) indicator* (see also Section 5.2), i.e. the servers are *fail-aware.*

Typically, some processes of an application will monitor the servers the application depends upon and switch the system to a safe state when "too many" servers cannot provide their standard semantics anymore. The communication between these processes to coordinate the switch to a safe state should – whenever possible – use communication based on measuring the passage of time. This ensures that communication takes place between these processes even when the network is overloaded or partitioned (see Section 6 and [8] for examples). We will refer to this type of communication as *communication by time.* In our designs we mainly use a *(time) locking* mechanism to facilitate communication by time between processes (see Section 5.3).

The specifications of the fail-aware services we have designed so far are typically derived from the specifications of the corresponding synchronous services, i.e. services that were originally specified to be implemented in synchronous systems. We transform the specification S of a synchronous service into a new, but similar, specification FA so that FA becomes implementable in timed asynchronous systems that are characterized by having no bound on the number of failures per time unit and the possibility of communication partitions. Fail-awareness for partitionable systems is based on the concept of a *logical partition* [9] to represent communication partitions: a logical partition consists of a unique id and a sequence of memberships [5].

The transformation of a synchronous specification S into a fail-aware specification FA is done in four steps:

- the interface of FA is augmented with an exception indicator,
- an FA server is required to provide its standard semantics, defined to be identical or very close to the synchronous semantics S, whenever its indicator signals that it is part of a logical partition,
- otherwise, when the indicator signals that a server is not part of a logical partition, it has to provide a specified exception semantics, and
- the indicator of a server p in a communication partition SP must signal that it is part of some logical partition that contains SP whenever the communication and process services in SP exhibit "synchronous behavior", that is, the number of failures per time unit within SP is within some a priori given bound. We call SP a *stable partition.* A formal definition of a stable partition can be found in [2].

The detection that a server cannot provide its standard semantics anymore is based on several mechanisms that we detail in Section 5. Note that a server does not actually have to decide if it is in a stable partition or not since (1) a server has to guarantee by design (i.e. by masking failures) that it provides its standard semantics as long as it is part of a stable partition, (2) it has to detect when it cannot mask all failures and hence, cannot provide its standard semantics anymore (i.e. by design this can only happen when it is not part of a stable partition), and (3) a server typically provides its standard semantics as long as it masks all failures even though it might not be part of a stable partition. Section 5 also describes our implementation of an exception indicator: an indicator has to enable clients to detect that a server cannot provide its standard semantics even when the server suffers performance failures. This allows the processes that monitor the servers (and switch the system to a fail-safe mode) to query the current semantics of a server at any point in time.

4 Fail-Aware Services

We have designed and implemented a hierarchy of fail-aware services to support the design and implementation of fail-safe real-time applications (see Figure 1) [8]. The foundations of the hierarchy are an asynchronous datagram service and a process management service that provide the semantics assumed by the timed asynchronous system model, i.e. messages have omission/performance and processes have crash/performance failure semantics. We give an overview of the goals of the different fail-aware services in the presence of partitions. We refer the reader to [8] for a description and performance measurements of our implementation of the protocol stack.

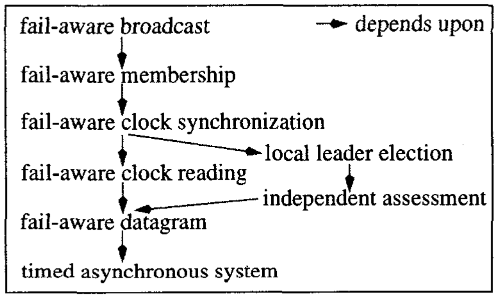

Figure 1: Hierarchy of fail-aware services to support the design of fail-safe partitionable real-time applications.

4.1 Partitionable Systems

When it comes to designing distributed protocols, the ideal underlying system is one that is completely synchronous: each pair of non-crashed processes p and

q is *0-connected*, i.e. p and q are timely and each message sent between p and q is timely (see [2] for a formal definition). Since processes and messagees do not suffer performance failures, this greatly simplifies protocol design. In practice, systems are often not completely synchronous, in particular, we are considering systems in which the probability that partitions occur is not negligible. When a system splits into partitions, the ideal situation (with respect to the design of protocols) would be that each partition shows completely synchronous behavior (see Figure 2): all process pairs in a partition are 0-connected and they are *disconnected* from processes outside their partition, i.e. they do not receive any messages from other partitions (see [2] for a formal definition). If a system only splits into such "ideal partitions", the design of protocols would be reasonably simple since the protocols do not have to handle situations like the inability of some processes to communicate with other processes in the same partition, or sporadic message arrivals from other partitions.

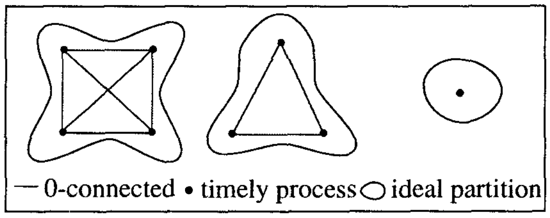

Figure 2: Ideally all processes in a partition should be mutually 0-connected with all processes in the same partition and be disconnected from all processes in other partitions.

Real communication partitions are not always ideal in the above sense. For example, for some time interval a process r might be linked to a process q by a "one-way connection", i.e. a connection that allows r to send timely messages to q but does not allow r to receive timely messages from q (see Figure 3). Due to local network overload, two processes n and o might be linked by a "slow-connection", i.e. all messages sent between n and o suffer performance failures. Instead of each process pair p,q in a partition being 0-connected, they could only be *F-connected* [2] for some $F > 0$, i.e. p and q are timely and at most F messages per time unit sent between p and q suffer omission/performance failures. In what follows, we use the term "connected" to denote "F-connected" for some fixed F. Note that the connected-relation in a partition might also not be transitive, i.e. each of the process pairs (o, k) and (k, q) might be connected while process pair (o, q) is not connected (see Figure 3).

The goal of the fail-aware protocol hierarchy is to provide – whenever possible – an application with an abstraction similar to that of an ideal partition to reduce the complexity of programming distributed real-time applications for partitionable systems: a *logical partition*. When a server is not part of a logical partition, it has to switch to its exception semantics. To make logical partitions similar to ideal partitions, we require that logical partitions do not overlap and all processes in a logical partition be able to communicate via *atomic broadcasts* with all processes in their logical partition in a timely manner and that they be "broadcast-disconnected" from processes in other logical partitions, i.e. they only receive broadcasts that were sent by processes within their logical partition. Thus, the fail-aware services have to provide an application with a view of the system in which the connected-relation (with respect to atomic broadcasts) is transitive. There are two main approaches to achieve this goal:

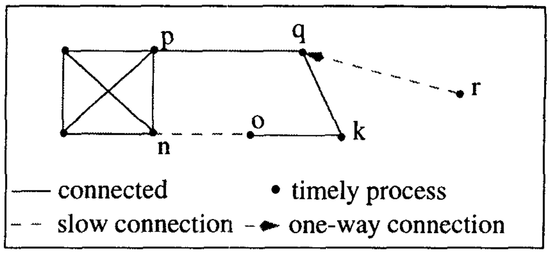

Figure 3: Real partitions are not always 'ideal': slow connections, one-way connections, and non transitivity of the connected-relation complicate the protocol design.

- *message forwarding:* a process o can send messages via process k to destination q when o is disconnected from q while k is connected to q (see Figure 3). One way to implement message forwarding is message diffusion, i.e. a sender of a message m sends m to all connected processes and each process q that receives m sends m to all connected processes unless q is the destination of m or q has already forwarded m. Message diffusion can be prohibitively expensive for many applications.
- *removing connections:* even though two processes p and q can communicate via datagram messages, higher level protocols are forbidden to send broadcast messages between p and q since they are in different logical partitions. Removing "too many" connections could however split the system in partitions "too small" to do useful work.

Our approach tries to combine these two extreme approaches: (1) use a limited amount of forwarding at the level of the broadcast service, and (2) logically disconnects (when necessary) some processes even though they are capable of communicating with each other.

The foundation of our fail-aware protocol stack is the *fail-aware datagram service* [7]. Its purpose is to reject messages that arrive via slow or one-way connections (see Figure 4). This service computes an upper bound on the transmission delay of each message it delivers (see Section 5.1). The implementation of this service only depends upon the fact that hardware clocks proceed within a known linear envelope of real-time; the service does not need synchronized clocks.

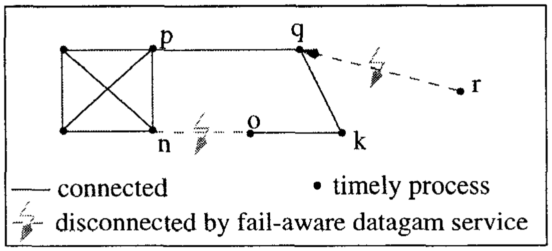

Figure 4: The fail-aware datagram service allows to reject messages that arrive via one-way or slow links.

The calculated upper bound for a timely message (i.e. the transmission delay is $\leq\delta$) is at most some known Δ ($\geq\delta$), while the upper bound for a message sent via a one-way or a slow connection is greater than Δ. Higher level services, such as local leader election, reject messages with a calculated upper bound greater than Δ time units.

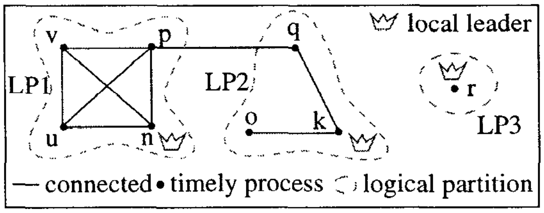

Figure 5: The local leader election service elects local leaders and creates non-overlapping logical partitions each consisting of the 'supporters' of a local leader.

The next step is to combine processes into logical partitions. This is the goal of the *local leader election* service [9]. It has to create logical partitions such that each set S of processes that are mutually connected (i.e. each two processes in S are connected) is in the same logical partition. For example, all processes in *LP1* of Figure 5 are mutually connected and therefore have to be in the same logical partition. To create logical partitions, the local leader protocol tries to elect a process in each communication partition as local leader (see [9] for details): a process r ***supports*** the election of the process l only if l's identification is smaller than the id of any other process that r is connected with. A process l becomes local leader only if it has the support of all processes it is connected with. A local leader l creates a logical partition LP that contains all processes that support l's election. The id of LP is unique. For example, in the situation illustrated in Figure 5 the protocol elects processes n, k, and r as local leaders (assuming the following order $n<o<...<u<v$). Because processes q, o and k support k's election, k creates a logical partition ($LP2$) consisting of the processes o, q and k. The local leader election service ensures that at no point in time two logical partitions overlap, that is, at any point in time a process is in at most one logical partition. This non-overlapping is ensured using the time locking mechanism (Section 5.3): a process p stays in a logical partition LP for only a bounded amount of time before p has to acknowledge that it wants to stay in LP and thus, p can use its local hardware clock to make sure that it is removed from all logical partitions before it joins a new logical partition [9]. Like the set of processes that form a communication partition can change by processes joining or leaving the partition, processes can leave and join a logical partition. The fail-aware clock synchronization service ensures that the deviation between the clocks provided by each of the servers with the same logical partition is bounded by some a priori given constant Ψ. The membership service [5] keeps track of the set of processes in a logical partition and guarantees that all processes in a logical partition agree (at any point in clock time) on the current members of that logical partition.

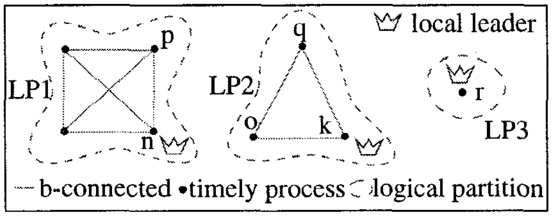

Figure 6: The broadcast service connects all processes in a logical partition and disconnects them from the processes in all other partitions.

The next problem we address is that the *connected* relation of a logical partition is not necessarily transitive. It is the goal of the fail-aware ***atomic broadcast*** service to ensure that a broadcast in a logical partition LP is delivered to all processes in LP and to no process outside of LP. To ensure that all these processes get all broadcasts, the local leader k of a logical partition forwards a broadcast message m to a process q when m is sent by some process o that is not connected to q. When the forwarding also fails, the processes that cannot deliver all broadcasts are removed from the logical partition and these processes have to switch to their exception semantics. The broadcast service connects all processes in a logical partition via broadcast messages, making the *b-connected* ("broadcast connected") relation transitive (see Figure 6).

4.2 Fail-Awareness Properties

All fail-aware services of our protocol stack have a standard semantics that is similar to that of the corresponding synchronous services. Due to space constraints we can only sketch a few properties of some of the fail-aware service (for a detailed description of their semantics and implementations see [4, 5, 9, 7]). The *fail-aware clock synchronization* service synchronizes the clocks of all clock servers in a logical partition. Each server p provides an exception indicator I_p. The indicator tells p's clients if p's clock C_p is synchronized or not: I_p shows the id of p's logical partition when C_p is synchronized, and otherwise, it shows *out-of-date* (denoted by $\perp$). The service provides the following property: **(BD)** when two servers p and q are in the same logical partition at real-time t, their clocks are at most Ψ apart from each other, where Ψ is an a priori given constant,

$$I_p(t)=I_q(t)\neq\perp \Rightarrow |C_p(t)-C_q(t)|\leq\Psi.$$

Our implementation of the service makes sure that the indicator I_p of server p will show $\perp$ before p's clock can be more than Ψ apart from the other clocks in p's current logical partition. We will show in Section 5 how to achieve that even when p suffers a performance failure.

The *fail-aware membership* service takes the logical partition LP created by the local leadership service [9] and maintains agreement on the membership of LP among the members of LP (see [4] for a detailed description of the semantics). Each server p maintains an indicator MS_p that shows the id of p's current logical partition and a set $mset_p$ that shows the current members of p's logical partition. The service ensures that at any clock time T when the indicators MS_p and MS_q of two servers p and q show the same logical partition id LP, then the two servers agree on the membership of LP:

$$MS_p(T)=MS_q(T) \Rightarrow mset_p(T)=mset_q(T).$$

A fail-aware membership service ensures that departures and joins of servers are detected and result in a new membership of the logical partition within a known amount of time. In particular, the fail-aware membership service has also to ensure that (1) whenever a server can keep up with the servers in some logical partition LP (i.e. the server agrees with the processes in LP on the membership of LP), it will be included in the membership of LP, and (2) whenever a server cannot keep up with the other servers in LP (i.e. its indicator shows $\perp$, it is crashed, or in an other communication partition), it is removed from the membership of LP. Since servers agree on the membership of their logical partition LP at any point in clock time, they agree of course on the order in which servers are included in or removed from the membership of LP. Each server q that cannot update its membership in time has to set its indicator to $\perp$ to signal to its clients that its membership information is out-of-date. The service has to guarantee that at any point in real-time a process is in the membership of at most one logical partition, i.e. the memberships of two logical partitions never overlap. Our implementation of the membership service [5] uses the messages sent by the local leader election service to reach agreement on the members of a logical partition and the time locking mechanism (see Section 5.3) to guarantee that the memberships of two logical partitions do not overlap.

The *fail-aware atomic broadcast* service delivers messages within a constant Ω clock time delay after they are sent and uses their send time stamps and their sender's id to totally order all delivered messages. A broadcast message is time-stamped by reading the synchronized clock C_p of the sender p. Thus, the broadcast service guarantees causal delivery even in the presence of a hidden channel like a file-system whenever the delay of the channel is greater than the maximum deviation between clocks [14] (which is of the order of a few milliseconds in our implementation). The service ensures the atomicity property that either all servers or no server in a logical partition deliver a broadcast message. A server that (1) does not deliver all broadcasts in time, or (2) has broadcasted a message that is not delivered in its logical partition, has to signal to its clients that it is out-of-date. Each server p maintains therefore an indicator BI_p. More precisely, when a server q broadcasts a message m at clock time T, then either (A) all processes in LP deliver m by $T+\Omega$ (i.e. the indicators of all processes that have not delivered m by $T+\Omega$ must not show LP by $T+\Omega$), or (B) no process delivers m and q signals by $T+\Omega$ to its clients that not all messages it has broadcasted are delivered in LP by setting its indicator to $\perp$. When server p's indicator shows LP at clock time T, i.e. $BI_p(T)=LP$, p knows that it has delivered all broadcast messages delivered in LP no later than T and that all messages it has broadcasted before $T-\Omega$ are delivered by all processes in LP. When a process p cannot keep its broadcast indicator up-to-date, it is removed from the membership of its previous logical partition LP within a bounded amount of time, i.e. all processes in LP learn that p has not necessarily delivered in a timely fashion all broadcasts that are delivered in LP.

The broadcast service uses fail-aware datagram broadcasts to send broadcast messages to the other servers. A local leader l decides what broadcast messages are delivered in its logical partition and in what order. It piggy-backs that ordering information on the datagrams sent by the leader election protocol. When the "early delivery option" of the broadcast service is activated, the local leader broadcasts additional ordering datagrams to allow the servers an earlier delivery of broadcast messages. The local leader rejects all broadcast messages from other logical partitions and all broadcasts that arrive in slow datagrams, i.e. with a transmission delay greater than Δ. Since the broadcasts are ordered with respect to their send time stamps, the local leader waits for $\Delta+\Psi$ before it orders a broadcast message to make sure that no broadcast message with an earlier send time stamp arrives. When the local leader detects that some server has not received all broadcast messages, it forwards these messages to that server. A server that does not deliver all broadcast messages in time has to set its exception indicator to $\perp$. A local leader can re-integrate such out-of-date servers by transferring them the current state of the logical partition that it maintains.

5 Mechanisms

Most of the fail-aware protocols we have designed use time redundancy to mask a bounded number of performance failures per time unit ("round"). Because the number of failures per time unit cannot be bounded a priori, not all performance failures are necessarily maskable. Since such non-maskable failures can lead to system failures, fail-safe applications require their detection so that they can switch to a safe state. We review some of the mechanisms we use to detect performance failures.

5.1 Fail-Aware Datagrams

The fail-aware datagram service [7] calculates an upper bound on the transmission delay of some mes-

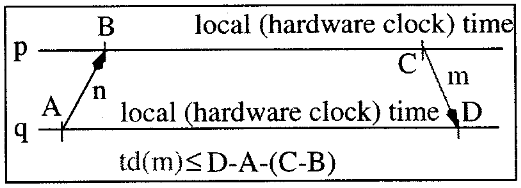

Figure 7: When the drift rate of hardware clocks can be neglected, the transmission delay of m can be bounded by $(D - A) - (C - B)$ even though p's and q's hardware clocks are not synchronized.

sage m using the four time-stamps (each taken with local unsynchronized hardware clock) of some round-trip n, m (see Figure 7). The service makes sure that two connected processes p and q exchange periodically messages such that when p sends a message m to q it can piggy-back the time stamps of some message n that p has previously received from q. This enables q on the reception of m to calculate an upper bound for m. Since the drift rates of hardware clocks are in general very small (in the order of 10^{-6}), even when p has received n multiple seconds before sending m, the increase of the upper bound for m due to the maximum hardware clock drift rate is very small.

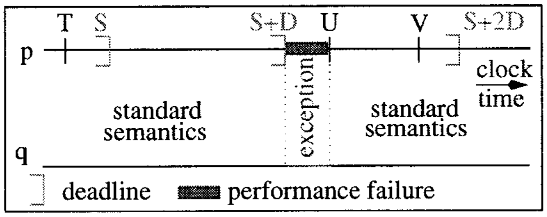

Figure 8: Process p has to adjust its clock before hardware clock times S, $S + D$, and $S + 2D$. It actually performs the adjustments at times T, U, and V. Since p misses deadline $S + D < U$, a process q that reads p's clock between $[S + D, U]$ has to detect that C_p is not synchronized.

5.2 Indicators

Real-time communication protocols can be divided into two broad classes [13]: *time triggered* and *event triggered*. Event triggered systems react to events directly while time-triggered systems react only at predefined points in time. Orthogonal to the above classification, protocols can also be classified as *clock-driven* or *timer-driven* [18]. Clock-driven protocols rely on synchronized clocks while timer-driven protocols rely on (unsynchronized) timers. All our fail-aware protocols are event-triggered and all protocols above the fail-aware clock synchronization layer are clock-driven. We chose event-triggered protocols since operating systems like Unix have relatively good reaction times to events like a message reception but they have poor real-time scheduling support. The protocols are clock-driven because clock-driven protocols simplify the implementation of indicators: a process knows the clock deadline beyond which it cannot provide its standard semantics anymore, unless 'something good' happens before that deadline. Our indicator design relies on this knowledge to signal when a server starts providing its exception semantics.

To explain how clock-driven protocols help to maintain indicators, consider a simple fail-aware clock synchronization protocol. A process p has to adjust a clock C_p periodically, say, before its local hardware clock shows values S, $S + D$, ... to keep its clock synchronized (see Figure 8). When process p does not adjust its clock before the given deadlines, its clock C_p is not necessarily synchronized to the other clocks. Let process q be another process that is executed on the same computer node as p (p and q use the same hardware clock H_p). When process q tries to read C_p between $S + D$ and U (measured by H_p), q has to detect that C_p is out of synchrony. We achieve that by using the following mechanism (see Figure 9). An indicator I_p of a process p consists of two parts: (1) the identification of p's logical partition (*lpartition*), and (2) the expiration time (*expTime*) beyond which the indicator has to signal that p provides its exception semantics. When process q evaluates I_p, it first reads the local hardware clock H_p. If its hardware clock shows at most time *expTime*, the value of I_p is *lpartition*. Otherwise, the value of I_p is *out-of-date* ($\perp$) which tells q that p provides its exception semantics. Process p updates its indicator periodically, e.g. at time T it sets the expiration time to its next deadline $S + D$ (see Figure 8). When p suffers a performance failure, it does not update its indicator in time and hence, any client that reads the indicator will evaluate I_p to *out-of-date*. For example, when q evaluates I_p during interval $(S + D, U)$, the expiration time is $S + D$ while H_p shows a value greater than $S + D$. Thus, I_p returns value *out-of-date* that allows q to detect that C_p is not in synchrony anymore.

A process q that reads I_p might itself suffer a performance failure while evaluating the current value of I_p. An indicator I_p therefore returns the hardware clock time stamp used in its evaluation. This allows the detection of performance failures that occur during the evaluation of I_p or during the usage of the value returned by I_p.

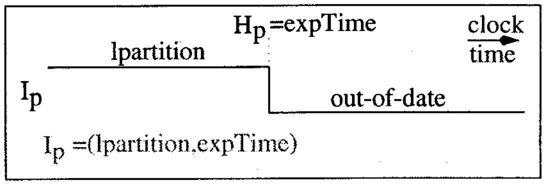

Figure 9: The indicator of a process p consists of the logical partition *lpartition* of p and an expiration time *expTime*, i.e. the time when the indicator will become *out-of-date*.

5.3 Locking Mechanism

Several of our protocols, e.g. the leader election service, use a *locking mechanism* to communicate by the passage of time. This mechanism is similar to the 'leases' mechanism of [10] which was introduced to ensure cache consistency in distributed file systems. The locking mechanism requires only one fail-aware datagram message instead of a round-trip message pair (or synchronized clocks) used in the leases mechanism.

The advantage of using time for interprocess communication is that – when applied properly – this communication is possible even when the communication partners become partitioned. This mechanism is in particular useful for the coordination of the processes that switch the system to a safe mode (see Section 6). The locking mechanism works as follows (see Figure 10). A process p sets some local variable LV to a value V and sends a message m to a process q at t telling q that it will not change the value of LV for at least $lockTime$ real-time units. When process q receives m at s, it knows that p will not change LV for at least $lockTime - td(m)$ time units, where $td(m) = s - t$ is the transmission delay of m. Process q can use the upper bound $ub(m) > td(m)$ calculated by the fail-aware datagram service to determine a lower bound for the time p will not change LV, i.e. at least until time $u \overset{\Delta}{=} s - ub(m) + lockTime$. The interesting part is that at time $t + lockTime$ process p can change the value of LV without having to notify q about that change because q will not use its knowledge that LV equals V beyond time $u \leq t + lockTime$. For example, consider that p lets q know by m that p wants to be part of q's logical partition LP until at most time $t + lockTime$, then p can try after time $t + lockTime$ to become part of another logical partition without sending q another message since q will remove p from LP before $t + lockTime$. In other words, the locking mechanism can be used to guarantee that a process is at any point in real-time in at most one logical partition.

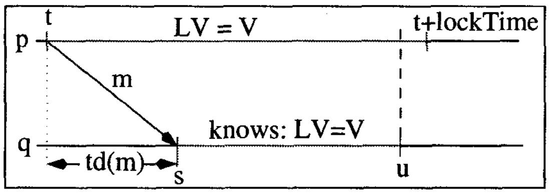

Figure 10: A process p guarantees not to change its variable LV for at least $lockTime$ time units. Process p knows that q will use this information (transmitted in m) at most to some time $u \leq t + lockTime$.

6 Traffic Signaling Example

We illustrate in this section how fail-aware services could be used in a practical setting in which computer controllers that are physically close to sensors or actuators are linked by a communication network. Such systems of distributed controller are common in factory floor control applications. Space considerations do not allow us to describe such a real-world application. Instead, we will illustrate the use of fail-awareness on a simpler, although slightly contrived example: a synchronized traffic signaling application.

We consider a system with two intersections (see Figure 11). There are four traffic lights per intersection. Each of the four directions of an intersection is sensed by a pair of sensors. During normal operation at least one of the two sensors of each pair has to be "operational", otherwise, the subsystem controlling the intersection has to switch to a "round robin" mode to guarantee a certain amount of fairness for all cars. For each intersection there are two safe states: (A) all four traffic lights show red, or (B) all four traffic lights flash red. The system should only transition to safe state (A) for a bounded amount of time before it transitions to state (B). During non-partitioned operation the traffic lights of the two intersections have to be synchronized to maximize the flow of cars. When the system partitions, the intersections are allowed to be controlled independently of each other.

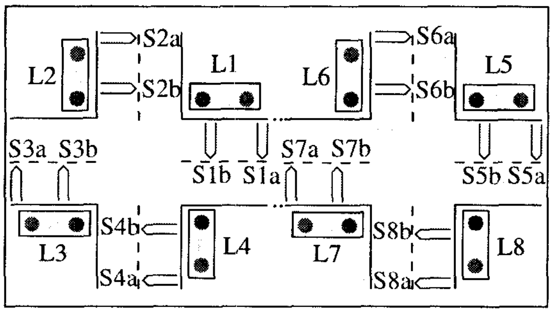

Figure 11: The traffic lights (L1-L8) of two intersections have to be synchronized. There are two sensors (S?a,S?b) for each traffic light (L?) (and there are two redundant controllers for each of the two intersections).

The distributed subsystem that controls an intersection consists of 1) one pair of redundant intersection controllers, 2) eight sensor nodes, and 3) four traffic light actuator nodes, one per traffic light. The sensors broadcast periodically their sensor information. During normal operation all controllers get the same sensor broadcasts in the same order. We assume that the controllers are implemented by a deterministic algorithm. Hence, the controllers of an intersection implicitly agree on the commands to send to the traffic lights. Since the controllers get also the sensor information from the other intersection when the system is not partitioned, they can synchronize the traffic lights of the two intersections without any further communication.

When a sensor S becomes partitioned from a controller C or the sensor broadcasts of S are not delivered to C in a timely manner, then C and S cannot be in the same logical partition. To guarantee the fairness condition (at least one of the two sensors of each sensor pair has to be operational), it is sufficient that when the membership of the logical partition of a controller C does not contain at least one sensor for each direction of C's intersection, C switches to a "round robin" mode. To avoid that the two controllers of an intersection send conflicting commands to the traffic light actuators, a controller only broadcasts commands to the traffic lights when at least all four traffic light actuators of its intersection are in the membership of its logical partition: when both controllers have all traffic light actuators in their membership then both controllers are in the same logical partition (because at any point in real-time a process is in at most one logical partition) and hence, both controllers receive the same sensor broadcasts in the same order and therefore, implicitly agree on the commands they send to the traffic light actuators.

The traffic light actuator has to switch to a safe mode whenever it cannot be guaranteed that the four traffic light actuators receive the same sequence of commands in a timely manner. Since the broadcast indicator of a broadcast server p signals whenever p has not delivered all broadcasts in a timely manner, by monitoring its broadcast indicator a traffic actuator node can determine when it has missed a broadcast from a controller unit and it has to switch to a fail-safe mode. To detect when one or more of the other three traffic light actuators of an intersection do not get all broadcasts sent by the controllers, a traffic light actuator can simply check that all four actuators are in its current membership because a process that does not get all broadcasts will be removed from the membership from a logical partition within a bounded amount of time. Therefore, it is sufficient that each traffic light node p has a high priority process that switches that traffic light to a safe state when (1) p's broadcast indicator signals that it has not received all broadcasts, or (2) p's membership does not contain at least one of the two controllers and all four traffic light actuators of the intersection. Note that some simple hardware circuit that checks if the indicator of some server is up-to-date (see [8] for an example circuit) can in some cases replace the necessity of such high priority processes.

The switch to a safe state of the four traffic lights of an intersection has to be synchronized in the sense that (1) when one is flashing red the other lights have to show red or have to flash red, and (2) after switching to a fail-safe mode all lights have to flash red within a bounded amount of time. Let us now describe how the time locking mechanism can be used to coordinate the switch to a safe state even when all four traffic lights are partitioned from each other. For simplicity of exposition, let us assume that there are only two lights $L1$ and $L2$. To achieve an coordinated switch, $L1$ and $L2$ send each other periodic fail-aware unicast messages during their normal operation which lets the other traffic light know that the sender will not switch to the "flash state" for at least lt time units. $L1$ can switch to flash state whenever it has not sent such a message for at least lt time units (see Figure 12). When $L2$ does not receive at least every lt time units a message from $L1$, it first switches to red and when it is sure that $L1$ shows red or flashes red, it transitions to flash red. $L1$ knows that at least lt time units after its last message to $L2$ that $L2$ has to switch to red or flash state. Note that $L2$ has to stay in the red state for *at most* lt time units before it can switch to the flash state.

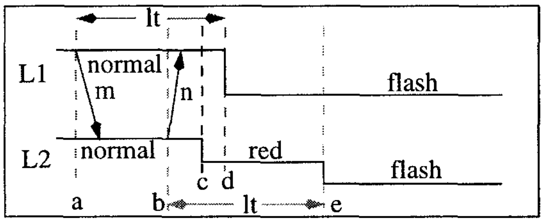

Figure 12: The two lights coordinate their switch to the flash state using a locking mechanism. $L1$ can switch to the flash state at d because it knows that $L2$ cannot assume after $c < d$ that $L1$ is in the normal state. $L2$ cannot switch to the flash state until time e since it has sent n ensuring that it will not switch to the flash state before e.

7 Related Work

Much of the research in distributed real-time systems has focused on the simpler, guaranteed response paradigm. Some systems designed according to that paradigm are Mars [13], XPA [20], TTP [12], and the Advanced Automation System family [1]. Recently, some of the research has focused on adaptive real-time systems. Research on detecting performance failures in 'quasi-synchronous' systems is described by Almeida and Verissimo [19]. Their approach depends on the existence of a lower level synchronous communication channel to detect such failures. In contrast, our approach does not require such a basic channel, but uses unsynchronized local clocks with bounded drift rates to detect performance failures. There also exist at least two systems that support the construction of partitionable fault-tolerant distributed applications on top of a network of workstations and that aim to provide real-time support by providing high throughput and predictable latency: Transis [3] and Totem [15]. In contrast to these systems, our approach aims to support the design of real-time systems by simplifying the detection of situations when the delays of messages and processes become that high that not all performance failures can be masked and an application has to switch to a fail-safe mode.

We introduced the concept of fail-awareness in [6] as a general method of transforming synchronous service specifications into weaker, fail-aware service specifications that are implementable in timed asynchronous systems [2]. In our earlier work on fail-awareness [6] we did not address the issue of partitionable operation: only servers in the partition that contains a majority of processes were allowed to make progress, i.e. at most the servers in one partition can provide their standard semantics. The main contribution of this paper is to show how fail-awareness can be extended such that servers in multiple partitions can make progress (and hence, can increase the availability of the system) by introducing the concept of logical partitions.

8 Conclusion

The guaranteed response paradigm [13], which always ensures timely responses, depends on the use of synchronous services, which in turn depend in a basic way on the assumption that the maximum number of failures per time unit is known. If this failure assumption, fundamental to all synchronous service implementations, can be violated at run-time, these implementations can be subject to unpredictable behavior. Much of the off-the-shelf hardware and software makes it very hard to guarantee a failure assumption at run-time. To address the current trend towards using off-the-shelf components in system design, *fail-awareness* no longer depends on a failure assumption: as long as the number of failures per time unit stays bounded, a fail-aware service provides its standard synchronous semantics and when too many failures occur per time unit the service performs a timely switch to its specified exception semantics. A description of our implementations of the fail-aware services we have introduced in this paper and their performance can be found in [8]. A detailed description of some of the fail-aware services are given in [7, 9, 5, 4]. A more detailed rail-way crossing example is presented in [8] and it shows how an application can use application level redundancy to mask a bounded number of performance failures per time unit and how the system can be switched to a safe state using a simple hardware circuit when the amount of redundancy is exceeded due to the occurrence of too many failures. (All these reports are available via our home pages.)

References

[1] F. Cristian, B. Dancey, and J. Dehn. Fault-tolerance in air traffic control systems. *ACM Transactions on Computers*, 14(3):265–286, Aug 1996.

[2] F. Cristian and C. Fetzer. The timed asynchronous system model. Technical Report CS97-519, UCSD, Jan 1997.

[3] D. Dolev and D. Malki. The transis approach to high availability cluster communication. *Communications of the ACM*, 39(4):64–70, Apr 1996.

[4] C. Fetzer and F. Cristian. Derivation of fail-aware membership service specifications. Technical Report CS96-502, UCSD, Nov 1996.

[5] C. Fetzer and F. Cristian. A fail-aware membership service. Technical Report CS96-503, UCSD, Nov 1996.

[6] 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.

[7] 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.

[8] C. Fetzer and F. Cristian. Fortress: A system to support fail-aware real-time applications. Technical Report CS97-520, UCSD, Jan 1997.

[9] 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.

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

[11] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger. Distributed fault-tolerant real-time systems: The Mars approach. *IEEE Micro*, pages 25–40, Feb 1989.

[12] H. Kopetz and G. Grunsteidl. Ttp-a protocol for fault-tolerant real-time systems. *IEEE Computer*, pages 14–23, Jan 1994.

[13] H. Kopetz and P. Verissimo. Real time and dependability concepts. In S. Mullender, editor, *Distributed Systems, Second Edition*, chapter 16, pages 411–446. Addison-Wesley, New York, 1993.

[14] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of ACM*, 21(7):558–565, Jul 1978.

[15] L. Moser, P. Melliar-Smith, D. Agarwal, R. Budhia, and C. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4):54–63, Apr 1996.

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

[17] D. Stewart and P. Khosla. Mechanisms for detecting and handling timing errors. *Communications of the ACM*, 40(1):87–93, Jan. 1997.

[18] P. Verissimo. Real-time communication. In S. Mullender, editor, *Distributed Systems, Second Edition*, chapter 17, pages 447–490. Addison-Wesley, New York, 1993.

[19] P. Verissimo and C. Almeida. Quasi-synchronism: a step away from the traditional fault-tolerant real-time system models. *IEEE TCOS Bulletin*, 7(4), Dec 1995.

[20] P. Verissimo, P. Bond, A. Hilborne, L. Rodrigues, and D. Seaton. The extra performance architecture (xpa). In D. Powell, editor, *Delta-4 – A Generic Architecture for Dependable Distributed Computing*. Springer Verlag, Berlin, 1991.