

# Fail-Awareness: An Approach to Construct Fail-Safe Systems

Christof FETZER and Flaviu CRISTIAN\*

AT&T Labs Research, 180 Park Ave, Florham-Park NJ07932, USA<sup>†</sup>

## Abstract

We present a framework for building fail-safe hard real-time applications in timed asynchronous distributed systems subject to communication partitions and performance, omission, and crash failures. Most distributed systems built from commercial-off-the-shelf (COTS) processor and communication services are subject to such partitions because their COTS components do not provide hard real-time guarantees. Also custom designed systems can be subject to partitions due to unmaskable link or router failures. 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 one 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 standard properties because of excessive delays. This allows an application to be *fail-aware*, i.e., to detect when it cannot guarantee all its safety properties and in particular, to detect when to switch to a fail-safe mode.

**Keywords:** fail-safe systems, fail-awareness, timed asynchronous systems, synchronous systems.

## 1 Introduction

In recent years there has been a trend towards the use of commercial-off-the-shelf (COTS) components such as real-time Unix and main-stream hardware platforms to build distributed hard real-time systems. A system is *hard real-time* if the consequence of a non-masked performance failure can be catastrophic [18]. Hard real-time systems can coarsely be classified into *fail-safe* and *fail-operational* systems. A typical real-time system consists of a controlled object and a controlling computing system. A *safe state* is a state of the controlled object in which no human life is at risk, e.g., a railway

crossing is in a safe state if the crossing arms are closed and it can stay in this safe state even when the controlling computer system has stopped working. In a *fail-safe system*, the controlled object has to transit to a safe state when a non-maskable component failure in the controlling computing system occurs and the controlled object has to be able to stay in this safe state without the help of the computing system. In the above rail-way crossing example, the crossing arms have to close automatically if the controlling computing system suffers an unmaskable failure.

In a *fail-operational system*, the controlled object does not have a safe state in which the controlled object can stay without the help of the controlling computing system. For example, a fly-by-wire system without a (mechanical) backup system does not have a safe state. However, a fly-by-wire system with a mechanical backup system can be viewed as a fail-safe system: the controlled object includes the mechanical backup systems which allows it to stay in a safe state even when the controlling computing system has failed. The mechanical backup system can be viewed as a fail-operational system. Note that the backup system does not have to be mechanical, it could also consist of a fail-operational computing system. Many complex systems can in this way be composed into a set of fail-operational and fail-safe sub-systems. In this paper we propose an approach of how to construct fail-safe (sub-)systems. However, one can use our approach to design fail-operational systems in case the underlying communication and process management service provide stronger properties (i.e., if they are synchronous).

### 1.1 Problem: Unknown Failure Frequency

The motivation for using COTS to build hard real-time systems 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 [16, 25, 17, 4] rely on the *guaranteed response paradigm* [18]. This paradigm depends on the assumption that the maximum number of failures per time unit, that is, the maximum failure frequency, is a priori known. Given this *failure assumption* is true, one can guarantee by design that the real-time system responds to events occurring in the controlled object within an a priori known time bound. Basically, the system has to have a sufficient degree of redundancy to be able to mask all failures as long as the maximum failure frequency is not exceeded. However, if this failure assumption can be vio-

\*Flaviu Cristian died in April 1999 after a long and courageous battle against cancer.

<sup>†</sup>This paper appears in the Journal of Real-Time Systems, March 2003, pp. 203-238, Kluwer Academic Publishers. This research was performed at the University of California, San Diego. There it was supported by grants F49620-93 and F49620-96 from the Air Force Office of Scientific Research. An earlier version of this paper appeared in the Proceedings of the 27th Annual International Symposium on Fault-Tolerant Computing, June 25-27, 1997, Seattle, Washington, USA.

lated at run-time (i.e., the maximum failure frequency can be exceeded), the real-time system can be subject to unpredictable behavior. Even in custom designed systems, it is extremely hard to determine a maximum failure frequency. Since there is always a non zero probability that the failure frequency in a system is higher than the assumed maximum frequency, it makes sense not to assume that there exists such a maximum frequency. In this paper we show how one can cope with any failure frequency (as long as the types of failures are known). In particular, we describe how to switch a system into a safe mode in case the failure frequency becomes too high.

One major problem in determining a maximum failure frequency is the occurrence of performance failures. A *performance failure* [3] occurs when a service responds after an a priori given maximum time quantum has expired, e.g., a message  $m$  suffers a performance failure if  $m$  is delivered after more than, say,  $\delta$  time units. To bound the frequency of performance failures, 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 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 [22]. 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.

## 1.2 Approach: Fail-Awareness

We present in this paper the *fail-awareness paradigm*. This paradigm emphasizes the detection of failures to address the problem of the violation of safety properties caused by non-maskable failures. A safety property is a property that has to hold all the time. Note that a safety property does not necessarily state whether the system is *safe* (i.e., no human life is at risk). However, some safety properties might state when the system is safe. For example, in a railway crossing, the system is safe if the following (informal) safety property (ST) holds: at any time  $t$ , if there is a train in the railway crossing at  $t$ , the crossing arms are down at  $t$ . Performance failures might not only delay the execution of a program, they might also invalidate safety properties. For example, the delay of a message  $m$  requesting to lower the crossing arms of a railway crossing might result in the invalidation of property (ST) unless one can mask or detect the failure of  $m$ . A violation of a safety property might also result in a *contamination problem*: late message deliveries and slow processes might corrupt the state of processes and processes with corrupted state might send messages that corrupt the state of the receiver processes. Fail-awareness addresses the detection and containment of failures such that processes are aware when certain safety properties are violated. This detection allows the avoidance of state contamination.

We show in this paper that fail-awareness can be used on differ-

ent levels of abstraction. We assume that a system can be described as a set of services. Each service is typically specified by a set of safety properties. Fail-awareness allows a higher level service  $S$  to detect what safety properties of lower level services are violated and in turn allows the service  $S$  to indicate to its clients which of its safety properties might be violated. On the highest layer, it allows a fail-safe system to detect when to switch the controlled object into a fail-safe state. In this paper we concentrate on the design of fail-safe systems. Nevertheless, the fail-awareness paradigm can also be advantageous for designing fail-operational systems and at the interface between a fail-operational and a fail-safe sub-system: 1) fail-awareness supports the detection of when a fail-operation sub-system has to switch to a degraded mode, 2) it allows a fail-operation sub-system to detect when it cannot depend on the services provided by a fail-safe sub-system, and 3) if the failure frequency can be bounded in a subsystem, the fail-aware services in this sub-system are guaranteed to be fail-operational (i.e., no safety property is ever violated).

Due to the problems posed by the guaranteed response paradigm, many practical systems are based on the *best effort paradigm* which emphasizes low latencies but 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 in this paper the *fail-awareness paradigm* which can be viewed as a middle of the road approach between the “guaranteed response” and the “best effort” paradigms. A system designed according to the fail-awareness paradigm can provide the same properties like a system designed according to the guaranteed response paradigm as long as the failure frequency does not exceed some given bound. When the failure frequency rises above that bound, the services of the system provide their properties on a best effort basis, i.e., try to keep its properties valid as long as possible but there is no guarantee that a service actually succeeds to maintain all its properties. In contrast to a best-effort and a guaranteed response service, a fail-aware service allows its clients to determine which of its properties are currently valid and which properties might currently be violated. We explain later how this can be achieved by defining expiration times for properties and a client can determine if a safety property might be violated since its expiration time has already passed.

When the failure frequency drops below the given bound again, all fail-aware services will provide all their properties and also the clients learn that all properties are valid. The main feature of a fail-aware service is that it and its clients can determine at any point in time what properties are currently valid and what properties might be violated. Clients of the service can use that knowledge to determine if they can currently depend upon that service. Hence, fail-awareness allows to contain performance failures and to guard against contamination problems.

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 [7] and Totem [20]), we specify the standard hard real-time semantics of services using

real-time deadlines and provide mechanisms to detect when an application cannot depend upon the hard real-time properties of lower level services due to unmasked performance, omission, or crash failures. This detection is essential for fail-safe applications that have to switch to a fail-safe service mode whenever they cannot guarantee their hard real-time properties.

### 1.3 Partitionable Fail-Aware Services

The fail-awareness paradigm supports the design of partitionable fail-aware services to be able to increase the service availability. 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*. Our fail-awareness paradigm allows the servers in each partition to “make progress” independently of the servers in other partitions, i.e., these servers can maintain all their safety properties and hence, increase the availability of the system.

Clients can learn whether a safety property of a server currently holds by examining an *exception indicator* provided by the server. The implementation of an exception indicator that we propose uses the fact that a server has typically to update its “output” periodically, e.g., it has to update its output every  $R$  clock time units. When a server cannot update its output within  $R$  time units, it knows that it cannot provide certain properties anymore.

The indicators are also used to identify the partition of a server. We call a partition that is identified by a fail-aware service a “logical partition”. Later on, we will define the notion of a “stable partition” that identifies the communication partitions in which the failure frequency is bounded and in which we require the servers to make progress (i.e., these servers have to make sure and know that all their safety properties are valid). The fail-aware services identify at least all stable partitions as logical partitions, give these partition a name. A logical partition can actually be viewed as a synchronous sub-system, i.e., processes in the same logical partition can provide the same properties as the processes of a synchronous system. When a server provides its safety properties, its indicator(s) signal its clients to what logical partition it belongs. Otherwise, an indicator signals that certain safety properties of the server might be violated.

## 2 System and Failure Model

The guaranteed response paradigm assumes that (1) the classes of likely failures, and (2) their maximum failure frequency, i.e., the maximum number of failures per time unit, are a priori known. Using a sufficient amount of redundancy one can then mask all failures that are likely to occur and one can therefore guarantee that a system meets all its deadlines (as long as the two assumptions are valid). The fail-awareness paradigm also

assumes knowledge of the classes of likely failures, but does not assume any knowledge about their maximum frequency of occurrence (since in many practical system the number of performance failures per time unit cannot be bounded due to the use of COTS components). 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 [2]) and the fail-awareness paradigm (i.e., use of the timed asynchronous system model [5]). We review in this section the basic differences between these two models.

### 2.1 Synchronous Systems

In a *completely synchronous system* the real-time delays of all processes and all messages are within a priori known bounds and there is always a minimum number of processes that are correct. 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 ( $\sigma$ ) and message transmission delays ( $\delta$ ). When the transmission delay of a message  $m$  is greater than the maximum assumed message delay  $\delta$ , one says that  $m$  suffers a performance failure. Otherwise,  $m$  is said to be *timely*. The *scheduling delay* of a process is the time between a process is scheduled to execute some procedure in response to an event occurrence and the time the execution actually starts. When the scheduling delay of a process  $p$  is greater than the maximum assumed scheduling delay  $\sigma$ ,  $p$  suffers a performance failure. A process that does not suffer any performance failures in a given time interval is said to be *timely*.

A *failure model* specifies what kind of failures are *likely*: these kind of failures have to be considered in the design of a system and the probability that any other kind of failure occurs is negligible given the overall stochastic requirements for the system [3]. 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 [2]:

- a process can take more than  $\sigma$  to react to an event (performance failure), or a process can crash, but one can neglect the probability that a process suffers any other type of failure, and
- messages can be dropped (omission failure) or can arrive after  $\delta$  time units (performance failure), but one can neglect the probability that a message suffers any other type of failure.

A *failure assumption* specifies an upper bound on the maximum frequency of (performance, crash, and omission) failures. Thus, the failure assumption states that the probability that the failure frequency is higher than that bound is negligible. Since most distributed protocols are “round based” protocols, a failure assumption often states the maximum number of failures per round instead of specifying an explicit failure frequency.

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 failure assumption are correct, one can exclude the occurrence of non-maskable failures, and hence, 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 [21].

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. By “reasonable” we mean the probability that the failure assumption is violated (i.e., the experienced failure frequency is higher than the assumed upper bound) is negligible. We use the *timed asynchronous distributed system model* as the foundation of our work because it does not put any bound on the number of failures per time unit. For a detailed, formal description and comparison with other models like the quasi-synchronous model of [24] see [5].

## 2.2 Timed Asynchronous Systems

The timed asynchronous system model [5] assumes that processes have access to a local *unsynchronized* hardware clock with a bounded drift rate, i.e., the clock proceeds 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 assume any bound on the maximum failure frequency, i.e., it has no failure assumption. It is an accurate description of existing distributed systems like a network of workstations running Solaris, Linux 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 the synchronous system model, the timed asynchronous model does not necessarily allow one 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 failure frequency is within some given bound. However, the fail-aware protocols have to deal with situations when processes become partitioned, partitions merge, or the failure frequency becomes too high.

## 3 Fail-Awareness

We assume that a distributed service is specified by a set of safety properties, i.e., properties that have to hold all the time. In this paper, we only consider services that are implementable in completely synchronous systems, i.e., system characterized by a bounded failure frequency. Note that any service that is implementable in an asynchronous or synchronous system is implementable in a completely synchronous system since the completely synchronous system model provides stronger properties than any other asynchronous or synchronous distributed system model. However, a service that is implementable in a completely synchronous system is not necessarily implementable in a weaker model like the timed asynchronous system model. Fail-awareness allows one to transform a service specification that is implementable in a completely synchronous system such that it becomes implementable in timed asynchronous systems.

A fail-aware service is implemented by a set of fail-aware servers. Each fail-aware server maintains one or more (*exception*) *indicators* (see Figure 1). The indicators enable a client of the services to determine which properties of the service do currently hold and which might currently be violated (see Section 5.3 for how to implement indicators). For example, a time server could provide two indicators: one indicator signals if the local clock is currently synchronized with real-time while the second indicator shows if the clock is currently synchronized with the clocks of the other time servers.

When too many failures occur, a server cannot necessarily provide all its properties anymore. A fail-aware server signals such conditions by one or more indicators. To avoid trivial service implementations in which a server indicates all the time that it cannot maintain its properties, we introduce an integration requirement: whenever the failure frequency is not higher than some given threshold for some minimum time, none of the server’s indicators can signal an exception and hence, all properties are required to be valid.

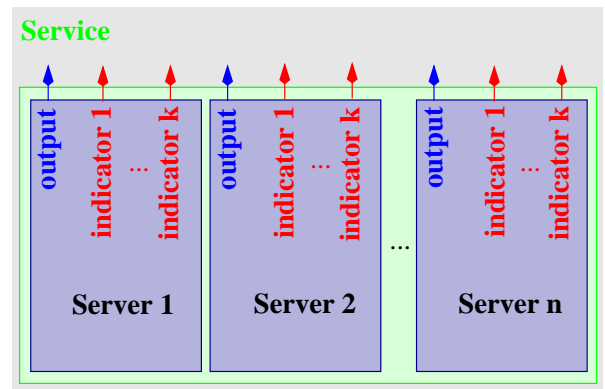


Figure 1: A fail-aware service is implemented by a set of fail-aware servers. Each fail-aware server maintains one or more indicators. An indicator enables a client to detect if certain properties might be violated.

### 3.1 Example: Internal Clock Synchronization

Let us give an example of how to transform a synchronous service specification into a “primary-partition” fail-aware service specification. By *primary-partition* we mean that only processes in at most one partition can make progress. In Section 3.3 we will continue this example to show how one can transform this primary-partition fail-aware service into a partitionable fail-aware clock synchronization service.

Consider the specification of an internal clock synchronization service for a synchronous system consisting of two safety properties: a bounded drift and a bounded deviation property. Let predicate  $crashed_p(t)$  be true if and only if server  $p$  is crashed at time  $t$ . The bounded drift property (S1) states that the drift rate of the clock  $C_p$  of a non-crashed server  $p$  stays bounded by an a priori known constant, say,  $\rho \ll 1$  as long as  $p$  does not crash:

$$(S1) \forall t, \forall s, s < t, \forall p : \neg crashed_p(s) \wedge \neg crashed_p(t) \\ \Rightarrow (t - s)(1 - \rho) \leq C_p(t) - C_p(s) \leq (t - s)(1 + \rho)$$

The bounded deviation property (S2) states that at any time  $t$  the deviation between two clocks  $C_p$  and  $C_q$  of two non-crashed servers  $p$  and  $q$  is bounded by a known constant, say,  $\mu$ :

$$(S2) \forall t, \forall p, \forall q : \neg crashed_p(t) \wedge \neg crashed_q(t) \\ \Rightarrow |C_p(t) - C_q(t)| \leq \mu$$

Internal clock synchronization (S1,S2) is implementable in synchronous systems but not in timed asynchronous systems. To implement its clock  $C_p$ , a server  $p$  can use its local hardware clock that has already a bounded drift rate. However, since two hardware clocks can drift apart from each other, the servers have to communicate to ensure property (S2). Since the processes can partition in a timed asynchronous system, servers might not be able to communicate for an arbitrary amount of time and hence, cannot necessarily keep their clocks synchronized within  $\mu$ .

In a fail-aware internal clock synchronization service, each clock  $C_p$  is associated with a Boolean indicator  $I_p$ . The informal meaning of  $I_p(t) = true$  is that server  $p$  provides the same properties as a non-crashed server in a synchronous system, i.e., its clock has a bounded drift rate and it is synchronized with all other clocks (unless a clock  $C_q$  signals that it is out-of-sync by setting  $I_q(t) = false$ ).

Safety properties (S1,S2) are replaced by the following properties (F1,F2) in which we replaced the term  $\neg crashed_p$  by indicator  $I_p$ :

$$(F1) \forall t, \forall s, s < t, \forall p : I_p(s) \wedge I_p(t) \\ \Rightarrow (t - s)(1 - \rho) \leq C_p(t) - C_p(s) \leq (t - s)(1 + \rho)$$

$$(F2) \forall t, \forall p, \forall q : I_p(t) \wedge I_q(t) \\ \Rightarrow |C_p(t) - C_q(t)| \leq \mu$$

Note that (F1,F2) are safety properties since they hold for any time  $t$ . However, they do permit that something bad can happen (i.e., the deviation of two clocks are more than  $\mu$ , or the drift rate of a clock is greater than  $\rho$ ) as long as the indicators say that something bad might happen (i.e.,  $I_p(t) = false$ ).

The integration requirement is defined as follows: given a constant  $RT$  that denotes the “maximum resynchronization time” of a clock, we require that:

$$(F3) \forall t, \forall p : \text{“failure frequency bounded in } [t - RT, t]\text{”} \\ \wedge \neg crashed_p(t) \Rightarrow I_p(t).$$

The indicator  $I_p$  of a crashed server  $p$  is assumed to be false. Since a synchronous server  $p$  has to synchronize its clock whenever a fail-aware server has to do that (because  $I_p(t) \Rightarrow \neg crashed_p(t)$ ), the new properties (F1,F2,F3) are strictly weaker than the original properties (S1,S2). In particular, properties (F1,F2,F3) are implementable in timed asynchronous systems because a server can reset its indicator when it fails to synchronize its clock<sup>1</sup>.

We weakened the specification (S1,S2) to make it implementable in timed asynchronous systems. We claim that the resulting specification (F1,F2,F3) is still sufficiently strong. Consider that a system consists of a hierarchy of fail-aware services (see Figure 2). A failure in a lower level server  $S$  (see server 2 in Figure 2) might result in the indication of a property violation of this server. Servers that depend upon  $S$  might still be able to maintain their properties without  $S$  (see servers 3 and 6) while other servers have to signal a property violation themselves (see server 4). On the highest level there might be servers that have to decide whether to switch the system to a safe mode in case the safety of the system cannot be guaranteed anymore (see Section 6). In summary, a violation of a service property is acceptable since higher level servers can either mask them or signal a property violation themselves.

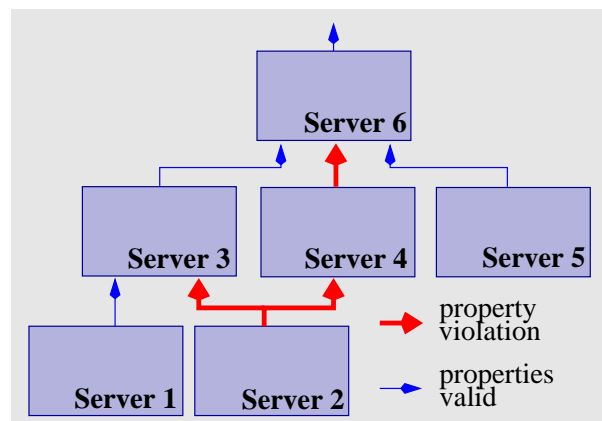


Figure 2: The violation of a safety property of a lower level server can result in the invalidation of properties of higher level servers. Higher level servers have to indicate property violations if they are not able to mask property violations of lower level servers.

<sup>1</sup>An implementation of (F1,F2,F3) that cannot change the speed of the hardware clocks needs access to a stable storage to guarantee (F1). If a process  $p$  crashes, it loses the information of how far the synchronized clock  $C_p$  is apart from the local hardware clock. This offset is needed to guarantee a bounded drift rate during reintegration of the clocks after a total system failure. Later we will transform (F1,F2,F3) for partitionable systems such that one does not need a stable storage.

To explain how a client can mask a violation of the bounded deviation property by using the indicator of a server, consider a leader election problem. The leader election problem is characterized by the property that there is at most one leader at a time. One can use an internal clock synchronization service (S1,S2) to define non-overlapping time slots during which exactly one process is the “dedicated leader” and a process can use its local clock to decide whether it is currently the leader. One can also use properties (F1,F2) to ensure that there exists at most one leader at a time. If a process  $q$  reads the clock  $C_p$  of server  $p$ , it reads at the same time the value of the indicator  $I_p$ . This enables  $q$  to determine if the clock  $C_p$  is synchronized. Process  $q$  is only leader in one of its time slot if clock  $C_p$  is synchronized. Since a process  $p$  can only become leader if its local clock  $C_p$  is synchronized and synchronized clocks are at most  $\mu$  apart from each other, one can guarantee by assigning processes non-overlapping time-slots (i.e., slots are more than  $\mu$  apart from each other) that there is at most one leader at a time.

Note that it makes sense that a service provides more than one indicator if a server  $S$  can maintain some property  $P_1$  independently of another property  $P_2$  (i.e., sometimes  $S$  might be able to provide  $P_1$  while it cannot maintain  $P_2$ , and vice versa). For example, consider that the fail-aware clock synchronization service also tries to synchronize the clocks externally, e.g., with GPS time. Sometimes the service might fail to achieve external clock synchronization, e.g., if the GPS receiver has failed. Since services like a leader election service do not dependent upon the property that a clock is externally synchronized, it make sense to introduce in this case a second indicator that signals whether a clock is externally synchronized.

### 3.2 Specification Transformation

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 show how to 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 failure frequency and the possibility of communication partitions.

The idea behind the support of partitionable fail-aware services is a follows. The properties of a synchronous service are defined for all non-crashed processes in the system. We want to transform these properties such that they are defined for all processes in the same communication partition. For example, instead of requiring that all clocks of non-crashed processes are synchronized with each other, we only require that the clocks in the same communication partition are synchronized with each other. There are two difficulties in restricting properties to a communication partition. First, we cannot require that properties be holding in communication partitions in which processes experience a too high failure frequency. Second, we have to be

able to express that two processes are in the same communication partition. We address the first issue by introducing the notion of a *stable partition*, that is, a communication partition in which the failure frequency is bounded. Only processes in stable partitions have to provide all their properties. We address the second issue by identifying at least all stable partitions and requiring that if a server is part of a stable partition, its indicators be showing the name of this partition. Hence, we can express that a property  $P$  has to hold for all processes in a communication partition by stating that  $P$  has to hold for all processes whose indicators show the same partition name.

The naming of a communication partition is typically done by one service in the system and all other services use the same names for partitions. We describe in [14] how one can name partitions. The basic idea is that one can elect a leader in each stable communication partition and if a process is elected as the local leader of a communication partition, it gives this communication partition a unique name. We call a communication partition that is named a *logical partition* [14]. Since all stable partitions are named, they are also logical partitions. However, there might exist logical partitions that are not stable partitions.

Let us assume that the specification  $S = \{SP_1, \dots, SP_k\}$  of a synchronous service consists of  $k$  safety properties and each safety property  $SP_i$  is of the form  $\forall t : P_i(t)$ . Hence, safety property  $SP_i$  says that property  $P_i$  has to hold for all times  $t$ . We show how one can transform the specification  $S$  into a specification  $FA$  of a fail-aware partitionable service implementable in timed asynchronous systems. The idea is that we introduce an indicator  $I$  and each server  $p$  maintains an instance  $I_p$  of the indicator. If the indicator  $I_p$  of a server  $p$  shows the name  $LP$  of a logical partition at time  $t$ , the clients of  $p$  can learn by querying  $I_p$  that  $p$  maintains all its properties and its properties are defined with respect to the processes in  $LP$ . As we mentioned before, to permit a better detection of what properties currently hold and what not, a service might provide more than one indicator. The transformation of  $S$  into  $FA$  using one indicator can be performed as follows.

- The interface of  $FA$  is augmented with an indicator  $I$ . Each server  $p$  of the service has to maintain an instance  $I_p$  of indicator  $I$ . We denote the value of  $p$ 's indicator  $I$  at real-time  $t$  by  $I_p(t)$ . To signal that a server cannot provide its properties, it sets its indicator to some value  $\perp$ .
- For each property  $SP_i = \forall P_i(t)$  in  $S$  we define a new property  $FP_i$ . In  $FP_i$  we introduce a new variable  $L$ , quantify  $L$  over all logical partitions ( $\forall L \neq \perp$ ), and replace terms of the form “ $crashed_p(t)$ ” by “ $I_p \neq L$ ”<sup>2</sup>. Informally, if  $SP_i$  states that a property has to hold for some non-crashed processes  $p_1, \dots, p_m$ ,  $FP_i$  states that this property has to hold for some processes  $p_1, \dots, p_m$  whose indicators show that they in the same logical par-

<sup>2</sup>One does not necessarily want to replace all  $crashed_p(t)$  terms. Typically, one wants to keep the original terms  $crashed_p(t)$  if 1) the resulting specification is still implementable in timed asynchronous systems and 2) it still permits processes in different partitions to make progress.

tion. If a server  $p$  cannot maintain some property  $P_j$ , it has to signal this by setting its indicator to  $\perp$ .

- We extend  $FA$  by an integration requirement that restricts the naming of stable partitions. 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 failure frequency within  $SP$  is not higher than some a priori given bound. We call  $SP$  a *stable partition*.

Note, we do not try to automate or formalize this transformation. This transformation is intended as a guideline for designers of distributed services. While the sketched transformation steps have worked for all synchronous services we have considered so far, these steps do not apply to specifications that are defined without the use of predicate *crashed* (or, some “equivalent” predicate like “*correct*”). Such specifications can still be translated into a fail-aware specification. However, a designer has to extend “manually” at least all properties by an indicator that are not implementable in timed systems. This requires the skill to determine what is possible to implement in a timed system and what is not. We have no rule how to determine this.

Due to uncertainties in a distributed system (e.g., a process cannot measure the exact transmission delay of a message), a process cannot always correctly decide whether it is member of a stable partition. However, a server does not actually have to decide whether it is in a stable partition since

- a server has to guarantee by design (i.e., by being able to mask a certain number of failures) that it provides all its properties as long as it is part of a stable partition,
- it has to detect when it cannot mask all failures and hence, cannot provide all its properties anymore (i.e., by design this can only happen when it is not part of a stable partition), and
- 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 describes our implementation of an exception indicator: an indicator has to enable clients to detect that a server cannot provide all its properties even when the server suffers performance failures. This is necessary to permit the watchdog processes to monitor the safety of the system (and switch the system to a fail-safe mode if needed). They can check at any point in time if some properties of a server might be violated.

### 3.3 Example: Partitionable Fail-Aware Clock Synchronization

We show in this section how the specification (F1,F2,F3) of a primary partition fail-aware internal clock synchronization

service (see Section 3.1) can be transformed into a partitionable fail-aware service specification (P1,P2,P3). In the primary partition specification, any two clocks that are synchronized, are at most  $\mu$  apart from each other. In the new specification (P1,P2,P3), the deviation between two clocks in the same partition has to be at most  $\mu$ . However, the deviation between clocks in two separate partitions is not bounded. To know which processes are in the same partition and which are in different partitions, the indicator of a server shows the name of the partition, if a server can keep its clock synchronized. Otherwise, the indicator shows a special value  $\perp$  to indicate that its clock is out of sync. We have to make sure that if two processes are in the same stable partition, they name the partition in the same way to make sure that they have to synchronize their clocks.

The new integration property states that if two processes are in the same stable partition for at least  $RT$  time units, then their indicators say that they are in the same logical partition, i.e.,

$$(P3) \forall t, \forall s \leq t - RT, \forall p, \forall q : \\ \text{“}p \text{ and } q \text{ are in the same stable partition in } [s, t]\text{”} \\ \Rightarrow I_p(t) = I_q(t) \neq \perp.$$

The new bounded drift condition (P1) states that the drift rate of a server has to be bounded by  $\rho$  between any two times for which the process is in the same partition, i.e., the value of its indicator is the same,

$$(P1) \forall t, \forall L, L \neq \perp, \forall s, s < t, \forall p : I_p(s) = L \wedge I_p(t) = L \\ \Rightarrow (t - s)(1 - \rho) \leq C_p(t) - C_p(s) \leq (t - s)(1 + \rho)$$

The deviation between the clocks of two processes in the same logical partition is at most  $\mu$ ,

$$(P2) \forall t, \forall L, L \neq \perp, \forall p, \forall q : I_p(t) = L \wedge I_q(t) = L \\ \Rightarrow |C_p(t) - C_q(t)| \leq \mu$$

### 3.4 Stable Partitions

To support partitionable operations, we do not require the existence of a global maximum acceptable failure frequency. Instead we distinguish those parts of a system that have a maximum acceptable failure frequency as *stable partitions*. Informally, a set of processes  $SP$  forms a stable partition when

- all processes in  $SP$  are timely (i.e., they do not miss any deadlines),
- all but a bounded number of messages sent between these processes per “round” are delivered in a timely manner, and
- from any other partition either no or only “old” messages arrive.

Formally, we define a stable partition by a *stability predicate*. For concreteness, let us introduce the stability predicate  $\Delta$ -1-partition that we use in the described fail-aware service implementations. To this end, we first define the concepts of 1-connected processes and of  $\Delta$ -disconnected processes, which generalize the notions of connectedness and disconnectedness introduced in [6].

Two processes are  $1$ -connected in  $[s, t]$  iff (1)  $p$  and  $q$  are timely in  $[s, t]$ , and (2) all but at most one message sent between the two processes in  $[s, t]$  are delivered in a timely manner, i.e., within at most  $\delta$  time units (see Figure 3). We denote the fact that  $p$  and  $q$  are 1-connected in  $[s, t]$  by the predicate  $1\text{-connected}(p, q, s, t)$ .

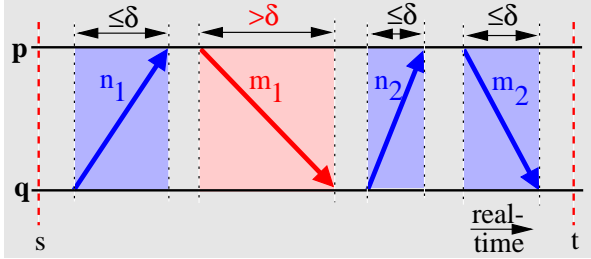


Figure 3: Two timely processes  $p$  and  $q$  are 1-connected in  $[s, t]$  iff at most one message sent between  $p$  and  $q$  in  $[s, t]$  suffers an omission/performance failure and all other messages are delivered within  $\delta$  time units.

A process  $p$  is *disconnected* from a process  $q$  in a time interval  $I$  iff  $p$  does not receive any message from  $q$  during  $I$  [6]. We generalize the notion of disconnectedness by allowing situations in which  $p$  receives old messages from  $q$  during  $I$ . These old messages typically contain out-of-date information and thus, have to be rejected by  $p$ . In Section 4 we give an overview of a fail-aware datagram service that classifies all messages with a transmission delay greater than some  $\Delta > \delta$  as “slow” and messages with a transmission delay of at most  $\delta$  as “fast”. The second constant  $\Delta$  was introduced because a receiver of a message  $m$  can only determine the transmission delay of  $m$  with some error [9]. The following definition of “ $\Delta$ -disconnected” is based on the fact that processes can identify “slow” messages: a process  $p$  is  $\Delta$ -disconnected from a process  $q$  in a time interval  $[s, t]$  iff every message  $m$  that  $p$  receives in  $[s, t]$  from  $q$  has a transmission delay of more than  $\Delta > \delta$  time units (see Figure 4). Common situations in which two processes are  $\Delta$ -disconnected are when the network between them is overloaded, or an intermediate router or link is down. We use the predicate  $\Delta\text{-disconnected}(p, q, s, t)$  to denote that  $p$  is  $\Delta$ -disconnected from  $q$  in  $[s, t]$ .

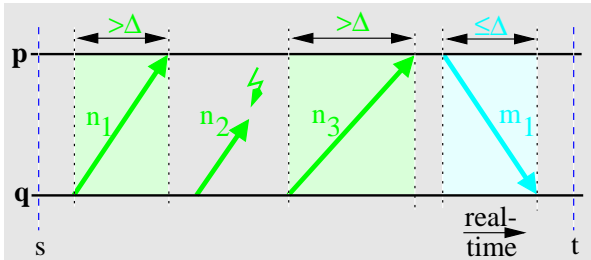


Figure 4: Process  $p$  is  $\Delta$ -disconnected from  $q$  in  $[s, t]$  when all messages that  $p$  receives from  $q$  in  $[s, t]$  have a transmission delay of more than  $\Delta$ . Note that  $q$  might receive messages from  $p$  with a delay of less than  $\Delta$ .

Let  $\mathcal{P}$  be the set of all processes. We say that a non-empty set of processes  $S$  is a  $\Delta$ -1-partition in an interval  $[s, t]$  iff all processes in  $S$  are 1-connected in  $[s, t]$  and the processes in  $S$  are  $\Delta$ -disconnected from all other processes (see Figure 5):

$$\begin{aligned} \Delta\text{-1-partition}(S, s, t) \triangleq & \\ & \wedge S \neq \emptyset \\ & \wedge \forall p, q \in S : 1\text{-connected}(p, q, s, t) \\ & \wedge \forall p \in S, \forall r \in \mathcal{P} - S : \Delta\text{-disconnected}(p, r, s, t). \end{aligned}$$

We say that  $S$  is a *stable partition* (in  $[s, t]$ ) iff predicate  $\Delta$ -1-partition( $S, s, t$ ) holds. We use the predicate  $\Delta$ -1-partition in the specifications of round based protocols such that the interval  $[s, t]$  encompasses one round. This typically implies that these protocols have to mask at least one message failure per round and process pair. The extension of the definitions to  $F$ -connected and  $\Delta$ - $F$ -partition (i.e., up to  $F \in \{0, 1, \dots\}$  messages per process pair and round can suffer a failure) is straightforward.

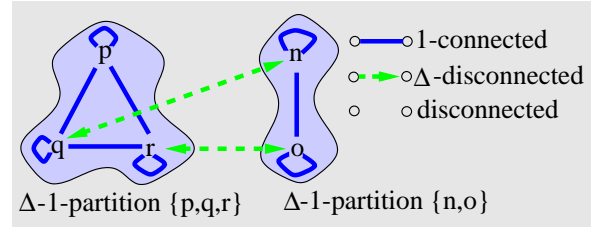


Figure 5: All processes in a  $\Delta$ -1-partition are 1-connected and all messages from outside the partition have a transmission delay of more than  $\Delta$ . Note that the  $\Delta$ -disconnected relation is actually symmetric for any two processes (like  $q$  and  $n$ ) that are in two different  $\Delta$ -1-partitions.

## 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 6) [11]. 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.

### 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. Since processes and messages do not suffer performance failures, this greatly simplifies the protocol design. In practice, systems are often not completely synchronous, in particular, we are considering systems in which the probability that partitions

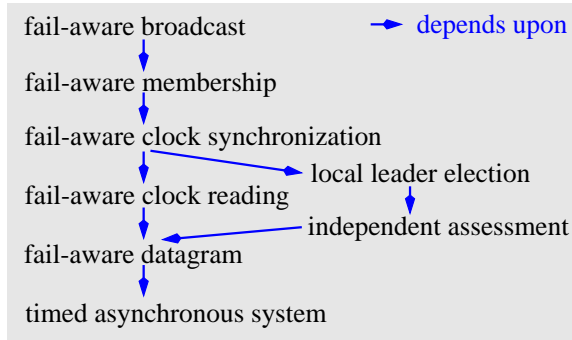


Figure 6: Hierarchy of fail-aware services to support the design of fail-safe partitionable real-time applications.

occur is not negligible. When a system splits into partitions, the ideal situation (with respect to simplifying the design of protocols) would be that each partition shows completely synchronous behavior (see Figure 7): 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. 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.

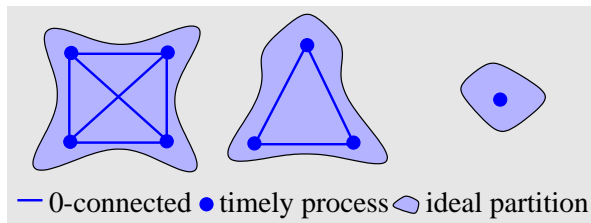


Figure 7: Ideally all processes in a partition should be mutually 0-connected with all processes in the same partition and be disconnected from all processes in other partitions.

Real communication partitions do not behave in the ideal manner described above. 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 8). 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 [5] for some  $F > 0$ , i.e.,  $p$  and  $q$  are timely and up to  $F$  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 8).

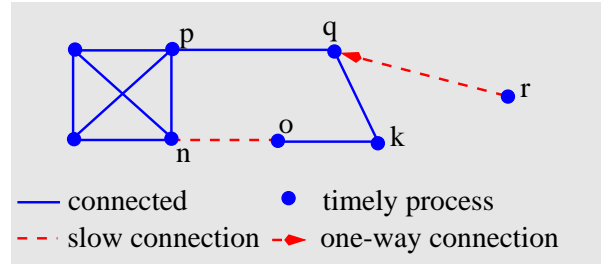


Figure 8: Real partitions are not always ‘ideal’: slow connections, one-way connections, and non transitivity of the connected-relation complicate the protocol design.

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 signal an exception by setting its indicator to  $\perp$ . To make logical partitions similar to ideal partitions, we require that logical partitions never overlap, 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:

- *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 8). 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) we use a limited amount of forwarding at the level of the broadcast service, and (2) we logically disconnect (when necessary) some processes even though they are capable of communicating with each other.

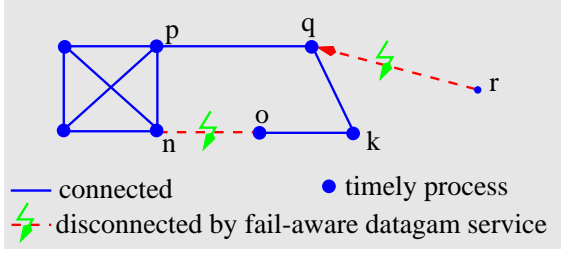


Figure 9: The fail-aware datagram service allows a receiver process to reject messages that arrive via one-way or slow links.

## 4.2 Service Hierarchy

The foundation of our fail-aware protocol stack is the *fail-aware datagram service* [13]. Its purpose is to reject messages that arrive via slow or one-way connections (see Figure 9). 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. The calculated upper bound for a timely message (whose transmission delay is  $\leq \delta$ ) is at most some known  $\Delta$  ( $\geq \delta$ ), i.e., the maximum error of the upper bound for a timely message is at most  $\Delta - \delta$ . The upper bound calculated for a message sent via a one-way or a slow connection is greater than  $\Delta$ . Higher level services, such as local leader election, reject messages with a calculated upper bound greater than  $\Delta$  time units.

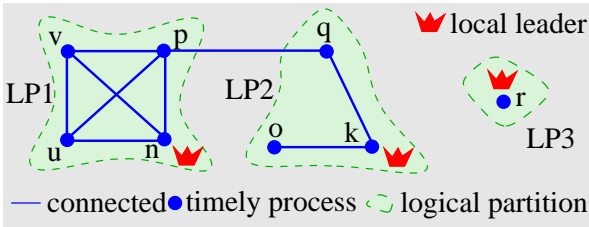


Figure 10: The local leader election service elects local leaders and creates non-overlapping logical partitions, each consisting of ‘supporters’ of a local leader.

The next step is to group processes into logical partitions. This is the goal of the *local leader election* service [14]. It has to create logical partitions such that all processes in some set  $S$  that are mutually connected (i.e., each two processes in  $S$  are connected) are in the same logical partition. For example, all processes in  $LP1$  of Figure 10 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 [14] for details): a process  $r$  supports the election of the process  $l$  only if  $l$ 's identification is smaller than the identifier (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 10 the protocol elects processes  $n$ ,  $k$ , and  $r$  as local leaders (assuming the following order  $n < o < \dots < u < v$ ). 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 5.4): a process  $p$  stays in a logical partition  $LP$  for only a bounded amount of time 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 [14]. Since the set of processes of 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 within the same logical partition is bounded by some a priori given constant  $\Psi$ . The membership service [10] 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.

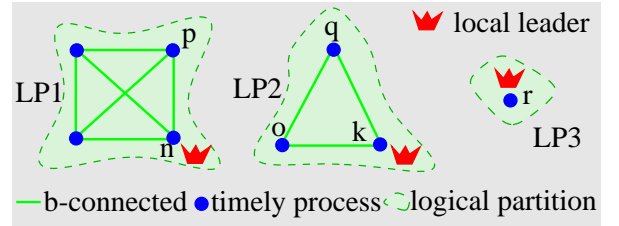


Figure 11: The broadcast service connects all processes in a logical partition and disconnects them from the processes in all other partitions.

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 11).

## 4.3 Fail-Awareness Properties

All fail-aware services of our protocol stack provide properties that are similar to that of the corresponding synchronous services. We will only sketch a few properties of some of the fail-aware service (for a detailed description of their semantics and implementations see [12, 10, 14, 13]).

The *fail-aware membership* service takes the logical partition  $LP$  created by the local leadership service [14] and maintains agreement on the membership of  $LP$  among the members of  $LP$  (see [12] 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

- 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
- whenever a server cannot keep up with the other servers in  $LP$  (i.e., its indicator shows  $\perp$ , it is crashed or slow, 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 (only allowed when  $q$  is not part of a stable partition) 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 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 [10] 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 5.4) to guarantee that the memberships of two logical partitions do not overlap.

The *fail-aware atomic broadcast* service delivers messages within a constant  $\Omega$  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 [19] (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  $\Delta$ . 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 (or, "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 need to detect these failures so that they can switch to their safe state. We review some of the mechanisms we use to detect performance failures and to implement indicators.

### 5.1 Fail-Aware Datagrams

The fail-aware datagram service [13] calculates upper bounds on the transmission delays of one-way messages by using round trip delay measurements [1]. For example, to compute an upper bound on the transmission of  $m$ , it uses the four time-stamps

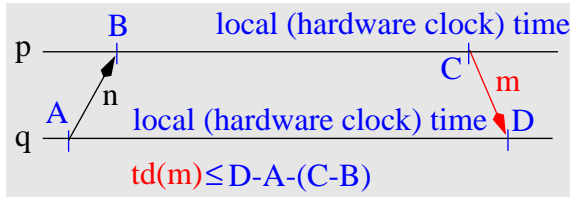


Figure 12: When the drift rate of hardware clocks can be neglected, the transmission delay of  $m$  can be bounded by  $(D - A) - (C - B)$  even though  $p$ 's and  $q$ 's hardware clocks are not synchronized.

(each taken with local unsynchronized hardware clock) of the round-trip  $n, m$  (see Figure 12). The service makes sure that two connected processes  $p$  and  $q$  exchange periodically messages so 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 (of the order of  $10^{-6}$ ), even when  $p$  has received  $n$  several seconds before sending  $m$ , the increase of the upper bound for  $m$  due to the maximum hardware clock drift rate is very small.

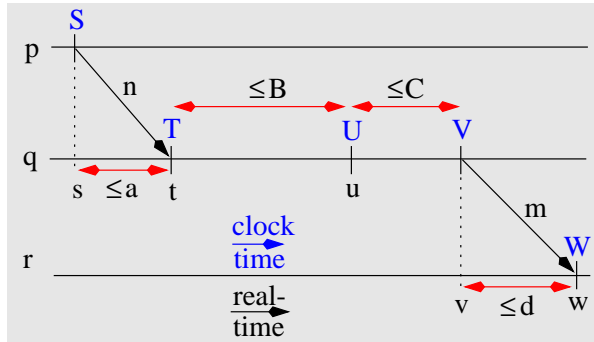


Figure 13: Processes use hardware clock time stamps to detect performance failures.

## 5.2 Detecting Process Performance Failures

Detecting performance failures is vital to many of our protocols to ensure their safety and timeliness properties. Our protocols read the local hardware clock at certain points during protocol execution. For example, all our protocols read the local hardware clock when a process (1) receives a message, (2) sends a message, (3) reads an indicator, or (4) is awakened by the operating system. Consider the situation shown in Figure 13 and let us assume that the standard execution can be stopped as soon as one of the bounds  $a, B, C$ , or  $d$  is violated. Process  $q$  reads its clock at real-times  $t, u$ , and  $v$  and its hardware clock returns the values  $T, U$ , and  $V$ , respectively. Process  $q$  checks in interval  $[t, u]$  if the transmission delay of  $n$  was at most  $a$  real-time units (using the fail-aware datagram service) while it checks in  $[u, v]$  that the processing time between  $t$  and  $u$  was at most  $B$  clock time units. The latter is easy to achieve because  $q$  can use

the time stamps  $T, U$  to test if  $U - T > B$ . Similarly,  $q$  can check if  $V - U > C$ . When  $q$  detects that at least one of the three bounds  $a, B$ , or  $C$  is violated, it does no longer send  $m$ .

## 5.3 Indicators

Real-time communication protocols can be divided into two broad classes [18]: *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* [23]. 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 (see also Figure 6). 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 time beyond which it cannot provide all its properties 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  $S, S + D, S + 2D, \dots$  to keep its clock synchronized (see Figure 14). When process  $p$  does not adjust its clock before a given deadline, its clock  $C_p$  may no longer be 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  $H_p = H_q$ ). 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 sync. We achieve that by using the following mechanism (see Figure 15). 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 14). 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 sync anymore.

A process  $q$  that reads  $I_p$  might itself suffer a performance failure while evaluating the current value of  $I_p$ . An indicator  $I_p$

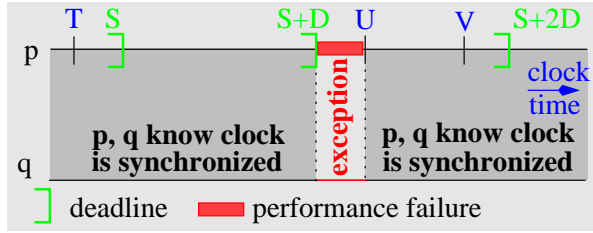


Figure 14: Process  $p$  has to adjust its clock before hardware clock times  $S$ ,  $S + D$ , and  $S + 2D$ . It actually performs the adjustments at times  $T$ ,  $U$ , and  $V$ . Since  $p$  misses deadline  $S + D < U$ , a process  $q$  that reads  $p$ 's clock between  $[S + D, U]$  has to detect that  $C_p$  might not be synchronized.

therefore returns the hardware clock time stamp used at the time its evaluation was requested. This allows the detection of performance failures that occur during the evaluation of  $I_p$  or during the usage of the value returned by  $I_p$ .

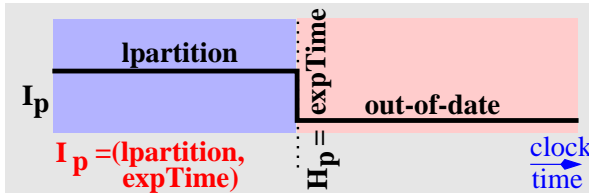


Figure 15: The indicator of a process  $p$  consists of the logical partition  $lpartition$  of  $p$  and an expiration time  $expTime$ , i.e., the time when the indicator will become *out-of-date*.

## 5.4 Locking Mechanism

Several of our protocols, e.g., the leader election service, use a *locking mechanism* to communicate by measuring the passage of time. This mechanism is similar to the “leases” mechanism of [15] 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 particularly useful for the coordination of the processes that switch the system to a safe mode (see Sections 6 and 7).

The locking mechanism works as follows (see Figure 16). A process  $p$  sets some local variable  $LV$  to a value  $V$  and sends a message  $m$  to a process  $q$  at real-time  $t$  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  $u \triangleq s - ub(m) + lockTime$ . The interesting part is that at time  $t + lockTime$  process  $p$  can change the value of  $LV$  without having to notify  $q$  about the 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 sending  $m$  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 (among other things) to guarantee that a process is at any point in real-time in at most one logical partition. We describe in Section 7 how this locking mechanism can be used to coordinate the switch to a safe mode in a traffic signaling application.

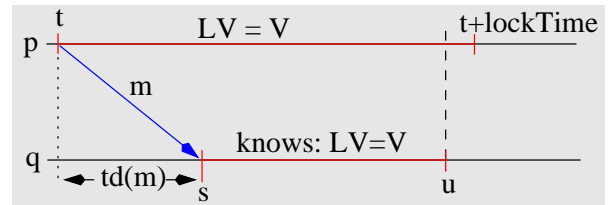


Figure 16: A process  $p$  guarantees not to change its variable  $LV$  for at least  $lockTime$  time units. Process  $p$  knows that  $q$  will use this information (transmitted in  $m$ ) at most to some time  $u \leq t + lockTime$  and hence,  $p$  can change  $LV$  after time  $t + lockTime$  without any further message based communication with  $q$ .

## 6 Switching to a Safe Mode

Switching a system to a safe mode is an important and challenging task: one has to detect when a system has to be switched and then one has to be able to perform this switch even if there are failures in the system. We explain in this section how fail-awareness can help in the detection and also in the actual switch. For concreteness, we will explain our approach using a simple railway crossing example.

Whether a system is safe is typically specified by one or more safety properties. For example, in Section 1.2 we stated such a property (ST) for a railway crossing: at any time  $t$ , if there is a train in the railway crossing at  $t$ , the crossing arms are down at  $t$ . We can formalize this using a predicate *TrainInCrossing* and *GatesDown*:

$$(ST) \forall t : TrainInCrossing(t) \Rightarrow ArmsDown(t).$$

Of course, one would like to specify that whenever there is no train in the crossing, one wants to open the crossing arms. Let predicate *OpenArms*( $t$ ) denote that the computing system requests to open the crossing arms. One can specify a liveness condition (L) that states that if there has not been a train in the crossing for some time  $RD$  (reaction delay) and there will not be a train in the crossing for  $RD$ , the system requests to open the crossing arms:

$$(L) \forall t : (\forall u \in [t - RD, t + RD] : \neg TrainInCrossing(u) \Rightarrow OpenArms(t).$$

To enforce these properties (ST,L), one has to make sure that whenever a train approaches the crossing, it is detected sufficiently early to be able to close the crossing arms before the train reaches the crossing. One can use sensors to detect an approaching train. Consider that the sensor data has to be sent from the sensor nodes via messages to some remote nodes that have to determine if a train is approaching. One has to make sure that even if sensor messages (or their processing) are delayed, one can reliably detect an approaching train.

In synchronous systems, one typically makes the assumption that at most  $F$  out of  $F + 1$  messages are delayed or dropped, where  $F$  is a small, a priori known constant. One can then send a message  $F + 1$  times to make sure that each sensor message is delivered in a timely manner. In this work, we do not want to make such an assumption because there is always the possibility that all  $F + 1$  messages are lost or delayed (e.g. in case all network cables are cut by some accident). Furthermore, such an assumption is not necessary to ensure the safety of the system, i.e., (ST).

Since we cannot assume that messages are timely, a computing system cannot always determine if there is a train approaching. The solution to this problem is to use a Boolean indicator  $I$  such that only if  $I(t)$  is true, a process can trust that the computed value of  $TrainInCrossing(t)$  is valid. In other words, one can implement to following liveness condition:

$$(FL) \forall t : (\forall u \in [t - RD, t + RD] : I(u) \wedge \neg TrainInCrossing(u) \Rightarrow OpenArms(t).$$

The crossing arms have to be closed if indicator  $I$  signals that the computation of  $TrainInCrossing$  might not be correct. In a system that supports hard real-time processes, a hard real-time process could check the indicator  $I$  periodically and then requests the arms to be closed if needed. In systems without hard real-time scheduling, we suggest the following interface to the external world (see Figure 17). The controlling computing system exports

- the current local time which is defined by local hardware clock,
- a Boolean  $ArmsUp$  that is true if the controlling computing system requests the crossing arms to be opened, and
- an expiration time (defined by indicator  $I$ ) that states when  $ArmsUp$  expires, i.e., when the arms should be closed.

As Figure 17 shows, one can use a comparator and an and-gate to make sure that the arms are closed in case the controlling computing system has no up-to-date information.

To implement the indicator  $I$ , sensors have to send their current state periodically to a process  $p$  that maintains the expiration time of  $I$ . If  $p$  receives a sufficient number of timely sensor messages, it can extend the expiration time for a certain time.

In other words, as long as a sufficient number of timely messages are delivered in time, the system can maintain the liveness property. However, if too many failures occur or sensors become disconnected, the system is automatically switched to a safe mode.

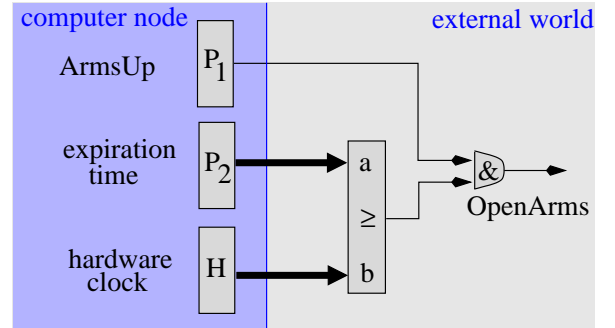


Figure 17: Fail-aware interface of a computing system.

In some systems, processes might need more information (e.g. some sensor data) to switch the system to a safe mode. To address this problem, processes should exchange the needed information  $SI$  periodically such that each process that participates in the switch has always an up-to-date  $SI$ . Only, if a sufficient number of processes can keep  $SI$  up-to-date, the system can be prevented from switching to a fail-safe mode. In case too many processes fail to update  $SI$ , the system switches to safe mode using the still up-to-date information  $SI$ .

## 7 Traffic Signaling Example

In this section we give a more detailed example of 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 controllers are common in factory floor control applications. We will illustrate the use of fail-awareness on an example that has a more complex switch to safe mode: a synchronized traffic signaling application. For a real-world, fully automated train system that uses fail-awareness please see [8].

We consider a system with two intersections (see Figure 18). 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.

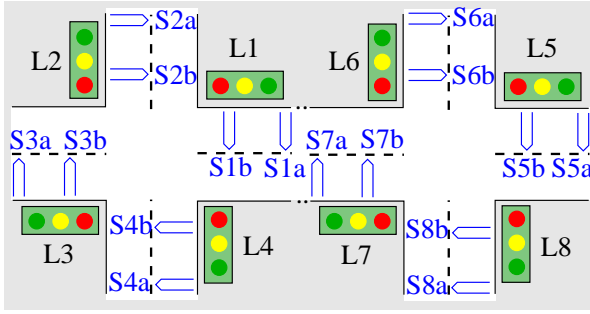


Figure 18: The traffic lights (L1-L8) of two intersections have to be synchronized. There are two sensors (S?a,S?b) for each traffic light (L?) (and there are two redundant controllers for each of the two intersections).

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 (see Figure 19). 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, both controllers are in the same logical partition (because at any point in real-time a process is in at most one logical partition). Hence, both controllers receive the same sensor broadcasts in the same order and thus, 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

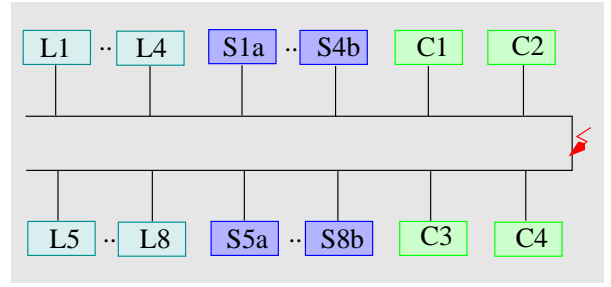


Figure 19: The computer system has one node for each traffic light (L?) and sensor (S?a,S?b). There are also two controllers (C1,C2) and (C3,C4) for each intersection.

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 watchdog process that switches it 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 a simple hardware circuit that checks if the indicator of a server is up-to-date (similar to that depicted in Figure 17) could be used to eliminate the high priority watchdog process.

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 20). 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.

## 8 Performance

We measured the performance of our services on a 10MBit Ethernet connecting several relatively slow SUN IPX workstations running SunOs 4.1.2.

The fail-aware datagram service calculates an upper bound

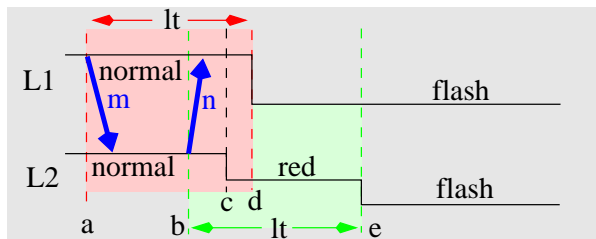


Figure 20: The two lights coordinate their switch to the flash state using a locking mechanism.  $L1$  can switch to the flash state at  $d$  because it knows that  $L2$  cannot assume after  $c < d$  that  $L1$  is in the normal state.  $L2$  cannot switch to the flash state until time  $e$  since it has sent  $n$  ensuring that it will not switch to the flash state before  $e$ .

$ub(m)$  on the transmission delay  $td(m)$  of each message the service delivers, i.e.,  $ub(m) \geq td(m)$ . In our first measurement we determine an upper bound on the error  $ub(m) - td(m)$  of the calculated upper bound (see Figure 21). Since we cannot measure the exact one-way transmission delay  $td(m)$  of a message, we approximate the error made by the fail-aware datagram service by the difference between the calculated upper bound and a known lower bound for the message transmission delay  $\delta_{min}$ , i.e we plot  $ub(m) - \delta_{min}$ . Note that this is a conservative approximation, in that the real error is always smaller than our approximation, i.e.,  $ub(m) - \delta_{min} \geq ub(m) - td(m)$ . This measurement is based on 20,000 round-trips of unicast message with a length of 248 bytes.

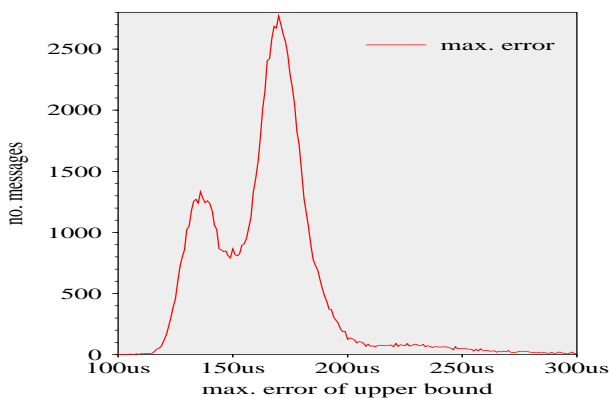


Figure 21: Measured error for the calculated upper bound on the transmission delay of unicast messages.

The next plot, shown in Figure 22, shows the measured times needed to elect a local leader [14] (these local leaders are instrumental in defining the logical partitions mentioned earlier). The measurements were based on 100,000 elections. The election time increases linearly with the number of processes participating in the election: the average election time and the 99% election time, i.e., a process succeeds with a 99% probability to become leader within that time, are shown in Figure 22.

The next measurement determines the deviation between synchronized clocks within a logical partition. The clock synchronization protocol synchronizes the clocks within a logical partition to the hardware clock of the local leader of the logical

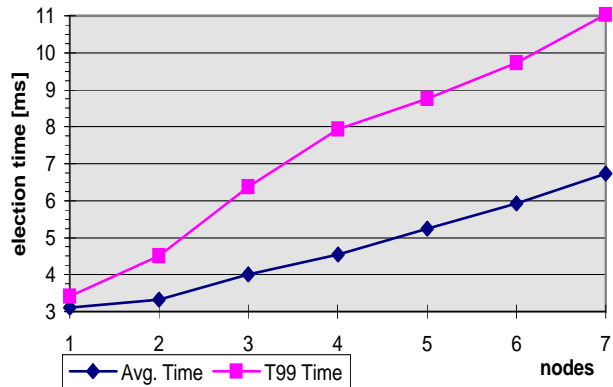


Figure 22: The average election time and 99% election time for 1 to 7 participating processes.

partition. Figure 23 plots an upper bound calculated for the deviation between the hardware clock of a local leader  $l$  and the synchronized clock of a process in  $l$ 's logical partition.

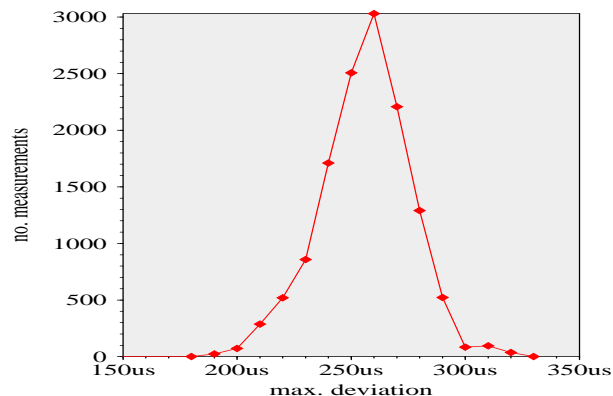


Figure 23: Measured deviation between the hardware clock of a local leader and the virtual clock of one of its supporters.

We also measured the time needed to remove a crashed process from the membership of a logical partition (see Figure 24). During this experiment the membership was updated every  $80ms$ . Because the membership protocol gives a process a “second chance” to prove that it is still in the same partition, it takes typically between  $80ms$  and  $160ms$  to remove a crashed process from the membership (see [10] for details).

The last measurement shows the delivery times of atomic broadcasts that are ordered according to their send time stamps (see Figure 25). We used the early delivery option that orders a broadcast message  $m$  as soon as the local leader knows that no other message has to be delivered before  $m$ . In our measurements a local leader had to wait for about  $10ms$  after it received a message and before it was able to order the message. The width of the experienced delivery times ( $[13ms, 23ms]$ ) is about  $10ms$ , which reflects the scheduling resolution of the operating system that is also  $10ms$ .

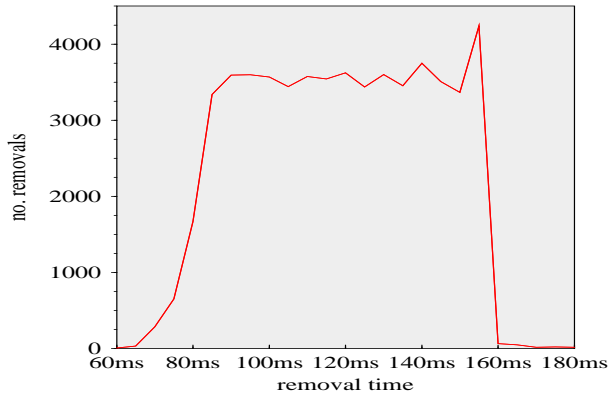


Figure 24: Measured removal times of a crashed process by a fail-aware membership protocol. Processes have to send an “alive-msg” at least every 80ms and get a “second chance” in case such a message is delivered late.

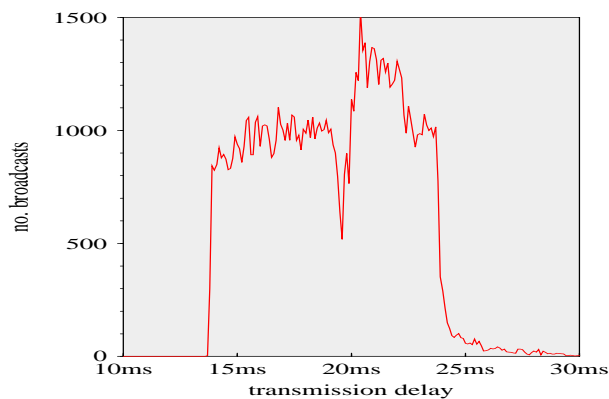


Figure 25: Measured delivery times of atomic broadcasts.

## 9 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 [18], XPA [25], TTP [17], and the Advanced Automation System family [4]. 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 [24]. 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 [7] and Totem [20]. 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 so 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 [9] as a general method of transforming synchronous service specifications into weaker, fail-aware service specifications that are implementable in timed asynchronous systems [5]. In our earlier work on fail-awareness [9] 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. The main contribution of this paper is to show how fail-awareness can be extended so that servers in multiple partitions can make progress (and hence, can increase the availability of the system) by introducing the concept of logical partition. The protocols presented in this paper support both operations, i.e., they allow us to either force only the servers of a majority partition to make progress, or to force servers to make progress even when they are in a minority partition. A service that can be viewed as an early example of a fail-aware service is the probabilistic clock synchronization service proposed in [1] for timed asynchronous systems: that service has a Boolean variable ‘synchronized’, that is true only when a clock is synchronized and is false when the clock may be out of synch.

## 10 Conclusion

The guaranteed response paradigm [18], which aims at guaranteeing timely responses, depends on the use of synchronous services, which in turn depend in a fundamental way on the assumption that the maximum number of failures per time unit is known at design time. If this failure assumption, basic to all synchronous service implementations, can be violated at runtime, these implementations can be subject to unpredictable behavior (due to delays and in particular, state contaminations). It is very difficult to guarantee such a failure assumption, in particular, for much of the off-the-shelf hardware and software. 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 properties like a synchronous service and when too many failures occur per time unit, the service lets its client know – in a timely manner – that it cannot guarantee its properties anymore. Fail-awareness allows the containment of failures by detecting when properties become invalidated due to excessive performance/crash/omission failures.

The target application domain of the fail-aware services we have designed are hard real-time applications built from COTS hardware and software that naturally have a safe state. Applications can use the indicators of the servers to detect when it is necessary to switch to a safe state. We believe that our fail-awareness approach is also advantageous in the construction of custom designed systems and for fail-operational systems.

## References

- [1] F. Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3:146–158, 1989.
- [2] F. Cristian. Reaching agreement on processor-group membership in synchronous distributed systems. *Distributed Computing*, 4:175–187, 1991.
- [3] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of ACM*, 34(2):56–78, Feb 1991.
- [4] F. Cristian, B. Dancy, and J. Dehn. Fault-tolerance in air traffic control systems. *ACM Transactions on Computers*, 14(3):265–286, Aug 1996.
- [5] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, pages 642–657, Jun 1999.
- [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 and D. Malki. The transis approach to high availability cluster communication. *Communications of the ACM*, 39(4):64–70, Apr 1996.
- [8] D. Essame, J. Arlat, and D. Powell. Padre: A protocol for asymmetric duplex redundancy. In *Proceedings of the Seventh IFIP International Working Conference on Dependable Computing for Critical Applications*, San Jose, USA, Jan 1999.
- [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. <http://www.christof.org/FA>.
- [10] C. Fetzer and F. Cristian. A fail-aware membership service. In *Proceedings of the 16th Symposium on Reliable Distributed Systems*, pages 157–164, Oct 1997. <http://www.christof.org/FAMS>.
- [11] C. Fetzer and F. Cristian. Fortress: A system to support fail-aware real-time applications. In *IEEE Workshop on Middleware for Distributed Real-Time Systems and Services*, San Francisco, Dec 1997. <http://www.christof.org/FORTRESS>.
- [12] C. Fetzer and F. Cristian. Derivation of fail-aware membership service specifications. In *Proceedings of the 3rd Annual Workshop on Fault-Tolerant Parallel and Distributed Systems*, Orlando, Florida, Apr 1998. <http://www.christof.org/DFAMS>.
- [13] C. Fetzer and F. Cristian. A fail-aware datagram service. *IEE Proceedings - Software Engineering*, pages 58–74, April 1999.
- [14] C. Fetzer and F. Cristian. A highly available local leader service. *IEEE Transactions on Software Engineering*, pages 603–618, Sept.-Oct. 1999.
- [15] 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.
- [16] 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.
- [17] H. Kopetz and G. Grunsteidl. TTP-a protocol for fault-tolerant real-time systems. *IEEE Computer*, pages 14–23, Jan 1994.
- [18] 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.
- [19] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of ACM*, 21(7):558–565, Jul 1978.
- [20] 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.
- [21] D. Powell. Failure mode assumptions and assumption coverage. In *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing Systems*, pages 386–395, 1992.
- [22] D. Stewart and P. Khosla. Mechanisms for detecting and handling timing errors. *Communications of the ACM*, 40(1):87–93, Jan. 1997.
- [23] P. Verissimo. Real-time communication. In S. Mullender, editor, *Distributed Systems, Second Edition*, chapter 17, pages 447–490. Addison-Wesley, New York, 1993.
- [24] 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.
- [25] 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.



**Christof Fetzer** received his diploma in computer science from the University of Kaiserslautern, Germany (12/92) and his Ph.D. from UC San Diego (3/97). He is a senior member of technical staff at AT&T Labs-Research. He received a two-year scholarship from the DAAD and won two best student paper awards. He was a finalist of the 1998 Council of Graduate Schools/UMI distinguished dissertation award. Dr. Fetzer has published over 30 research papers in the field of distributed systems.



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