PADRE : A Protocol for Asymmetric Duplex REdundancy

D. Essamé, J. Arlat, D. Powell
LAAS-CNRS, 7 avenue du colonel Roche, 31077 Toulouse cedex 4, France
{essame, arlat, dpowell}@laas.fr

Abstract

Safety and availability are issues of major importance in many critical systems. Ensuring simultaneously both attributes is sometimes difficult. Indeed, the introduction of redundancy to increase the overall system availability can lead to safety problems that would not otherwise exist. In this paper, we present a protocol for duplex redundancy management in critical systems that aims to increase the system availability without jeopardizing its safety. An application to a fully-automated train control system is described.

1 Introduction

Fault-tolerant computing systems are increasingly used to meet the stringent dependability requirements of automatic train control systems that, besides safety, extend to availability and to maintainability. Indeed, improvement of quality of service, continuity of service and cost-effective exploitation are adding new challenges to railway system designers beyond the underlying safety concerns. The introduction of redundant components is a necessary condition for increasing the overall system availability with respect to physical component failures. Here, we consider redundancy based on replicated *fail-safe* components. We formally investigate the conditions under which the safety properties of fail-safe components are preserved when they are replicated. We focus our analysis on duplicated fail-safe units interconnected with other such duplex systems by means of a local area network.

Given some safety constraints, we show that inconsistency of replicated units can lead to safety degradation even if each replicated component (taken individually) satisfies the given safety constraints. One way to circumvent such a problem is to avoid inconsistency by using, for example, an atomic broadcast protocol which ensures that replicated components agree on a consistent computational state. Unfortunately, such protocols rely on strong assumptions that cannot be satisfied with a sufficiently high confidence to meet the safety requirements of highly critical applications. Indeed, such a solution requires a perfectly reliable network ensuring bounded inter-process communication times.

Given that one cannot ensure that such strong assumptions will hold all the time in the real system, we propose a technique to tolerate state inconsistency. This technique consists in detecting potential inconsistencies and switching the system to a configuration that does not compromise safety in case of a real inconsistency. *PADRE* (Protocol for Asymmetric Duplex REdundancy) is an implementation of this technique for duplex redundancy, using the timed asynchronous system model [1]. We have chosen the timed asynchronous model because this model relies on realistic assumptions. Furthermore, our target systems (railway systems) have at least one safe state into which they can be switched at any point in time. This allows us to use the fail-awareness paradigm [2] to build a fail-safe protocol.

The rest of this paper is structured as follows. In *Section 2*, we state the problem by showing how state inconsistency can lead to safety degradation of replicated components even when each component is fail-safe. In *Section 3*, we present the system model. We formally investigate, in *Section 4*, the conditions under which the safety properties are preserved. In particular, we present a technique based on detection of potential inconsistencies and switching the duplex controller to a configuration where it does not impair safety in case of a real inconsistency. In *Section 5*, we present PADRE, which is a protocol that implements this technique using the timed asynchronous system model. Finally, *Section 6* concludes the paper.

2 Problem statement

The ability to build large complex systems from independently verified components is necessary for building affordable safe systems. For a critical system, it is important to guarantee that safety properties or *safety constraints* (*Sc*) of individual components are preserved in a system composed from those components. Given a set of n fail-safe redundant units u_1, u_2, …, u_n which satisfy individually a safety constraint *Sc* and a system composed of u_1, u_2, …, u_n, the main problem we address in this paper is the preservation of *Sc* in that composite system. We consider the special case where $n = 2$.

In this section, we show by means of an example that state inconsistency between redundant units can lead to violation of *Sc*.

2.1 Formalism and notations

In this sub-section, we define the formalism and notations that will be used in the rest of this paper. In particular, we formally define safety constraints by using temporal logic.

Here we consider a linear temporal logic, based on the set of real-time values Ω, with three temporal operators:

$\Box$ — meaning "*now and forever*",

$\Diamond$ — meaning "*now or sometime in the future*",

$\Diamond_d$ — meaning "*now or within bounded delay d in the future*".

Given a formula A then $\Box A$, $\Diamond A$, and $\Diamond_d A$ are also formulas. We use:

$\vdash A$ to express that the formula A is a theorem, i.e., it is always true.

We now use this linear temporal logic to express safety constraints of a critical system processing a *critical transaction* defined as a sequence of actions which, if executed incorrectly, can lead to a catastrophic failure.

Given a critical transaction c_t and a unit u, we denote by h the predicate such that $h(c_t,u)$ is true if the unit u is executing the critical transaction c_t and false if not. We define a safety constraint with respect to a critical transaction c_t by means of two predicates L and R on the state S_u of a unit u as follows:

$$(h(c_t,u) \quad L(S_u)) \qquad \Box(h(c_t,u) \quad R(S_u))$$

Such a formula expresses that, while executing the critical transaction c_t, if the state S_u of unit u becomes such that the predicate L holds then it must be the case that predicate R holds, and that S_u can only evolve in a way that allows the predicate R to continue to hold. We say that such a formula is a property of a unit u iff this formula is a theorem for that unit:

$$\vdash (h(c_t,u) \quad L(S_u)) \qquad \Box(h(c_t,u) \quad R(S_u))$$

2.2 State inconsistency

The introduction of redundancy can lead to safety problems that would not otherwise exist. We illustrate this problem by means of an example. We consider a fully-automated train control system made up of a set of *section controllers*, each in charge of a section of railway track (Fig. 1). Control of a train is handed over from one controller to the next when the train is located in an *inter-section lock*. In such a system, one of the critical transactions to be handled is automatic train driving, which is carried out by assigning each train a "target". A target is the point up to which a train may proceed, e.g., the next station, the next intersection lock, the track block before the next train, etc. The section controller must ensure that there is no obstacle, e.g., another train, between the current position of the train and the assigned target.

One way to achieve such a requirement is to ensure that trains' positions are known precisely by the section controllers. The following strategy can be used. To enter into a new section, a train must be identified by the controller of that section while the train is in the lock. If the train is successfully identified, the train is registered in the *monitoring list* of the controller. This monitoring list allows the controller to periodically poll the trains under its control to check their positions.

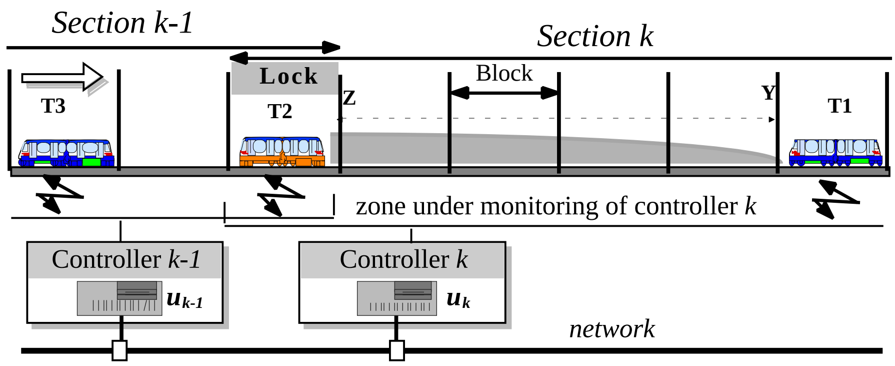

Fig. 1: Intersection handover with simplex controllers

Let us denote by *drive* the critical transaction of automatic driving. Let us consider the predicates $\textit{P}\text{ROCEED}_{Ti,Y}$ and $\textit{F}\text{REE}_{Ti,Y}$ such that: $\textit{P}\text{ROCEED}_{Ti,Y}(Su_k)$ is true if the state of unit u_k allows the train *Ti* to proceed to the target *Y* and false otherwise; $\textit{F}\text{REE}_{Ti,Y}(Su_k)$ is true if according to the state of unit u_k the track is free between the current position of the train *Ti* and the target *Y* and false otherwise. By using the previous formalism, one can define a safety constraint for the critical transaction *drive* with the following formula:

$$(h(drive,u_k)\quad \textit{P}\text{ROCEED}_{Ti,Y}(Su_k)) \qquad \Box(h(drive,u_k)\quad \textit{F}\text{REE}_{Ti,Y}(Su_k))$$

This formula means that, while unit u_k is executing drive ($h(drive,\ u_k)$ is true) then to authorize train *Ti* to proceed to *Y* ($\textit{P}\text{ROCEED}_{Ti,Y}(Su_k)$ is true), it must be the case that unit u_k perceives the track to be free between the train *Ti* and the target *Y* ($\textit{F}\text{REE}_{Ti,Y}(Su_k)$ is true). Note that $\textit{P}\text{ROCEED}_{Ti,Y}(Su_k)$ cannot be true if u_k has not registered *Ti* (a controller cannot assign a target to a train it does not know), but this does not violate the safety constraint (without a new target, train *Ti* will stop at its previous target, in this case, point *Z*).

Let us consider now that the controller of section *k* is made up of two units u_{k1} and u_{k2} in primary/secondary configuration and with unit u_{k1} as the primary. Consider a handover scenario where, due to transmission errors, the primary unit

u_{k1} identifies and registers the train *T2* at time *t* and assigns it the target *Y*, while the secondary unit u_{k2} does not register the train *T2* (Fig. 1).

Now assume that unit u_{k1} fails at some time $t' > t$ while the train *T2* is advancing to the target *Y*, so u_{k2} becomes the primary unit of the duplex controller (Fig. 2).

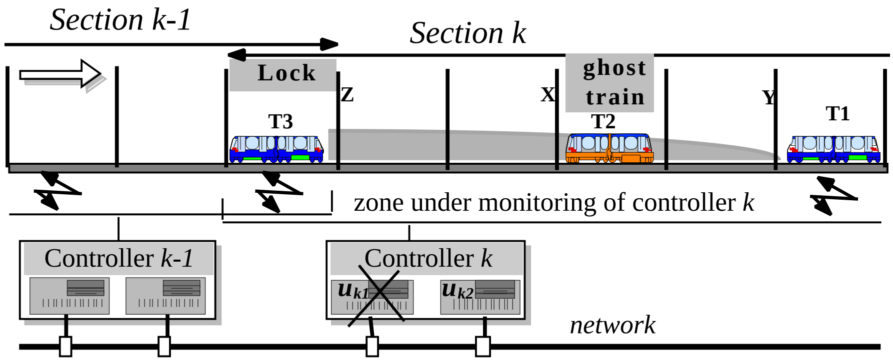

Fig. 2: Intersection handover with duplex controllers

Since u_{k2} has not registered the train *T2*, this train has become a “ghost train”. Indeed, when the train *T3* reaches the lock, it is identified by the unit u_{k2} which assigns it the target *Y* instead of *X* (Fig. 2). Of course, according to u_{k2}’s computational state there is no train between the current position of train *T3* and the target *Y* ($\text{FREE}_{T3,Y}(Su_{k2}) = true$) so unit u_{k2} satisfies the safety constraint. However this will not prevent the train *T3* from crashing into the train *T2* if it attempts to reach its target *Y*. Such a situation cannot happen in absence of replication. Indeed, if the section controller consists of only a single unit and if this unit fails to register train *T2* then, as stated earlier, train *T2* would be forced to stop at its previous target *Z* instead of being authorized to enter section *k*.

To circumvent such a problem, one approach would be to re-specify the application, by taking into account the fact that the section controller is made up of two units. Before issuing a critical output (such as allowing a train to enter a new section), the lists of trains monitored by each unit could be compared to check that they are consistent. Unfortunately, such an *ad hoc* solution introduces redundancy-related considerations at the application level. This makes the application programs very expensive to build, to verify and to maintain. To avoid this, we propose a redundancy management mechanism (PADRE) that frees the application programmer from such complications.

3 System model

We consider distributed real-time systems for critical applications with networked controllers made up of two fail-safe units The units can communicate with each other and with remote units by messages sent over a network (Fig. 3).

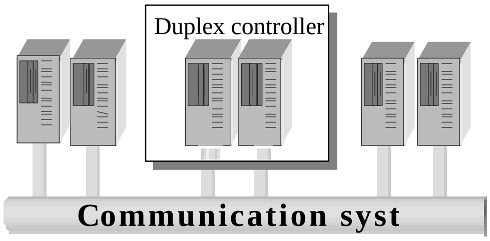

Fig. 3: System architecture

We base our approach on the timed asynchronous model [1].

First, the network is very reliable, but not enough for human lives to depend on it. Thus, from a safety viewpoint, we must assume that the network can lose or delay messages. However, message integrity is ensured by error-detecting codes. In the terminology of [3], messages sent over the network have omission/performance failure semantics.

Second, the units use a cyclic real-time executive such that it can be guaranteed that a message accepted by an operational unit will be processed in a bounded time interval or the unit will halt[1].

Third, although the local clocks of units are not synchronized, every unit checks the rate of drift of its local clock with respect to real-time and switches itself off if the drift exceeds a predefined bound[2]. Consequently, the local clock of an operational unit has a bounded rate of drift from real-time.

3.1 Local hardware clock

All processes that run on a unit can access the unit's hardware clock. Given a hardware local clock H, we denote by $H(t)$ the value displayed by H at real time t. We denote by Ω the set of real time values. Let ρ be the constant maximum drift rate that bounds the drift rate of a correct clock with $\rho \ll 1$. We assume that the clock granularity is negligible with respect to any useful time interval. In fact, we assume that a correct hardware local clock H satisfies the following relation:

$$t_1, t_2 \quad \Omega \qquad (t_2 - t_1)(1 - \rho) \leq H(t_2) - \mathrm{H}(t_1) \leq (t_1 - t_2)(1 + \rho)$$

[1] Note that this is a stronger condition than that imposed by the process management service of [1].
[2] This requires the local clock to be self-checking; see [4, pp 94-97] for a rudimentary technique.

3.2 The datagram service

The datagram service provides primitives for transmitting unicast and broadcast messages. The datagram service can delay or lose messages but provides the following properties:

- *Validity*: if the datagram service delivers a message m to a process p at time t and identifies a process q as m's sender, then q has indeed sent m at some earlier time $s < t$.
- *No-duplication*: each message has a unique sender and is delivered at a destination process at most once.

We denote by $td(m)$ the transmission delay of a message m.

We assume that any message sent between two remote processes p and q has a transmission delay that is at least δ_{min} : m, $td(m) \geq \delta_{min}$.

Messages can experience arbitrary transmission delays. However, we define a one-way time-out delay δ, such that: a message m whose transmission delay is at most δ, i.e., $td(m) \leq \delta$, is called *timely* otherwise, the message is *late*. The constant δ is closely related to the availability of services built using the timed asynchronous model. So, δ must be chosen such that most messages are delivered within δ time units (Fig. 4).

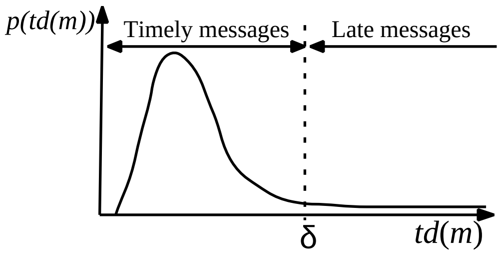

Fig. 4: Timely and late datagram messages

3.3 Δ-F-subsets and stable-subsets

Given a time interval and two processes p and q, we say that p and q are *F-connected* in that time interval, iff *i)* p and q are timely (their scheduling delays are bounded), *ii)* all but at most F messages sent between p and q are timely in that time interval (delivered within δ time units). When $F = 0$ we shall simply say that p and q are *connected*.

Given a constant Δ such that $\Delta > \delta$, a process p is Δ-*disconnected* from a process q in a given time interval, iff any message m that is delivered to p during that time interval from q has a transmission delay greater than Δ time units.

We say that a non empty subset of processes S is a *Δ-F-subset* in an interval [s, t] iff all processes in S are F-connected in [s, t] and processes in S are Δ-disconnected from all other processes. The notion of *Δ-F-subset* was introduced in [1][3].

We say that a non empty subset of processes S is a *stable-subset* in an interval [s, t] iff S is a *Δ-0-subset* in [s, t]. The notion of stable-subset is very useful when using the timed asynchronous system model. Indeed, a stable subset has the same behavior as that defined by the synchronous system model. This allows problems such as consensus to be specifiable in the timed asynchronous system model, but with respect to stable-subsets. Such specifications rely on a *progress assumption* [5] that states that the system is infinitely often "stable": there exists some constant η such that for any time s, there exists a time $t \geq s$ and a majority of processes SS such that SS forms a stable-subset in [t, $t + \eta$].

3.4 The fail-aware datagram service

The basic datagram service presented in Section 3.2 can delay messages. However, since such late messages can impair safety, we require a fail-aware datagram service similar to the one described in [6].

Given a constant Δ, the fail-aware datagram service computes an upper bound $ub(m)$ on the real transmission delay of each m and classifies m as *fast* or *slow* according to the following rule:

if $ub(m) < \Delta$ **then** m is *fast*
else m is *slow*.

In particular, if processes exchange message periodically, the constant Δ can be chosen such that timely messages are always classified as fast messages. Then, the fail-aware datagram service provides the following properties:

- *Validity*: if the fail-aware datagram service delivers a message m to a process p at time t and identifies a process q as m's sender, then q has indeed sent m at some earlier time $s < t$.
- *No-duplication*: each message has a unique sender and is delivered at a destination process at most once.
- *Fail-awareness*: each message classified as a *fast* message has experienced a real transmission delay of at most Δ time units.

$$m\ (m \text{ is } \mathit{fast}) \quad (td(m) < \Delta)$$

[3] In [1], the term *Δ-F-partition* was used. We have chosen the term *Δ-F-subset* to avoid misunderstanding with the mathematical meaning of a partition as a set of disjoint subsets.

- *Timeliness*: there exists a constant τ such that if two processes p and q exchange messages at least every τ time units then, if p and q are connected in an interval $[s - \delta, s + \tau]$, each timely message sent in $[s, s + \tau]$ must be classified as a *fast* message.

For more details on how to compute the upper bound $ub(m)$ and how to choose the constant Δ such that the Fail-awareness property and the Timeliness property always hold, the reader should refer to [6].

The fail-aware datagram service is fundamental for our redundancy management mechanism. Indeed, this service is used to detect when the communication system has suffered a performance failure. In addition, our mechanism uses the same fail-awareness philosophy to deliver messages to the application layer.

4 Safety properties preservation in a duplex controller

We have shown in Section 2.2 that state inconsistency can impair safety. One solution to handle this problem could have been to avoid state inconsistency by having units agree before accepting new inputs. Unfortunately, it has been shown that two units cannot achieve agreement if messages between them can be lost (e.g., see the two generals' problem in [7]). Here, we present an alternative solution that allows the safety constraints to be guaranteed by tolerating state inconsistency. The key idea of our approach is to detect potential state inconsistencies and to switch the duplex controller to a configuration which allows the safety constraints to hold in case of a real inconsistency. To achieve that, we use an *asymmetric coordination* of replicated units. The asymmetric coordination of replicated units consists of letting one unit, called the *Primary*, have a dominating role with respect to the other unit. The Primary can take unilateral decisions (such as the order in which the inputs must be accepted). The Primary can impose its choice on the other unit without resorting to a consensus protocol. Asymmetric coordination is particularly attractive when it is impossible to have a majority agreement, as in the case of a duplex controller. However, one can only use this technique if the Primary cannot send erroneous messages, which is the case here since all units are fail-safe.

4.1 Mode of operation of a fail safe-unit

To achieve the asymmetric coordination of a duplex controller, we define four modes of operation for each fail-safe unit: *primary*, *standby*, *quarantine* and *failed* (Fig. 5). When the unit is in the quarantine or failed modes, it is said to be *non-*

operational or *safe*, and cannot deliver outputs to the environment. A unit in the primary or standby modes is said to be *operational*.

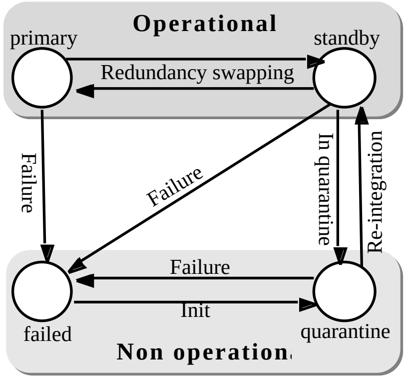

Fig. 5: Modes of operation of a fail safe unit

A unit is in the *primary* mode when it is the current Primary. A unit in the standby or the quarantine modes is called the Secondary. The current Secondary can be in the *standby* mode only if its state is consistent with the current Primary's state. Otherwise, it is in the *quarantine* mode. The quarantine mode is an intermediate mode that is introduced for safety purposes: the Secondary is put in quarantine when its state might be inconsistent with the Primary's state. Toward this goal, we can state the following objective for safe operation: *The protocol that manages the redundant pair of units must either ensure that their states are kept consistent or else force the Secondary into the quarantine mode.*

The quarantine mode allows a configuration of the duplex controller (one unit in the primary mode and the other unit in the quarantine mode) which does not impair safety. Indeed, when the Secondary unit is in quarantine, it is non-operational, so it cannot carry out any interaction with the environment (thus avoiding actions that could be in conflict with safety actions carried out by the Primary unit), nor can it be switched to the primary mode (thus avoiding the sort of situation described in Section 2.2).

We call *nominal configuration*, the configuration where one unit is in the primary mode and the other unit is in the standby mode. We call *safe configuration*, the configuration where one unit is in the primary mode while the other unit is in quarantine. In the nominal configuration, availability is ensured even a unit fails. In the safe configuration, fault-tolerance is sacrificed temporally so as to ensure safety. In that configuration, availability is only ensured if the current Primary does not fail. However, in both configurations, safety is always ensured. In the safe configuration, a recovery procedure allows the Secondary unit to reinitialize its state from the Primary unit, allowing the duplex controller to revert to the nominal configuration.

In the next two subsections, we give some safety-related and availability-related properties of the redundancy management mechanism.

4.2 Safety-related properties

We call *history* of unit u at time t, denoted $H(u,t)$, the string of events that unit u has accepted since its initialization up until time t, ordered following the time of taking them into account. Given two units i and j, we say that $H(i,t)$ is a *prefix* of $H(j,t)$ (denoted $H(i,t) \prec H(j,t)$) iff $H(i,t)$ is a segment of $H(j,t)$ and $inf(H(i,t)) = inf(H(j,t))$ where $inf(H(i,t))$ denotes the first element of $H(i,t)$.

Let $S_u(t)$ be the state of a unit u at time t and $E_u(t)$ its mode, with $E_u(t) \quad \{primary, standby, quarantine, failed\}$. Here we define three properties that are needed to guarantee a safe behavior of a redundant pair of fail-safe units u_1 and u_2. Given i and j, such that $i,j \quad \{u_1,u_2\}$, $i \neq j$, then the following properties are required[4] :

UP: *Unique Primary*: both units cannot be in the primary mode simultaneously:

$\vdash (E_i = primary) \quad (E_j \neq primary)$

MQ: *Quarantine*: the Secondary unit must leave the standby mode within a bounded delay Q if its state is inconsistent with the Primary's state and it cannot return to the standby mode while its state is inconsistent:

$\vdash (E_i = primary) \quad (S_i \neq S_j) \quad \Diamond_Q \Box((S_i \neq S_j) \quad (E_j \neq standby))$

PH: *Prefix of History*: the history of the Primary unit must always be a prefix of the history of the Secondary unit:

$\vdash (E_i = primary) \ (E_j = standby) \quad (H(i) \prec (H(j))$

The *UP* property prohibits the possibility of having two Primary units. This is for safety, since we must have only one Primary at any given instant. The *MQ* property reflects the need to ensure that the Secondary unit cannot be maintained in the standby mode if its state is inconsistent with the Primary's state. However, inconsistency is authorized for a bounded duration (operator $\Diamond_Q$) at the end of which the Secondary unit must leave the standby mode. The Secondary unit cannot come back to the standby mode while its state is still inconsistent (operator $\Box$). The *PH* property ensures that the Secondary unit is aware of all events that have been taken into account by the Primary. In particular, this property ensures that the computation carried out by the Secondary unit cannot be late with respect to that of the Primary unit. This is a very useful property when redundancy switching occurs. Indeed, it ensures that the Secondary unit has

[4] The temporal logic notation allows all properties to be stated without explicit time parameters.

at least the same knowledge of the controlled process as the Primary unit. So, since both units are fail-safe, if the Primary unit leaves the controlled process in a non-dangerous state then the Secondary unit, when becoming Primary will maintain the controlled process in a non-dangerous state.

Moreover, property *PH* guarantees that the Secondary unit cannot revert to the standby mode without recovering its history from the Primary unit. Indeed, let us suppose that unit i is in the primary mode ($E_i = primary$) and that unit j does not recover its history from unit i, i.e: $\neg (H(i) \prec (H(j))$, we have:

(1) $\vdash (E_i = primary)$; (hypothesis: unit i is in the primary mode)

(2) $\vdash \neg (H(i) \prec (H(j))$; (hypothesis: unit j does not recover its history from unit i)

(3) $\vdash (E_i = primary) \quad (E_j = \text{standby}) \qquad (H(i) \prec (H(j))$; property *PH*

(4) $\vdash \neg (H(i) \prec (H(j)) \qquad \neg ((E_i = primary) \quad (E_j = standby))$; contraposition of (3)

(5) $\vdash \neg (E_i = primary) \quad \neg(E_j = standby)$; modus ponens on (2) and (4)

(6) $\vdash (E_i = primary) \qquad \neg (E_j = \text{standby})$; rewriting of (5) and definition of

(7) $\vdash \neg (E_j = standby)$; modus ponens on (1) and (6); or equivalently:

(8) $\vdash E_j \neq standby$; unit j cannot be in the standby mode ■.

This result can be summarized by the following relation:

$$\vdash (E_i = primary) \quad \neg(H(i) \prec (H(j)) \qquad (E_j \neq standby).$$

In conclusion, property *PH* imposes the implementation of a recovery mechanism that allows the Secondary unit to recover its history from the Primary unit. It has been shown formally in [8] that properties *UP*, *MQ* and *PH* are sufficient to ensure preservation of the safety constraints of redundant fail-safe units in a duplex controller.

4.3 Availability-related properties

While a unit is in the quarantine mode, it cannot deliver outputs to the controlled process. Moreover, it is unable to replace the other unit should the latter fail. Consequently, to provide availability, the protocol must attempt to maintain the Secondary in the standby mode or to bring it back to that mode when it has been put into quarantine.

To ensure availability, two progress properties must therefore be respected, but only in the absence of failures:

AG: *Agreement*: there exists a constant τ such that, in absence of failures, every message accepted by one unit at time t must have been accepted by the other unit within the interval $[t - \omega, t + \omega]$

LQ: *Limited Quarantine*: in absence of failures, a unit in the quarantine mode must eventually switch back to the standby mode.

Property *AG* prevents useless solutions in which the unit in the standby mode immediately switches to the quarantine mode. Property *LQ* prevents the trivial solution in which one unit always remains in the quarantine mode. For safety, there is no obligation to achieve consistency or to maintain both units operational since those are availability requirements. However, there is an obligation to put and keep the Secondary unit in quarantine whenever its state is inconsistent with the Primary's state.

5 PADRE

In this section, we present a redundancy management protocol called PADRE (Protocol for Asymmetric Duplex REdundancy) which ensures the properties *UP*, *MQ*, *PH*, *AG* and *LQ*. The protocol is implemented by means of two protocol entities (one in each unit) which intercept all messages addressed to the application layer (Fig. 6). The protocol attempts to deliver all messages to the application layer of both units. If a transmission error should occur that prevents the message from being delivered to the application layer of both units, then a state inconsistency can occur. In this case, the protocol must ensure that property *MQ* holds by forcing the Secondary unit to switch to the quarantine mode.

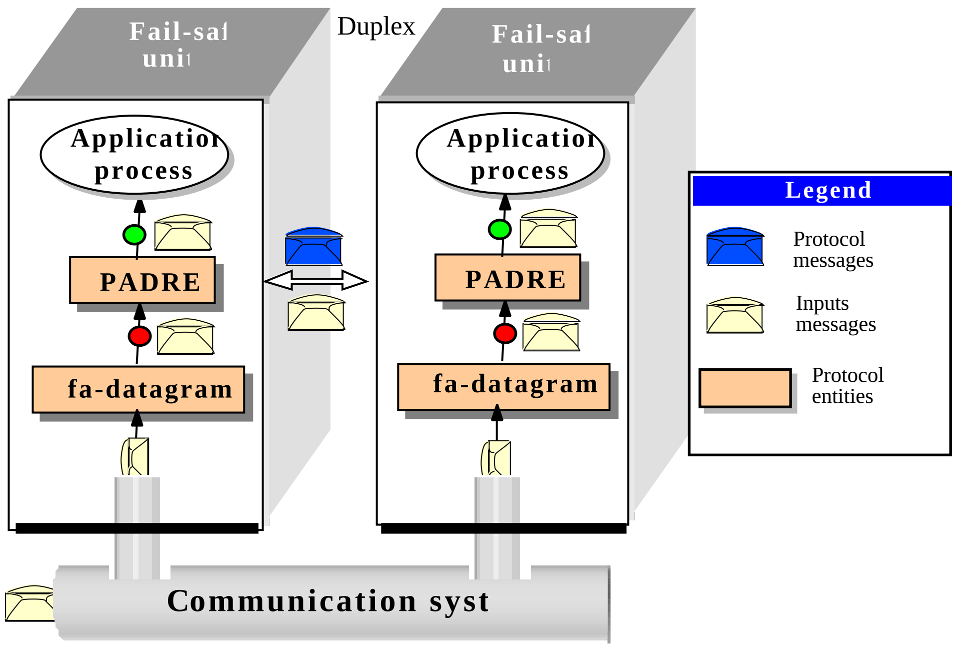

Fig. 6: Padre

5.1 Specifications

PADRE uses the fail-aware paradigm to ensure safety. The protocol relies on a fail-aware datagram service and satisfies the following requirements:

R1: *Validity*: if PADRE delivers a message m to an application process p at time t and identifies a process q as m's sender, then q has indeed sent m at some earlier time $s < t$.

R2: *No-duplication:* each message has a unique sender and is delivered at a destination process at most once.

R3: *Fail-awareness:* there exist two constants ω and Q such that, if the protocol entity of the Primary unit receives a message m at time t, and the Secondary unit is in the standby mode, then:

- *a*) either the protocol entity of the Primary delivers m to its application layer at time t_1 ($t_1 > t$) with a *true* indicator if m has been delivered to the application layer of the Secondary within the interval $[t - \omega, t]$.
- *b*) or the protocol entity of the Primary delivers m to its application layer at time t_1 (with $t_1 > t + Q$) with a *false* indicator and the Secondary is put into quarantine, if it does not fail, at the latest at time $t + Q$.

R4: *Asymmetry:* the protocol entity of the Secondary only delivers to the application layer messages that are forwarded to it by the protocol entity of the Primary. All such messages are delivered with their indicator set to *true.*

R5: *Timeliness*: There exists a constant I such that when the Secondary unit is in the standby mode:

- *a*) if both protocol entities are connected in the interval $[t, t + I]$, then the Secondary unit is not put in quarantine in this interval;
- *b*) if the Primary unit fails at time t, the Secondary unit must switch to the primary mode within the time interval $[t, t + I]$.

The requirements *R1* and *R2* are safety-related requirements which ensure that PADRE delivers only real messages. Furthermore, requirement *R1* is necessary to ensure property *PH*. Requirement *R3* bounds the difference between the instants at which the protocol entities deliver messages to the application layer. Indeed, this requirement ensures that when the protocol entity of the Primary delivers a message to its application layer, this message has been delivered to the application layer of the Secondary or the Secondary has been put into quarantine. In particular, its clause *R3-a*) allows properties *AP* and *PH* to be satisfied while its clause *R3-b*) allows property *MQ* to be satisfied. Requirement *R4* expresses the asymmetric behavior of the protocol. An indicator set to *true* tells the application entity (should it wish to know) that the duplex pair is still capable of tolerating a fault. Requirement *R5* ensures that the duplex controller remains available in the absence of communication failures.

Property *UP* concerns the designation of the Primary unit. To fulfil this, it would have been desirable to use a software mechanism such as a leader election protocol. Therefore, considering the fault assumptions of our system model (unbounded communication delay and messages can be lost), this would require a third unit to allow a majority decision. The principle used would then be as follows: a unit of the duplex computer which wishes to become the Primary must obtain the support of that third unit, knowing that the latter can only support one unit at a time, and this support is of limited duration and has to be renewed periodically. Such an extension to redundancy levels greater than two is described in [8].

However, such an approach is contrary to the principle of autonomous duplex controllers. Therefore, the property *UP* is handled by hardware using a *bi-stable safety relay* which ensures that only one unit can be in the primary state at once.

In conclusion, the requirements *R1* through *R5*, together with the bi-stable safety relay, allow satisfaction of the properties *UP*, *MQ*, *PH* and *AP*. The property *LQ* is taken into account by the Secondary recovery mechanism, which will be described later.

5.2 Description

We successively describe the nominal and safe configurations.

Nominal configuration: in this configuration, both units are operational. So, safety is the key issue. The main idea is to attempt to ensure state consistency through broadcasting inputs to both units atomically. If atomicity cannot be ensured, the Secondary unit is put into quarantine, to ensure safety property *MQ*. The principle used is the following (Fig. 7):

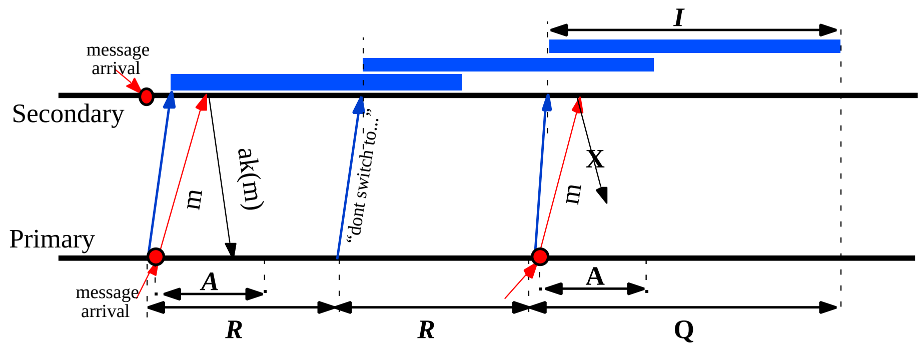

Fig. 7: PADRE principle

a) Primary

- Send, every R time units, a message to the Secondary "*Don't switch to quarantine*" and set the quarantine time-out delay Q (the quarantine time-out delay is the time-out delay such that if the Primary stops sending "Don't switch to quarantine" messages at time t then the Secondary will leave the standby mode at the latest by time $t + Q$).
- Each time an input message is received from a remote controller, forward this message to the Secondary, set a *wait time-out* delay of A time units and wait for an acknowledgement:
  - if the acknowledgement is received before the *wait* time-out expires, accept the message;
  - if the *wait* time-out expires then start operating autonomously:
    - stop sending "*Don't switch to quarantine*" messages,
    - stop forwarding input messages to the Secondary,
    - accept the pending message(s) after the quarantine time-out delay Q expires.
- If the Primary fails, the safety relay will switch the current Secondary to the primary mode.

b) Secondary

- Wait for the periodic "*Don't switch to quarantine*" message. When receiving such a message, set a *stay-alive time-out* delay I (Fig. 7). If no such message is received before the stay-alive time-out expires, then switch to the quarantine mode.
- Each time an input message is received directly from a remote controller, forward this message to the Primary (when the Primary receives this message, it behaves as previously).
- Each time an input message is received from the Primary, send an acknowledgement to the Primary and accept the message. (Note that the message can be accepted immediately by the Secondary since the Primary has seen the same message, so the latter will either accept the message in a bounded time or cause the Secondary to switch to the quarantine mode.)
- The failure of the Secondary has no immediate effect. The Primary will be informed of the failure when it next attempts to forward a message since it will not receive an acknowledgement.

Safe configuration: in this configuration, the Secondary unit is in quarantine. So, the key issue is availability since, while the Secondary unit is in quarantine, it is not in a position to replace the Primary should the latter fail. For availability, the state of the Secondary has to be made consistent with that of the Primary so that it can revert to its backup role. This is achieved by executing a protocol that copies the state of the Primary to the Secondary. We use the recursive state recovery protocol of [9]. In this protocol, the state is divided into "chunks" that can be transferred in a single message. A tagging mechanism is used to identify chunks that have been modified since they have been transferred. The transfer of chunks continues recursively until there are no newly-tagged chunks. The last chunk is identified as such by the Primary. The Secondary must remain in quarantine until state transfer has been successfully completed. The protocol can be summarized as follows:

a) Primary

- Transfer state to Secondary. When the last state transfer message has been sent to the Secondary[5]:
  - resume forwarding input messages, instead of delivering them directly to the application,
  - resume sending "*Don't switch to quarantine*" messages.

b) Secondary

- Wait for last state transfer message and switch to the standby mode.

For improved availability, messages and message acknowledgements can be repeated.

5.3 Choice of the quarantine time-out delay Q

Due to space restriction, we cannot give the expressions of all PADRE's parameters (A, R, I, ω and Q). However, since the quarantine time-out delay is a key parameter of PADRE, here we give the relationship that must be satisfied when choosing this delay.

To establish this relation, we consider the scenario illustrated by Fig. 8, where the Primary unit sends a "*Don't switch to quarantine*" message at time t_1 and this message is received by the Secondary unit at time t_2.

[5] Note that it is not necessary for the Primary to wait for the acknowledgment of the last chunk to resume forwarding input messages. If the last chunk is lost, then the Secondary will not switch to the standby mode and so will not acknowledge messages forwarded to it by the Primary, which will cause the latter to resume autonomous operation.

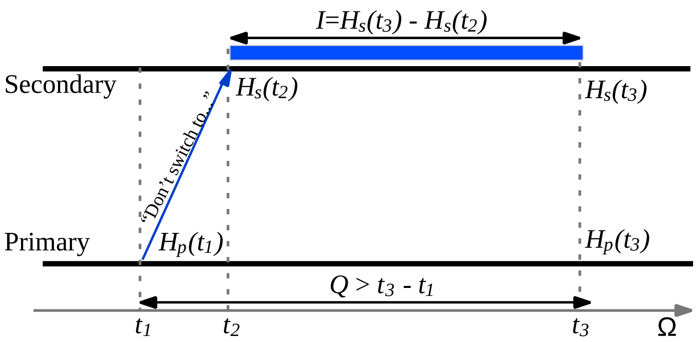

Fig. 8: The quarantine time-out delay Q

By definition, the "*Don't switch to quarantine*" message allows the Secondary unit to stay in the standby mode until time t_3 such that $I = H_s(t_3) - H_s(t_2)$ where I is the stay-alive time-out delay and H_s the clock of the Secondary unit. So that the Primary can be sure that the Secondary unit has put itself into quarantine, it is required that: $H_p(t_3) \leq H_p(t_1) + Q$ or equivalently: $Q \geq H_p(t_3) - H_p(t_1)$.

We have:

$$t_3 - t_1 = (t_3 - t_2) + (t_2 - t_1) \qquad (1)$$

Since clocks of both units have bounded drift rate (ρ <<1), one can write:

$$H_p(t_3) - H_p(t_1) \leq (t_3 - t_1)(1+\rho) \qquad (2)$$

and:

$$(t_3 - t_2) \leq (H_s(t_3) - H_s(t_2))(1+\rho) = I(1+\rho) \qquad (3)$$

We use the fail-aware datagram service to transmit the "*Don't switch to quarantine*" messages and throw away all slow messages. This guarantees that the Secondary cannot use a message with a transfer delay greater than Δ to refresh its stay-alive time-out delay. So, one can write:

$$t_2 - t_1 \leq \Delta \qquad (4)$$

Substituting (3) and (4) into (1) we obtain:

$$t_3 - t_1 \leq \Delta + I(1+\rho) \qquad (5)$$

It follows from (2) and (5):

$$H_p(t_3) - H_p(t_1) \leq \Delta\,(1+\rho) + I(1+2\rho) \qquad (6)$$

So, to ensure that the Secondary unit has put itself in quarantine knowing that the Primary unit has stopped sending "Don't switch to quarantine" messages, one must have:

$$Q \geq \Delta(1+\rho) + I(1+2\rho)$$

This use of time-out to communicate knowledge without explicit message passing is the "communication by time" paradigm of [1, 5].

5.4 Application of PADRE

PADRE has been applied to the fully automated train control system with networked duplex controllers considered in Section 2.2. We have shown that, in such a system, the inconsistency of lists of monitored trains maintained by each unit of a section controller can compromise safety. We illustrate in this subsection how PADRE can be used to circumvent such a problem without re-specifying the application. Indeed, PADRE allows transparent re-use of an application program designed for a non-redundant section controller.

Each replicated unit has a PADRE protocol entity which intercepts all messages that are addressed to the application modules of the unit. PADRE's protocol entity works as follows. When a message such as "*Train T2 is entering in section k*" is addressed to an application module, this message is intercepted by the protocol entity, which applies the protocol we have previously described. Let us suppose that the Primary unit of *section controller k* receives the message "*Train T2 is entering in section k*" (see Fig. 1). Its protocol entity will intercept this message and forward a copy to the protocol entity of the Secondary unit and set a timeout delay. If the protocol entity of the Secondary receives that copy, it sends an acknowledgment to the protocol entity of the Primary and delivers the message to the Secondary's application module. If the protocol entity of the Primary unit receives the acknowledgment before the timeout delay expires, it delivers the message to the Primary's application module. Otherwise, it stops refreshing the Secondary unit and waits Q time units since its last "Don't switch to quarantine" message before delivering the message to the Primary's application module.

This principle ensures that when the Secondary unit is not in quarantine, it is always aware of all events that have been taken into account by the Primary unit. This ensures safe operation of the duplex controller without changing the application modules.

6 Conclusion

In order to tolerate state inconsistency in a duplex fail-safe controller, we have presented a technique that consists in detecting potential inconsistencies and switching the duplex controller to a configuration where it does not impair safety in case of a real inconsistency.

Using the timed asynchronous model and the fail-awareness paradigm, we have developed a protocol which implements this technique. The key idea of the protocol is to try to keep both units consistent by attempting to agree on input messages; however, if this agreement fails, the protocol switches the duplex controller to a configuration ensuring safe operation.

This protocol has been applied to a fully-automated train control system made up of duplex controllers. This protocol provides a design paradigm enabling a substantial reduction in the cost of designing and validating critical systems.

Acknowledgments

This work was partially financed by Matra Transport International. The authors would particularly like to thank Philippe Forin and Benoît Fumery of Matra Transport International, for their stimulating inputs to this work. The authors also wish to express their sincere thanks to the anonymous reviewers for their valuable comments and suggestions.

7 References

[1] F. Cristian and C. Fetzer, “The Timed Asynchronous Distributed System Model”, in *28th Int. Symp. on Fault-Tolerant Computing (FTCS-28),* (Munich, Germany), pp.140-149, IEEE Computer Society Press, 1998.

[2] C. Fetzer and F. Cristian, “Using Fail-Awareness to Design Adaptative Real-Time Applications”, in *National Aerospace and Electronics Conference,* (Dayton, Ohio, USA), pp.101-115, IEEE Computer Society Press, July 14 -18 1997.

[3] F. Cristian, “Understanding Fault-Tolerant Distributed Systems”, *Comm. ACM*, 34 (2), pp.56-78, 1991.

[4] J. Wakerly, *Error Detecting Codes, Self-Checking Circuits and Applications*, Elsevier North-Holland, New York, 1978.

[5] C. Fetzer and F. Cristian, “On the Possibility of Consensus in Asynchronous Systems”, in *Pacific Rim Int. Symp. on Fault-Tolerant Systems,* (Newport Beach, CA), 1995.

[6] C. Fetzer and F. Cristian, “A Fail-Aware Datagram Service”, in *2nd Annual Worshop on Fault-Tolerant Parallel and Distributed Systems,* (Geneva, Switzerland), IEEE Computer Society Press, April 1997.

[7] J. Gray, “Notes on Database Operating Systems”, in *Operating Systems: An Advanced Course,* (R. Bayer, R. M. Graham and G. Seegmuller, Eds.), Lecture Notes in Computer Science, pp.393-481, Springer-Verlag, Berlin, 1978.

[8] D. Essamé, *"Fault Tolerance in Critical Systems: Application to Automatic Subway Control"*, Doctoral Thesis, National Polytechnic Institute of Toulouse, LAAS-CNRS, Report N°98414, November 1998 (in French).

[9] A. Bondavalli, F.-D. Giandomenico, F. Grandoni, D. Powell and C. Rabejac, “State Restoration in a COTS-Based N-Modular Architecture”, in *1st IEEE Int. Symp. on Object-oriented Real-time Distributed Computing (ISORC'98),* (Kyoto, Japan), pp.1-16, IEEE Computer Society, April 20-22 1998.