The Timely Computing Base Model and Architecture

Paulo Veríssimo, António Casimiro

Abstract— **Current systems are very often based on large-scale, unpredictable and unreliable infrastructures. However, users of these systems increasingly require services with timeliness properties. This creates a difficult-to-solve contradiction with regard to the adequate time model: synchronous, or asynchronous? In this paper, we propose an architectural construct and programming model, which address this problem. We assume the existence of a component that is capable of executing timely functions, however asynchronous the rest of the system may be. We call this component the Timely Computing Base, and it can be used by the other components to execute a set of simple but crucial time-related services. We also show how to use it to build dependable and timely applications exhibiting varying degrees of timeliness assurance, under several synchrony models.**

Keywords— **Distributed systems, Real-Time systems, Timely Computing Base, Partial synchrony models**

I. Introduction and Motivation

The growth of networked and distributed systems in several application domains has been explosive in the past few years. This has changed the way we reason about distributed systems in many ways. For a large number of today's services, the real-time and fault tolerance requirements reach levels only seen previously in smaller, ad-hoc systems.

Formally, these requirements are *timeliness* specifications, in essence meaningful only in the synchronous system model. Under this model there are known bounds for timing variables, and the mechanisms to meet reliability and timeliness requirements are reasonably well understood, both in terms of distributed systems theory and in real-time systems design principles (e.g., real-time communication, scheduling and replication management). However, unpredictable and unreliable infrastructures are not adequate environments for synchronous models, since it is difficult to enforce timeliness assumptions. Violation of assumptions causes incorrect system behavior. In alternative, the asynchronous model is a well-studied framework, appropriate for these environments. 'Asynchronous' means, in order to be simple at this point, that there are no bounds on essential timing variables, such as processing speed or communication delay. This model has served a number of applications where uncertainty about the provision of service was accepted.

P. Veríssimo and A. Casimiro are with the Faculty of Sciences of the University of Lisbon (FCUL), Lisboa, Portugal. E-mail: pjv@di.fc.ul.pt and casim@di.fc.ul.pt. This work was partially supported by the FCT, through projects Praxis/P/EEI/12160/1998 (MICRA) and Praxis/P/EEI/14187/1998 (DEAR-COTS), and by the EC, through projects IST-2000-26031 (CORTEX) and IST-1999-11583 (MAFTIA).

In consequence, this status quo leaves us with a problem: fully asynchronous models do not satisfy our needs, because they do not allow timeliness specifications; on the other hand, correct operation under fully synchronous models is very difficult to achieve (if at all possible) in infrastructures with uncertain baseline timeliness properties. One issue of definitive importance is the following: what system model to use for applications with synchrony (i.e. real-time) requirements running on environments with uncertain timeliness?

We propose the **Timely Computing Base (TCB)** model to address this problem. We assume that systems, however asynchronous they may be, and whatever their scale, can rely on services provided by a special module, the TCB, which is timely, that is, synchronous. Furthermore, in a distributed system the TCB provides such services in a distributed way.

All that is required is that applications follow a certain programming style, depending on the degree of timeliness assurance desired. Classically, a component timing failure is treated as a single phenomenon. However, we show that in fact there are three mechanisms by which timing failures impact a system: unexpected delay; contamination; and decreased coverage. The innovative aspect of this failure analysis is that one can fine-tune the treatment of timing failures and build applications with varying degrees of dependability and timeliness on systems with uncertain temporal behavior.

The TCB concept has a certain analogy with the approach described by the same acronym in security[1], the 'trusted computing base', a platform that can execute secure functions, even if immersed in an insecure system, subjected to intentional faults caused by intruders. The analogy is nevertheless superficial: whereas the trusted computing base framework is concerned with fault prevention, the timely computing base framework is concerned with fault tolerance: rather than mediating or supporting all system operations, it is only invoked in crucial steps of the activity of protocols and applications.

Since we devised the Timely Computing Base model [2], we have taken systematic steps to validate it. In [3] we have shown how to solve a fundamental problem: to interface a payload system of any degree of asynchrony, to a synchronous subsystem as the TCB. We have also discussed the implementation of one of the application classes we propose (fail-safe) on the TCB. In a recent work [4] we show how to implement time-elastic applications on the TCB. Also recently [5] we introduced a paradigm for generic timing fault tolerance with replicated state machines, which is based on the existence of services provided by the TCB.

Implementing a TCB is a subject of its own. In [6], we describe a possible implementation of the TCB, based on RT-Linux, using Linux as the payload support system.

The paper is organized as follows. In the next section we present a brief survey of related work. In Section III we introduce the system model and failure mode assumptions. In Sections IV and V we introduce the Timely Computing Base model and architecture, and define the properties of the TCB services. Then, in Section VI, we describe the dependability problems introduced by uncertain timeliness in applications and define desirable properties that applications should enjoy in the presence of timing failures. We show that these properties can be secured when programming with a TCB. Finally, in Section VII we explain how to use the TCB to achieve varying degrees of dependability vis-a-vis timing failures, from fail-safe halting to timing error masking. The paper concludes with some considerations about future work.

II. Related Work

This problem is extremely relevant for real-time systems in general, in particular when there is the need for reconciling timeliness expectations with the uncertainty of the environment, known to be a complex task. Note that one should take a broad view on what real-time is. Many current networked applications, from multimedia rendering to interactive commercial or financial transactions, have real-time requirements which must not be hidden just because they are difficult to solve. We advocate that the debate should no longer be about hard or soft real-time, but on *correct* real-time, for given expectations about *synchrony* of the system vs. that of the environment.

The problem has been addressed in several previous works, in a number of different ways, which in fact motivated the idea behind this paper: the search of a generic paradigm for systems with uncertain temporal behavior. Chandra & Toueg have studied the minimal restrictions to asynchronism of a system that would let consensus or atomic broadcast be solvable in the presence of failures, by giving a failure detector which, should the system be 'good' (in fact, synchronous) for a long enough period, would be able to terminate[7]. Cristian & Fetzer have devised the timed-asynchronous model, where the system alternates between synchronous and asynchronous behavior, and where parts of the system have just enough synchronism to make decisions such as 'detection of timing failures' or 'fail-safe shutdown'[8]. Almeida & Veríssimo have devised the quasi-synchronous model where parts of the system have enough synchronism to perform 'real-time actions' with a certain probability[9]. Other two works have dealt with systems that are not completely asynchronous [10], [11]. They assumed a time-free liveness perspective, while studying the minimum guarantees for securing the safety properties of the system.

These works share a same observation: *synchronism (asynchronism) is not an invariant property of systems.* That is, it varies with time, and it varies with the part of the system being considered. However, each model has treated these asymmetries in its own way: some relied on the evolution of synchronism with time, others with space or with both. All these systems are what we may call *partially synchronous.*

In this issue, two papers also devote attention to solving problems with timeliness constraints, in spite of the possible non-deterministic or asynchronous nature of the environment. Although they focus on protocols, it is worthwhile discussing the underlying models: the way in which they characterize the environment in terms of synchrony assumptions is not the same, and this leads to different solutions and alternatives for constructing applications. However, together we share a few important aspects, which characterize this emerging area.

The paper on the Timewheel group communication system[12] is based on the Timed Asynchronous (TA) model. This model basically assumes that systems are fully asynchronous, except that they have clocks with a bounded drift rate. This is a crucial assumption since it allows to measure any time interval with a known and bounded error. In consequence, a service constructed over the TA model can be aware of untimely events, and avoid doing "bad" things in response to them. The model is very attractive in the sense that it makes very weak assumptions about the environment and may be very easily applied in most existing distributed environments. On the other hand, its fundamental weakness is related with the impossibility of guaranteeing the execution of real-time actions. If augmented with a synchronous component such as hardware watchdog, it can perform immediate shutdown, achieving fail-safety, when the system becomes untimely. This partially solves the problem and allows the implementation of fail-safe applications, when fail-safety does not require the timely execution of shutdown routines.

It is interesting to observe how the TA model compares with the TCB model. The TCB model casts different synchrony assumptions on architecture: the TCB, synchronous, but just for control actions; the payload part, with any degree of synchronism, maybe even asynchronous, for the applications. To a certain extent, the TA model augmented with hardware watchdogs can be viewed as a simplified instance of the TCB model: the watchdog is a synchronous architectural device, able to measure local durations, do timing failure detection locally, and perform a very simple kind of timely execution (flip a hardware register to stop the system, if a timing failure occurs). The distributed nature of the TCB services, namely the properties of perfect timing failure detection, and the ability to timely execute sporadic real-time functions, cannot be emulated even by the augmented version of the TA model. It can thus address a smaller range of applications than the TCB model.

In the approach followed in the Asynchronous Uniform Consensus paper[13], services or applications can be constructed assuming an asynchronous model augmented with a strong failure detector. This is a promising approach, that may be compared with the design of asynchronous applications on the TCB model, which rely on the TCB

timing failure detector. The "immersion" approach, as the authors call it, appears not to make any synchrony assumption about the environment, which is not exactly true. The real-time behavior of applications is obtained when the system infrastructure is able to guarantee time bounds. In fact, to determine if timeliness requirements can be met, it is necessary to know, or to establish, time bounds for essential variables such as message delivery delays, which, at some point, requires synchronous assumptions to be made. Then it is necessary to solve a real-time scheduling problem for the whole system. Meeting these assumptions makes the authors face the assumption-coverage binomial, just like any classical real-time systems design: if assumptions on the infrastructure fail, the system fails, no matter how weak the high-level, applicational assumptions may be.

Comparing with the TCB model, we point once more to the 'architectural' support of the latter. The TCB requires the previous construction of just a small part of the system with synchronous properties. This might be done using the same strategies and techniques used to solve the real-time schedulability problem of the "immersion" approach. However, it would have concerned: a small part of the infrastructure; a small and very well-defined (fairly predictable) set of services. Coverage of real-time assumptions for this design would be incomparably higher than for one being concerned with the system-wide infrastructure and exposed to the full system functionality. It is interesting to verify that the scenarios based on the "immersion" approach can be implemented on the TCB model. The strong failure detector would be implemented in the TCB, with the coverage conferred by its design. The rest would be implemented in the payload part, with the desired synchronous properties, checked by the failure detector.

III. Failure Assumptions

We assume a system model of participants or processes (we use both designations interchangeably) which exchange messages, and may exist in several sites or nodes of the system. Sites are interconnected by a communication network. *The system can have any degree of synchronism*, for example, bounds may or not exist for processing or communication delays, and when they do exist, their magnitude may be uncertain. Local clocks may not exist or may not have a bounded rate of drift towards real time. We assume the system to follow an omissive failure model, that is, *components only have timing failures*— and of course, omission and crash, since they are subsets of timing failures— no value failures occur. More precisely, they only have *late* timing failures. In order to prove our viewpoint about the effect of timing failures, we need to establish three things, and we will try to be as brief as possible.

- high-level system properties— which allow us to express functional issues, such as the type of agreement or ordering, or a message delivery delay bound;
- timed actions— which allow us to express the runtime behavior of the system in terms of time, and detect timing failures in a complete and accurate way;
- the relationship between high-level properties and timed actions— here, rather than establishing an elaborate calculus of time, we simply wish to make the point that some properties imply the definition of some timed actions, since the way this relation develops is crucial for application correctness.

High-Level Properties

We assume that an application, service, protocol, etc. is specified in terms of high-level safety and liveness properties. A liveness property specifies that a predicate $\mathcal{P}$ *will be true*. A safety property specifies that a predicate $\mathcal{P}$ *is always true*[14]. Informally, safety properties specify that wrong events never take place, whereas liveness properties specify that good events eventually take place.

A particular class of property is a *timeliness property*. Such a predicate is related with doing timely executions, in bounded time, and there are a number of informal ways of specifying such a behavior: "any message is delivered within a delay Td (from the send time)"; "task T must execute with a period of Tp"; "any transaction must complete within Tt (from the start)". The examples we have just given can be specified by means of time operators or time-bounded versions of temporal logic operators[15]. An appropriate *time operator* to define timeliness properties in our context is based on real time **durations**: P *within* T *from* t_0. The real time instant of reference t_0 does not have to be a constant (can be, e.g. 'the send time'), but even for relative durations, it is mapped onto an instant in the timeline for every execution. The interest of the 'from' part of the operator becomes evident just ahead, as a condition for defining timed actions and detecting timing failures. The *within/from* operator defines a duration, the interval $[t_0, t_0 + T]$ or $[t_0 - T, t_0]$, depending on whether T is positive or negative, such that predicate P becomes true at some point of the interval.

Note that the specifications exemplified above contain both a liveness and a safety facet. This happens very often with timeliness specifications. If we wanted to separate them and isolate the safety facet, we should for example say for the first one: "any message delivered is delivered within a delay Td from the send time". In this paper we are going to focus on safety properties, and as such we wish to distinguish between what we call logical safety properties, described by formulas containing logic and temporal logic operators, and what we call timeliness safety properties, containing time operators, as the one exemplified above. For simplicity, in the remainder of the paper we will call the former *safety* properties, and the latter *timeliness* properties.

Timed Actions

Once the service or protocol specified, the next step is to implement it. However it is implemented, securing different properties implies different steps. Securing timeliness properties certainly implies the assurance that things are done in a timed manner. Let us be more precise. A timeliness property is specified as a predicate, expressed by a within/from operator. In order to express the fulfilment

of that predicate in runtime, we introduce *timed action*, which we informally define as the execution of some operation within a known bounded time T. T is the allowed maximum duration of the action. Examples of timed actions are the release of tasks with deadlines, the sending of messages with delivery delay bounds, and so forth. For example, timeliness property "any message delivered is delivered within T from the send time" must be implemented by a protocol that turns each request `send_request`(p, M_i) *of message* M_i *addressed to* p, issued at real time $t_s(i)$, into a timed action: *execute the delivery of* M_i *at* p ***within*** T ***from*** $t_s(i)$.

Runtime enforcement of timed actions obeys to known techniques in distributed scheduling and real-time communication[16]. However, the problem as we stated it in the beginning is that bounds may be violated, and in consequence a systematic way of detecting timing failures must be devised. We base our approach on the observability of the termination event of a timed action, regardless of where it originated, and of the intermediate steps it took. That is, in order to be able to verify whether a timeliness property holds or not, or in other words, to detect timing failures, we need to follow the outcome of the timed actions deriving from the implementation of that property.

Take again the example of message delivery to a process p with bounded delay T. In order to detect violations of this property, we have to check whether every message M_i arrives within T from its send time $t_s(i)$. Each termination event (`delivery`) must take place at p by a real time instant that is definable for each execution, given $t_s(i)$. In the example, it is $t_e(i) = t_s(i) + T$. Generalizing, a timed action can be defined as follows:

Timed Action: *Given process p, real time instant* t_A*, interval* T_A*, and a termination event e, a timed action* $X(p, e, T_A, t_A)$ *is the execution of some operation, such that its termination event e takes place at p, within* T_A *from* t_A

It is clear now that the time-domain correctness of the execution of a timed action may be assessed if e is observable. If a timed action does not incur in a timing failure, the action is *timely*, otherwise, a timing failure occurs:

Timing Failure: *Given the execution of a timed action X, specified to terminate until real time instant* t_e*, there is a timing failure at p, iff the termination event takes place at a real time instant* $t_e^{'}$*,* $t_e < t_e^{'} \leq \infty$*. The delay of occurrence of e,* $Ld = t_e^{'} - t_e$*, is the lateness degree*

Wrapping up

The assumption that the system's components *only* have timing failures seems to imply that timeliness properties may, under certain circumstances, be violated, but safety properties should not. However, as we will show, this may not always be true unless adequate measures are taken.

In what follows, we introduce some simple notation. An application A is any computation that runs in the payload system, on top of the TCB (e.g. a consensus protocol, a replication management algorithm, a multimedia rendering application). Any application enjoys a set of properties $\mathcal{P}_A$, of which some are safety ($\mathcal{P}_S$) and others are timeliness ($\mathcal{P}_T$) properties[1]. An application may require an activity to be performed within a bounded duration $\mathcal{T}$, in consequence of which a component may perform one or several timed actions.

Recall the message delivery example of the previous section, where timeliness property $\mathcal{P}$, "any message delivered to any process is delivered within $\mathcal{T}$ from the time of the send request", must be fulfilled. Each message send request originates a timed action. Although t_e is different each time and p may sometimes be different, all timed actions generated in the course of the execution of the protocol relate to the same bound $\mathcal{T}$, and to fulfilling the same property $\mathcal{P}$. In fact, the computation of t_e is derived from $\mathcal{T}$, or ultimately, from $\mathcal{P}$.

Note that the actual form of the relation itself is very implementation dependent. For example, timed actions may also derive from the code implementing safety properties, if time is used as an artifact: algorithms for solving consensus problems have used timeouts even in time-free models (where timeliness properties do not exist). Whether this is an adequate approach will be discussed in the following sections.

It is important to retain the relations of timed actions to duration bounds, and finally, to properties. For example, by logging a history of the actual duration of the execution of all timed actions related with bound $\mathcal{T}$, we can build a distribution of the variable bounded by $\mathcal{T}$. By following all timed actions related with a timeout implied by property $\mathcal{P}$, we can detect and assess the effect of timing failures in the protocol implementing $\mathcal{P}$. We introduce just the necessary notation to reflect these relations:

- We define a history $\mathcal{H} = R_1, ..., R_n$, as a finite and ordered set of executions of timed actions, where each entry R_i of $\mathcal{H}$ is a tuple $\langle X_i, T(i), timely \rangle$. $T(i)$ is the *observed duration* of the execution of timed action X_i. *timely* is a Boolean which is true if the action was executed on time, or false if there was a timing failure.
- We denote $\mathcal{H}(\mathcal{T})$ as a history $\mathcal{H}$ where $\forall X \in \mathcal{H}$, the observed duration of X is related to bound $\mathcal{T}$ (through the termination bound t_e). The existence of the relation is important for the results of this paper. The exact relation is only relevant for the implementation.
- We denote $\mathcal{T}_\mathcal{P}$ a duration bound derived from property $\mathcal{P}$. That is, whenever the bound is established in the course of implementing an application with property $\mathcal{P}$ (be it timeliness or safety). In order to simplify our notation in the rest of the paper, we also represent this fact through the informal relation *derived from*, $\not{-d\rightarrow}$, that is, $\mathcal{T} \not{-d\rightarrow} \mathcal{P}$. For example, when $\mathcal{P}$ is a timeliness property defined in terms of $\mathcal{T}$ leading to an implementation code using a bound $\mathcal{T}_\mathcal{P} = f(\mathcal{T})$, for example for a timeout, we have $\mathcal{T}_\mathcal{P} \not{-d\rightarrow} \mathcal{P}$.
- Finally, we generalize to histories by denoting $\mathcal{H}(\mathcal{T}_\mathcal{P})$ as a history where all timed actions are related to a duration bound $\mathcal{T}_\mathcal{P}$ derived from property $\mathcal{P}$. We also represent this fact by $\mathcal{H} \not{-d\rightarrow} \mathcal{P}$.

[1] We remind the reader that they are in fact (logical-)safety and timeliness(-safety) properties

We will omit parenthesis and subscripts whenever there is no risk of ambiguity.

IV. The Timely Computing Base Model and Architecture

Given the above introductory sections, the fundamental problem is how can distributed computations reliably take time into account. Its hardness lies on the difficulty of performing reliable and timely processing and communication steps in a distributed system over an infrastructure with uncertain timeliness.

In Section III we have explained that the system model we follow is one of uncertain timeliness: bounds may be violated, or may vary during system life. Still, the system must be dependable with regard to time: it must be capable of timely executing certain functions or detecting the failure thereof. The apparent incompatibility of these objectives with the uncertainty of the environment may be solved if processes in the system have access to a subsystem that performs those (and only those) specific functions on their behalf. In the following sections, we start by defining the subsystem, which we call a *Timely Computing Base (TCB)*, we describe the architecture, and introduce the set of services to be provided by the TCB. The design principles of the (security) trusted computing base[1] helped us define similar objectives for the design of the TCB, in order to guarantee that: a) its operation is not jeopardized by the lack of timeliness of the rest of the system; b) its timeliness can be trusted:

- **Interposition -** the TCB position is such that no direct access to resources vital to timeliness can be made bypassing the TCB
- **Shielding -** the TCB construction is such that it is itself protected from faults affecting timeliness (e.g. delays in the outside system, or incorrect use of the TCB interface, do not affect TCB internal computations)
- **Validation -** the TCB complexity and functionality is reduced, such that it allows the implementation of verifiable mechanisms w.r.t. timeliness

The architecture of a system with a TCB is suggested by Figure 1. Whilst there is a generic, *payload* system over a global network, or *payload* network, the system admits the construction of say, a *control* part, made of local TCB modules, interconnected by some form of medium, the *control* network. The medium may be a virtual channel over the available physical network or a network in its own right. Processes execute on the several sites, in the payload part, making use of the TCB only when needed. For simplicity, in what follows we assume that there is one local timely computing base at every site. Configurations where TCBs exist only at a few sites of the system (e.g., the system servers) are possible and are currently being studied.

We now define the fault and synchronism model specific of the TCB subsystem (in Section III we defined the general, or payload system assumptions). We assume only crash failures for the TCB components, i.e. that they are fail-silent. Furthermore, we assume that the failure of a local TCB module implies the failure of that site, as seen from the other sites. This comes from the Interposition principle. We proceed by defining a few synchronism properties that should be enjoyed by any TCB.

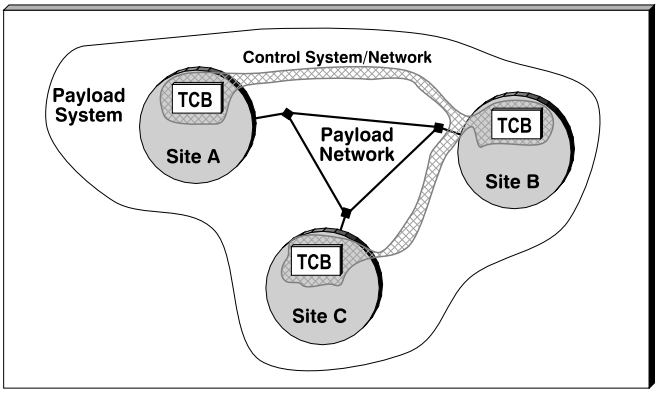


Fig. 1. The TCB Architecture.

Ps 1: There exists a known upper bound $T^1_{D_{max}}$ on processing delays

Ps 2: There exists a known upper bound $T^2_{D_{max}}$ on the rate of drift of local clocks

Property **Ps** 1 refers to the determinism in the execution time of code elements by a local TCB module. Property **Ps** 2 refers to the existence of a local clock in each TCB whose individual drift is bounded. This allows measuring local durations, that is, the interval between two local events. These clocks are internal to the TCB. Remember that the general system may or may not have clocks.

The TCB is distributed, composed of the collection of all local TCB modules interconnected by the control network, through which they exchange messages. Communication must be synchronous, as the rest of the TCB functions. Property **Ps** 3 completes the synchronism properties, referring to the determinism in the time to exchange messages among local TCB modules:

Ps 3: There exists a known upper bound $T^3_{D_{max}}$, on message delivery delays

The TCB subsystem, dashed in the figure, preserves, by construction, properties **Ps** 1 to **Ps** 3. For space reasons, the nature of the modules and the interconnection medium are outside the scope of this paper. Note however that there is a body of research on real-time operating systems and networks that has contributed to this subject[17], [18], [19], [20]. Interposition can be assured by implementing a native real-time system kernel that controls all the hardware and runs the TCB, besides supporting the actual operating system that runs on the site, or by installing the TCB in a separate appliance board or co-processor with a private network. Shielding can be achieved by scheduling the system in order to ensure that TCB tasks are hard real-time tasks, immune to timing faults in the other tasks.

Those principles also postulate the control of the TCB over the *control* network. The latter may or may not be based on a physically different infrastructure from the one supporting the *payload* network. The assumption of a restricted channel with predictable timing characteristics

(control) coexisting with essentially asynchronous channels (payload) is feasible in some of the current networks. Observe that the bandwidth required of the control network is much smaller than that of the payload network: local TCBs only exchange control messages. Besides, we said nothing about the magnitude of bound $T^3_{D_{max}}$, just that it must exist and be known. In a number of local area networks, switched networks, and even wider area networks, it is possible to give guarantees for the highest priority messages[21], [22], [23], [24]. In more demanding scenarios, one may resort to alternative networks (ISDN connection, GSM Short Message Service, Low Earth Orbit satellite constellations).

Finally, the TCB should be designed for validation, ensuring that it is simple and deterministic w.r.t. the mechanisms related to timeliness. As a proof of concept, we have indeed made an implementation prototype, using RT-Linux as the underlying kernel, with Linux as the payload operating system. This work, as well as measurements, are described in [6].

The specific protocols internal to the TCB, such as reliable multicast delivery, are not treated in this paper, since they depend on the particular implementation of the TCB. However, without loss of generality, we point the reader to a number of known reliable delivery protocols for synchronous systems on fail-silent models. Note also that any number of TCBs can fail, since under a fail-silent model, the distributed TCB continues to operate correctly until the last TCB module crashes. Defining bounds on the number of failed TCBs and/or hosts, and devising fault-tolerant TCB/payload system configurations concern work that can be performed by building on the results of this paper.

V. Services of the TCB

From now on, when there is no ambiguity, we refer to TCB to mean the 'distributed TCB', accessed by processes in a site via the 'local TCB' in that site. The TCB provides the following services: timely execution; duration measurement; timing failure detection. These services have a distributed scope, although they are provided to processes via the local TCB instantiations. Any service may be provided to more than one user in the system. For example, failure notification may be given to all interested users. We define below the properties of the services.

Timely Execution

TCB 1: Given any function F with an execution time bounded by a known constant T_{Xmax}, and a delay time lower-bounded by a known constant T_{Xmin}, for any execution of F triggered at real time t_{start}, the TCB does not start the execution within T_{Xmin} from t_{start}, and terminates F within T_{Xmax} from t_{start}

Timely Execution allows the TCB to execute arbitrary functions deterministically, given a feasible T_{Xmax}. Furthermore, the TCB can delay the execution by at least T_{Xmin}. Informally, the former ensures that something finishes before a deadline, whereas the latter ensures that something does not start before a liveline. The deterministic (time bounded) execution of a function after a delay (e.g., a timeout), can thus be achieved by this service.

Duration Measurement

TCB 2: There exist $T_{Dmin}, T_{D^2_{max}}$ such that given any two events e_s and e_e occurring in any two nodes, respectively at real times t_s and t_e, $t_s < t_e$, the TCB measures the duration between e_s and e_e as T_{se}, and the error of T_{se} is bounded by $(t_e - t_s)(1 - T_{D^2_{max}}) - T_{Dmin} \leq T_{se} \leq (t_e - t_s)(1 + T_{D^2_{max}}) + T_{Dmin}$

The measurement error has 1) a fixed component T_{Dmin} that depends on the measurement method, and 2) a component that increases with the length of the measured interval, i.e., with $t_e - t_s$. This is because the local clocks drift permanently from real-time as per Property **Ps** 2.

The measurement error can only be bounded a priori if the applications are such that we can put an upper limit on the length of the intervals being measured, say T_{INT}. This would bound the error by: $T_{Dmax} = T_{INT} T_{D^2_{max}} + T_{Dmin}$. Obviously, for events separated by less than T_{Dmax}, the measurement may be not significant (this is treated further in [25]).

When it is impossible or impractical to determine the maximum length of intervals, the clocks in the TCB must be externally synchronized. In that case it is guaranteed that at any time a TCB clock is at most some known α apart from real-time. In systems with external clock synchronization, the measurement error would be bounded by 2α. Incidentally, note that internal clock synchronization would not help here. Although given properties **Ps** 1–**Ps** 3 one could implement internal global time in the TCB, that would improve the error but would not bound it, and thus would not increase the power of the model. To keep the assumptions of the model to a minimum, we refrain from requiring synchronized clocks.

Another crucial service of the TCB is failure detection. We define a *Perfect Timing Failure Detector (pTFD)*, adapting the terminology of Chandra[7].

Timing Failure Detection

TCB 3: **Timed Strong Completeness:** There exists T_{Fmax} such that given a timing failure at p in any timed action $X(p, e, T_A, t_A)$, the TCB detects it within T_{Fmax} from t_e

TCB 4: **Timed Strong Accuracy:** There exists T_{Fmin} such that any timely timed action $X(p, e, T_A, t_A)$ that does not terminate within $-T_{Fmin}$ from t_e is considered timely by the TCB

Timed Strong Completeness can be understood as follows: "strong" specifies that any timing failure is perceived by all correct processes; "timed" specifies that the failure is perceived at most within T_{Fmax} of its occurrence. In essence, it specifies the detection latency of the TFD.

Timed Strong Accuracy can be understood under the same perspective: "strong" means that no timely action is wrongly detected as a timing failure; but "timed" qualifies what is meant by 'timely', by requiring the action to occur not later than a set-up interval T_{Fmin} before the detection threshold (the specified bound). In essence, it specifies the detection accuracy of the TFD. Note that the property is

valid if the local TCB does not crash until $t_e + T_{Fmax}$.

The majority of detectors known are *crash* failure detectors. For the sake of comparison, note that crash failures are particular cases of timing failures, where the process responsible for executing an action produces infinitely many timing failures with infinite lateness degree. Another perspective in taken in [26], where time is treated as a QoS parameter of crash failure detectors.

VI. Dependable and Timely Computing with a TCB

How can the TCB help design dependable and timely applications? We start by addressing an apparently controversial problem, the interaction of applications with the TCB: in the limit, how can an asynchronous, time-free application (payload) take advantage from a synchronous oracle (TCB)? While this is an issue we have tackled in another paper[3], it deserves a brief discussion for self-containment, so we leave here the intuition behind our approach. The first remark is that although the TCB may detect timing failures, nothing obliges an application to become aware of such failures. The second remark is that applications willing to use the TCB can only be as timely as allowed by the synchronism of the payload system. That is, the TCB does not make applications timelier, it just detects how timely they are, and helps them to remain correct. Finally, the TCB always detects a late event as a timing failure, within the terms of the definition of timing failure detector: it looks for the termination event of a timed action previously signalled to it. In consequence, applications take advantage from the TCB by construction, typically using it as a pacemaker, inquiring it (explicitly or implicitly) about the correctness of a set of past steps before proceeding to the next set of steps.

A. The Anatomy of Timing Failures

The next question to be asked is: *How can the TCB assist applications in the presence of failures?* A constructive approach consists in analyzing why systems fail in the presence of uncertain timeliness, and deriving sufficient conditions to solve the problems encountered, based on the behavior of applications and on the properties of the TCB.

As we explained earlier, we use 'application' to denote a computation in general, defined by a set of safety and timeliness properties $\mathcal{P}_A$. We also remind the reader that we consider a model where components only do late timing failures. In the absence of timing failures, the system executes correctly. When timing failures occur, there are essentially three kinds of problems, that we define and discuss below: *unexpected delay*; *contamination*; and *decreased coverage*.

The immediate effect of timing failures may be twofold: unexpected delay and/or incorrectness by contamination. We define **unexpected delay** as the violation of a timeliness property. That can sometimes be accepted, if applications are prepared to work correctly under increased delay expectations, or it can be masked by using timing fault tolerance (as we discuss in Section VII). However, there can also be **contamination**, that we define as the incorrect behavior resulting from the violation of safety properties on account of the occurrence of timing failures. This effect has not been well understood, and haunts many designs, even those supposedly asynchronous, but where aggressive timeouts and failure detection are embedded in the protocols[7], [27]. In fact, problems such as described in [28], [27] assume a simple dimension, when explained under the light of timing failures. These designs fail because: (a) although time-free by specification, they rely on time, often in the form of timeouts; (b) they are thus prone to timing failures (e.g., when timeouts are too short); (c) however, proper measures are not taken to counter timing failures (because they were not supposed to exist in the first place!); (d) in consequence, error confinement is not ensured, and sometimes timing failures contaminate safety properties[2]. A particular form of the contamination problem was described in the context of ordered broadcast protocols in [29].

In this paper, we give a generic definition of the problem, for a system with omissive failures. If the system has the capacity of *timely detecting timeliness violations*, then contamination can be avoided with adequate algorithm structure. To provide an intuition on this, suppose that the system does not make progress without having an assurance that a previous set of steps were timely. Now observe that "timely" means that the detection latency is bounded. In consequence, if it waits more than the detection latency, absence of failure indication means "no failure", and thus it can proceed. If an indication does come, then it can be processed before the failure contaminates the system, since the computation has not proceeded. The only consequence of this mechanism is an extra wait of the order of the detection delay (which is bounded, according to Property 3). A sufficient condition for absence of contamination is thus to confine the effect of timing failures to the violation of timeliness properties alone, specified by the following property:

No-Contamination: *Given a history $\mathcal{H}(\mathcal{T}_\mathcal{P})$ derived from property $\mathcal{P} \in \mathcal{P}_A$, $\mathcal{H}$ has no-contamination iff for any timing failure in any execution $X \in \mathcal{H}$, no safety property in $\mathcal{P}_A$ is violated.*

The reader will note that in the model of Chandra[7], the agreement algorithms have no-contamination, since their design is completely time-free, and all possible problems deriving from timing failures (such as "wrong suspicions") are encapsulated in the failure detector. Chandra then bases the reliability of his system on the possibility of implementing a given failure detector, but he does not discuss this implementation. In contrast, we define an architectural framework where aside of the payload part containing the algorithms or applications, a placeholder exists for the viable implementation of special services such as failure detection— the control part. One important advantage is generality of the programming model, by letting the payload system have any degree of synchrony. That is,

[2]As a matter of fact they may also contaminate liveness properties, by preventing progress. We do not explicitly discuss liveness properties in this paper.

we devise a single framework for correct execution of synchronous and asynchronous applications, of several grades that have been represented by partial models such as asynchronous with failure detectors, timed asynchronous, or quasi-synchronous. Or, in other words, from non real-time, through soft, mission-critical, to hard real-time.

Let us talk now about **decreased coverage** as the other effect of timing failures. Whenever we design a system under the assumption of the absence (i.e., prevention) of timing failures, we have in mind a certain coverage, which is the degree of correspondence between system timeliness assumptions and what the environment can guarantee. We define *assumed coverage* $P_{\mathcal{P}}$ of a property $\mathcal{P}$ as the assumed probability of the property holding over an interval of reference. This coverage is necessarily very high for timeliness properties of hard real-time systems, and may be somewhat relaxed for other realistic real-time systems, like mission-critical or soft real-time. Now, in a system with uncertain timeliness, the above-mentioned correspondence is not constant, it varies during system life. If the environment conditions start degrading to states worse than assumed, the coverage incrementally decreases, and thus the probability of timing failure increases. If on the contrary, coverage is better than assumed, we are not taking full advantage from what the environment gives. Both situations are undesirable, and this is a generic problem for any class of system relying on the assumption/coverage binomial[30]: if coverage of failure mode assumptions does not stay stable, a fault-tolerant design based on those assumptions will be impaired. A sufficient condition for that not to happen consists in ensuring that coverage stays close to the assumed value, over an interval of mission. Formally, we specify this condition by the following property:

Coverage Stability: *Given a history $\mathcal{H}(\mathcal{T}_{\mathcal{P}})$ derived from property $\mathcal{P} \in \mathcal{P}_{\mathcal{A}}$, with assumed coverage $P_{\mathcal{P}}$, $\mathcal{H}$ has coverage stability iff the set of executions contained in $\mathcal{H}$ is timely with a probability $p_{\mathcal{H}}$, such that $|p_{\mathcal{H}} - P_{\mathcal{P}}| \leq p_{dev}$, for p_{dev} known and bounded.*

The Coverage Stability property can be explained very easily. If we adapt our timeliness requirements say, by relaxing them when the environment is giving poorer service quality, and tightening them when the opposite happens, we maintain the actual coverage ($p_{\mathcal{H}}$) around a desirably small interval of confidence of the assumed coverage ($P_{\mathcal{P}}$). The interval of confidence p_{dev} is the measure in which coverage stability is ensured. Note that even if long term coverage stability is ensured, instantaneously the system can still have timing failures, as we have previously discussed.

The next step is to show that the TCB helps applications to secure coverage stability and no-contamination, despite the uncertainty of the environment. We propose three classes of applications, which can be combined in the solution of concrete problems. The *time-elastic* class is oriented to securing coverage stability under a varying environment, for example achieving what we have called *dependable QoS adaptation*[4]. The *fail-safe* class provides guidelines for ensuring the safe shutdown upon a timing failure that cannot be handled, and before contamination occurs. The *time-safe* class also aims at guaranteeing no-contamination, but this time by providing conditions for operation to continue. In all that follows, we consider the availability of the TCB services. Proofs of the lemmata and theorems given are shown in appendix.

B. Enforcing Coverage Stability with the TCB

We start with a definition of coverage stability for an application. Recall the definitions made in Section III, of histories and the 'derived-from' relation.

Definition 1: An application A has coverage stability iff all histories derived from properties of A have coverage stability:

$\forall \mathcal{P} \in \mathcal{P}_A,\ \forall \mathcal{H}\ s.t.\ \mathcal{H} \not\dashrightarrow \mathcal{P}$: $\mathcal{H}$ has coverage stability

Not all applications can benefit from the Coverage Stability property. Let us define a useful class that can indeed benefit.

Definition 2: **Time-Elastic Class ($\mathcal{T}\epsilon$) -** Given an application A, represented by a set of properties $\mathcal{P}_A$, A belongs to the time-elastic class $\mathcal{T}\epsilon$, iff none of the duration bounds derived from any property $\mathcal{P}$ of A are invariant[3]:

$A \in \mathcal{T}\epsilon \triangleq \forall \mathcal{P} \in \mathcal{P}_A,\ \forall \mathcal{T}\ s.t.\ \mathcal{T} \not\dashrightarrow \mathcal{P}$: $\mathcal{T}$ not an invariant

In practical terms, $\mathcal{T}\epsilon$ applications are those whose bounds can be increased or decreased dynamically, such as QoS-driven applications. We now show the conditions under which a $\mathcal{T}\epsilon$ application achieves coverage stability. We start by establishing the capability of the TCB to make histograms of the distribution of durations (service **TCB** 2) and thus gathering evidence about actual coverage.

Observe an example distribution (probability density function or *pdf*) of a duration T, depicted in Figure 2. If we focus on the pair $(\mathcal{T}_f, P_f)$ over curve A, P_f is the probability of finding executions lasting $\mathcal{T}_f$. Consequently, if $\mathcal{T}_f$ were an assumed upper bound, the shaded part of the distribution would contain executions with timing failures. The coverage of this assumption is thus given by the area of the distribution outside the shaded part. On the other hand, as we state in Lemma 1 below, the *pdf* will obviously have a certain error.

Lemma 1: Given a history $\mathcal{H}(\mathcal{T})$ containing a finite initial number of executions n_0, the TCB can compute the probability density function *pdf* of the duration T bounded by $\mathcal{T}$ and there exists a p_{dev0} known and bounded, such that for any $P = pdf(T)$, and p the actual probability of T, it is $|p - P| \leq p_{dev0}$

The *pdf* construction will be subjected to the TCB duration measurement error ($T_{DUR_{min}}$), and to extrapolation errors, given by the measure in which the n_0 samples are representative of the distribution.

On the other hand, in systems of uncertain timeliness, the *pdf* of a duration varies with time. For example, consider that curve A as depicted in Figure 2 represents the behavior of duration T during some initial period, and that from then on, the system's load increases such that executions become slower. Normally, if a sufficiently large inter-

[3]Invariant is taken in the sense of not being able to change during the application execution.

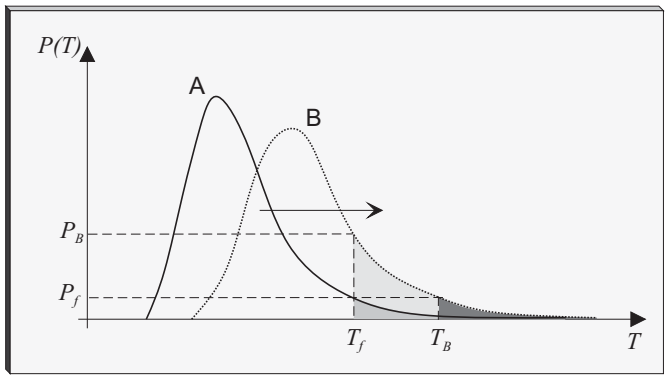


Fig. 2. Example variation of distribution $pdf(T)$ with a changing environment.

val of observation is considered, so as to damp short-term instability, what can be observed is a shift of the baseline *pdf*, as depicted in Figure 2 by curve B. However, more complex behaviors may arise. For the results of this paper, it is enough to establish that the long-term *pdf* error p_{dev} remains bounded, as we state in Lemma 2 below.

Lemma 2: Given $pdf_{i-1}(T)$, of duration T in $\mathcal{H}(\mathcal{T})$, and given any immediately subsequent n executions, the TCB can compute $pdf_i(T)$ and there exists a p_{dev} known and bounded, such that for any $P = pdf_i(T)$, and p the actual probability of T, it is $|p - P| \leq p_{dev}$

What Lemma 2 proposes is that a new set of samples composed of part of the old history, plus the new n executions, is representative (under the same error constraints discussed for Lemma 1) of the actual current distribution. Essentially, the lemma states that given n new samples, by using the adequate balance with a part of the previous distribution, the error p_{dev} remains bounded.

However, one also wishes to keep the error p_{dev} small in order to predict the probability of any T accurately, even with periods of timeliness instability. We are currently studying efficient functions to keep p_{dev} small, in the presence of large deviations[31]. The reader may wonder that it is impossible to predict the future, and thus using the *pdf* to assert timing variables for the future operation does not make sense. Note however: (a) some systems (with fixed assumptions) in fact predict the future off-line (e.g., by testing and assuming that the future will be like the present); (b) many QoS-adaptive systems adapt in an ad-hoc manner, making very coarse predictions of the future. We perform on-line adjustment of assumptions, which essentially innovates current state-of-the-art. When the system, despite unstable, is reasonably predictable, the environment does not keep changing abruptly. There is then a reasonable dependency of future histories on past histories during epochs of its operation, the error p_{dev} is small enough, and adaptation is effective in maintaining coverage stability. When the system is highly unstable, the error increases, perhaps to the point of exceeding the capacity of adaptation of the system. In our opinion, no other system would do better in these circumstances.

Next we show that every $\mathcal{H}$ can maintain coverage stability over time, by constructing a very simple algorithm which, assisted by the TCB, allows the execution of any application to adapt to the changing timeliness of the environment. We call it the **coverage-stabilization** algorithm, and it is given in Figure 3.

For any function f of an application A, with a measurable duration T_f, having an assumed bound $\mathcal{T}_f$ with probability P_f

// $pdf(T_f)$ exists and is kept updated by the TCB
// pdf^{-1} is the "inverse" of *pdf*
// $\mathcal{T}_f$ is initialized to some value
// h is the hysteresis for triggering of the algorithm
// c is the correction balance factor: $0 \leq c \leq 1$,
1- timeliness only; 0- coverage only
// TCB_setTo sets $\mathcal{T}_f$ to T_c and P_f to P_c

01 **foreach** f **do**
02 **when** $|pdf^{-1}(P_f) - \mathcal{T}_f| > h$ **do**
03 $T_c = \mathcal{T}_f + c(pdf^{-1}(P_f) - \mathcal{T}_f)$
04 $P_c = pdf(T_c)$
05 $TCB\_setTo(T_c, P_c; \mathcal{T}_f, P_f)$
06 **od**
07 **od**

Fig. 3. Coverage-Stabilization Algorithm.

Consider any function of an application A whose measurable duration T_f has a bound $\mathcal{T}_f$, assumed to hold with a probability P_f. For simplicity, and without loss of generality, we assume that any $P(T)$ on the right leg of the *pdf* represents the probability of T, as an upper bound, being met or exceeded. It is thus inversely proportional to the coverage of T, and can be used to represent it. Whenever the application executes the function, this is made known to the TCB in the form of a timed action with the adequate parameters. Typically, this action can be the sending of a message (which generates a remote event) or the execution of a local function, and its latest termination instant t_e is determined by the start instant and the bound $\mathcal{T}_f$ (*see* Section III). The TCB follows the execution of these actions, and in consequence the coverage-stabilization algorithm assumes that the TCB has created $pdf(T_f)$ and keeps it updated. We also assume that, given *pdf*, it is then easy to extract an approximation of value T for any P. We denote this operation by pdf^{-1}. In consequence, $pdf^{-1}(P_f)$ is the observed duration that holds with the desired probability P_f (it may have deviated from the assumed $\mathcal{T}_f$).

Consider that curve A in Figure 2 represents the initial steady-state operational environment for T_f: if $P = P_f$, then $T \simeq \mathcal{T}_f$. Whenever the TCB detects a significant variation in the environment (line 2), that is, when $pdf^{-1}(P_f)$ exceeds an interval h around $\mathcal{T}_f$, the algorithm is triggered. Parameter h introduces hysteresis in the algorithm operation, to prevent it from oscillating back and forth. The second parameter of the algorithm is c, the correction balance factor. c assumes any value between zero and one, and allows a combined correction of both timeliness and coverage.

Timeliness is adjusted in the following form: $\mathcal{T}_f$ is a variable of the application code from which all parame-

ters needed to substantiate the assumption in runtime (e.g., timeouts, multimedia protocol timing parameters, etc.) are derived. $\mathcal{T}_f$ is set to some initial value. The algorithm computes a new value T_c and automatically updates $\mathcal{T}_f$, influencing all parameters.

As for coverage, once having determined the new timeliness value, the algorithm computes $P_c = pdf(T_c)$. The reader should note that once the principle understood, nothing prevents us from implementing fancier schemes: a) replacing this function by one that computes the coverage value more accurately, given the area of the unshaded part; b) allowing to choose between computing timeliness first and then coverage or vice-versa.

These adjustments are computed in lines 3 and 4. Assume timeliness degrades: if $c = 1$, only timeliness is corrected, in order to preserve coverage at the originally desired value; if $c = 0$, the timeliness bound is preserved, and coverage expectations are decreased. We represent the update (line 5) by a function $TCB\_setTo()$, whereby the TCB automatically updates variables which are accessible to the application. The semantics of $TCB\_setTo()$ in our example is as simple as "setting $\mathcal{T}_f = T_c$ and $P_f = P_c$". However, a practical algorithm may have more elaborate semantics tuned to specific QoS applications.

To understand the mechanism, let us look again at Figure 2: the environment got slower, and there was a shift of *pdf* to the right (curve B). From the graphic, the value for the bound that still complies with $P = P_f$ is now T_B. If T_B is substituted in $\mathcal{T}_f$ (i.e., for $c = 1$, $T_c = T_B$), the application maintains the degree of coverage but admits to get slower. Alternatively, maintaining the original $t = \mathcal{T}_f$ timeliness bound implies that the application wishes to retain the same speed on average, but accepts a decreased coverage value, since the shaded area delimited by P_B is larger (i.e., for $c = 0$, $P_c = P_B = pdf(\mathcal{T}_f)$). For intermediate values of c the timeliness and probability values lie between these extremes.

Note that these remarks are also true when the environment performs better (faster and/or more reliable): the variables should get back to their original values as soon as possible, in order to take advantage from an improved QoS situation. However, they are only changed again when the measured bound deviates from $\mathcal{T}_f$ more than h, the hysteresis of the algorithm.

The discussion above allows us to introduce the first theorem about applications based on a TCB, whose proof is in the appendix.

Theorem 1: An application $A \in \mathcal{T}\epsilon$ using a TCB and the coverage-stabilization algorithm has coverage stability

C. Avoiding Contamination with the TCB

We start with a definition of no-contamination for an application.

Definition 3: An application A exhibits no-contamination iff all histories derived from properties of A have no-contamination:

$\forall \mathcal{P} \in \mathcal{P}_A$, $\forall \mathcal{H}$ *s.t.* $\mathcal{H} \not\xrightarrow{d} \mathcal{P}$: $\mathcal{H}$ has no-contamination

Firstly, we formalize the capability of failure detection of the TCB through services **TCB** 3 and **TCB** 4. We present two lemmata, and give the intuition in Figure 4, leaving the proof to the reader.

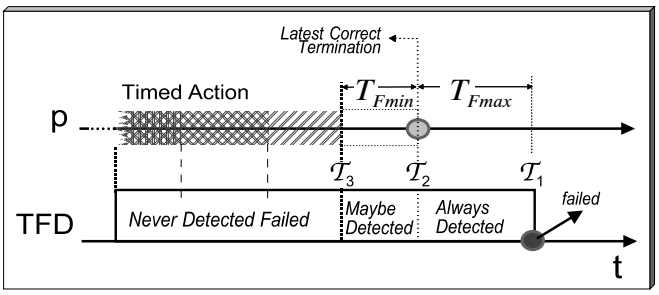


Fig. 4. Mechanism of Timing Failure Detection.

Lemma 3: Given any $\mathcal{H}(\mathcal{T}_2)$, and any $X \in \mathcal{H}$, if the execution of X fails the TCB detects it by $\mathcal{T}_1 = \mathcal{T}_2 + T_{Fmax}$

Lemma 4: Given any $\mathcal{H}(\mathcal{T}_2)$, and any $X \in \mathcal{H}$, the TCB never detects the execution of X as failed if it takes place by $\mathcal{T}_3 = \mathcal{T}_2 - T_{Fmin}$

These lemmata concern fundamental errors associated with timing failure detection in distributed systems, T_{Fmax} and T_{Fmin}, which cannot be eliminated. However, these can affect the system operation, for example by leaving way to contamination because of the detection latency. Next we show a technique to construct histories with no-contamination, which cancels the effect of these errors. It is called **error-cancellation**, and this rule only affects the design of applications in the early stage of definition of the required timing constraints. From then on, the TCB transparently ensures failure detection before contamination can occur.

Error Cancellation Rule

Given an application-level bound $\mathcal{T}_{APP}$ to be enforced:

- define the application logic (e.g., timeliness properties, timeouts) in terms of bound $\mathcal{T}_{APP}$
- design the environment in order to secure at least $\mathcal{T}_{ENV} = \mathcal{T}_{APP} - T_{Fmax} - T_{Fmin}$
- set the TCB failure detection logic to trigger at $\mathcal{T}_{TFD} = \mathcal{T}_{APP} - T_{Fmax}$

The error cancellation rule yields interesting results. For a real execution delay bound of $\mathcal{T}_{ENV}$, the application must work with a safety margin of at least $T_{Fmax} + T_{Fmin}$, because of the basic delay and inaccuracy of failure detection. However, for the adjusted bound $\mathcal{T}_{APP} = \mathcal{T}_{ENV} + T_{Fmax} + T_{Fmin}$, it is possible to simulate virtually instantaneous and accurate detection:

- any timing failure is detected (by the TCB) by $\mathcal{T}_{APP}$
- any timely execution is never detected as failed

In other words, the error cancellation rule proposes to use different values for each of the three above-mentioned aspects of building an application: given $\mathcal{T}_{ENV}$, the real bound met by the support environment, we use $\mathcal{T}_{TFD} = \mathcal{T}_{ENV} + T_{Fmin}$ as the failure detection threshold, and $\mathcal{T}_{APP} = \mathcal{T}_{TFD} + T_{Fmax}$ as the bound visible to the application.

Now we proceed with our treatment of contamination. Not all applications can benefit from the No-Contamination property. Intuitively, the least effective class that can indeed benefit is the fail-safe class, since it stops upon the first detected timing failure.

Definition 4: **Fail-Safe Class ($\mathcal{F}\sigma$) -** Given an application A, represented by a set of properties $\mathcal{P}_A$, A belongs to the fail-safe class $\mathcal{F}\sigma$, iff there is a state s_{FS} to which A can transit permanently, from any state s_i, at any time t_{FS}, such that for any safety property $\mathcal{P}$ of A, if $\mathcal{P}$ was true in time interval $[..., t_{FS}[$, it remains true for $[t_{FS}, ...[$:

$$A \in \mathcal{F}\sigma \triangleq \exists s_{FS} \forall s_i \forall t_{FS} : (s_i \overset{t_{FS}}{\rightarrow} s_{FS}) \wedge$$
$$\forall (\mathcal{P} \in \mathcal{P}_S \cap \mathcal{P}_A)((\forall t < t_{FS}, t \models \mathcal{P}) \Rightarrow (\forall t \geq t_{FS}, t \models \mathcal{P}))$$

In practical terms, $\mathcal{F}\sigma$ applications are those that can switch at any moment to a fail-safe state and remain there permanently, such that the system's safety properties remain valid. Note that whilst this seems to be a given in systems with hardware fail-safe switches, watchdogs, etc., it is not so in generic distributed systems with uncertain timeliness. Systems running under models that do not have the capabilities expressed by Property 1, of executing functions with guaranteed delays in any situation (in practice all but fully synchronous systems), cannot be guaranteed to be fail-safe. In our model, this ability, *for any asynchrony of the payload system*, is secured by running fail-safe shutdown routines *inside the TCB*. Plus, these routines can be coordinated and run locally in several hosts of the system. We now have the second theorem about applications based on a TCB.

Theorem 2: An application $A \in \mathcal{F}\sigma$ using a TCB has no-contamination

The discussion above allows us to capture the intuition behind this claim: by using the error cancellation rule, failures are not wrongly detected, and any failure is detected by $\mathcal{T}$; since prior to $\mathcal{T}$ there was no evidence of failure, if we switch the system to the fail-safe state immediately a failure is detected (i.e., t_{FS}), the corresponding history $\mathcal{H}(\mathcal{T})$ has no-contamination.

It is not interesting to halt an application upon the first timing failure, if better can be done. However, we know that not all applications can enjoy the No-Contamination property after timing failures occurring. We define a class that can.

Definition 5: **Time-Safe Class ($\mathcal{T}\sigma$) -** Given an application A, represented by a set of properties $\mathcal{P}_A$, A belongs to the time-safe class $\mathcal{T}\sigma$, iff no histories $\mathcal{H}(\mathcal{T}_\mathcal{P})$ are derived from any safety property in $\mathcal{P}_A$

$$A \in \mathcal{T}\sigma \triangleq \ \forall \mathcal{P} \in \mathcal{P}_A, \ \forall \mathcal{H} \ s.t. \ \mathcal{H} \not\overset{d}{\rightarrow} \mathcal{P} : \neg(\mathcal{P} \in \mathcal{P}_S)$$

In practical terms, $\mathcal{T}\sigma$ applications are those where: timeliness properties are clearly separated from logical safety ones; the code implementing safety properties does not depend on time (e.g. timeouts). These applications have a neat construction where timing failures are confined to just one of the three failure symptoms we studied in Section VI: unexpected delay. In consequence, one can implement mechanisms that address timing fault tolerance, without being concerned with the logical integrity of the application. We claim that a $\mathcal{T}\sigma$ application achieves no-contamination, in a theorem expressing the third and final facet of the timely computing base model.

Theorem 3: An application $A \in \mathcal{T}\sigma$ using a TCB has no-contamination

VII. Example Applications of the TCB Model

How practical are these application classes? Can one build real-life applications based on the TCB? Since we first devised the Timely Computing Base model [2], we have methodically addressed these issues. In [3] we have shown how to interface a payload system of any degree of asynchrony, to the TCB. We have also shown how a fail-safe application would look like, when implemented on the TCB. In a recent work [4] we have shown how to implement time-elastic applications on the TCB. Also recently [5] we introduced a paradigm for generic timing fault tolerance with replicated state machines, which is based on the existence of services provided by the TCB. This section discusses the design principles behind each of the application classes, under the perspective of timing fault tolerance.

Component timing failures, or system timing errors, in a fault-tolerance sense, can be handled in one of the following ways (in what follows, we use timing failure or timing error depending on the context): masking; detection and/or recovery. A generic approach to *timing fault tolerance*, that is, one that can use any or all of the methods above, requires the following basic attributes:

- timeliness- to act upon failures within a bounded delay;
- completeness- to ensure that failure detection is seen by all participants;
- accuracy- not to detect failures wrongly;
- coverage- to ensure that assumptions hold during the lifetime of the system (e.g. number of failures and magnitude of delays);
- confinement- to ensure that timing failures do not propagate their effect.

These attributes can be ensured by the TCB and its services. Furthermore, our model is able to distinguish between several mechanisms of timing failure: unexpected delay; contamination; decreased coverage. In consequence, one should be able to program real-time applications which have several degrees of dependability, despite the occurrence of timing errors: by detecting timing errors and still being able to recover by reconfiguration (e.g. by postponing a decision, by increasing the deadline, etc.); by detecting irrecoverable timing errors and doing a fail-safe shutdown; by masking independent timing errors with active replicas.

The TCB provides a framework for addressing all of these techniques, for any degree of synchronism of the payload system. We consider increasingly effective fault tolerance mechanisms, requiring combinations of the following attributes:

- timing failure detection (TFD)
- fail-safety ($\mathcal{F}\sigma$)
- time-safety ($\mathcal{T}\sigma$)
- time-elasticity ($\mathcal{T}\epsilon$)

- replication (REP)

TFD comes by default in the TCB. Fail-safety, time-safety, and time-elasticity correspond to application programming styles. Replication introduces a novel approach to real-time computing, by applying the classical concepts of fault tolerance to timing faults. As long as the latter are temporary and independent, the application remains timely, despite timing faults in some replicas.

In what follows, we explain the methodology for each of the above-mentioned classes of applications to achieve their objectives with the help of the TCB. For lack of space, this explanation will be necessarily summarized. However, when appropriate, we refer the reader to some example works.

A. Fail-safe Operation

By Theorem 2, any class of application with a **fail-safe** state can be implemented using a TCB. This is because the TCB has the ability of timely detecting timing failures. However, care must be taken with the setting of bounds, to avoid contamination. It is not enough to detect a failure and shutdown: the timely execution of these operations is crucial. The methodology should be the following:

- defining all timing parameters whose failure must be detected by the TCB;
- following the **error cancellation** rule (Section VI-C), so that failure detection can be done soon enough for the application to fail-safe before getting incorrect;
- configuring the switching to a fail-safe state: since all processes can be informed by the TCB of failure occurrences, it can be done in a controlled way in the case of distributed or replicated applications.

Several examples of applications with a fail-safe state can be encountered in the literature[32], [33], [34]. In the examples described in [33] and [34], the authors show how a fail-safe application can be implemented in the timed-asynchronous model. The detection of timing failures is done by using fail-aware services but the assurance of crucial safety properties also requires communication by time in the realm of the application. While fail-safety has been ensured in the works we know of on a case-by-case basis, the novelty of our approach is that instead of giving an implementation, we define a class ($\mathcal{F}\sigma$) of applications. Any implementation of an application of this class on the TCB can achieve no-contamination in the presence of timing failures, regardless of other aspects of its semantics. The TCB model also removes an ambiguity sometimes encountered in previous works: systems with unbounded execution time, state the capability of immediate (timely) fail-safe shutdown, an apparent contradiction. As we have seen, timely switching to the fail-safe state can be ensured by the TCB, no matter how asynchronous the payload system.

B. Reconfiguration and Adaptation

A more effective approach, other than simply halting the system, is trying to keep the application running. Again, this can be done on a case-by-case basis, or by defining classes of applications with given attributes, namely time-elasticity or time-safety, as we propose here.

It is very interesting to verify that a lot of work has been done recently in studying and proposing ways of dealing with QoS adaptation[35], [36], [37], [38], [39]. The point is that many applications are naturally able to provide services with different QoS, and so if there is any possibility of dynamically adapting this QoS to the changing environment, then the application should do that. However, the works we know of do not always follow a metrics that relates the QoS adaptation to the dependability issue: coverage stability. Others do establish thresholds for failure detection, but they cannot guarantee the precision of this detection, due to the lack of synchrony of the system.

By Theorem 1, applications of the **time-elastic** class, running under the TCB model provide a means to achieve QoS adaptation while maintaining coverage stability. The methodology follows what was laid down in Section VI-B:

- defining all timing parameters whose failure must be detected by the TCB;
- building a *pdf* for each of those parameters with the support of the TCB;
- applying the **coverage-stabilization** algorithm to relax or tighten the bounds.

Having this property means that it is possible to maintain the application running at a stable reliability level, despite environment changes to states worse than expected. This has a cost in the quality of the service the application can give. However, for many applications it is much better to keep running in a degraded mode (or with a lower QoS) than stopping, or worse, start having unexpected failures, as per the assumed coverage. Coverage stability is maintained in both directions: it also means that when the environment recovers, the application quickly comes back to the initial QoS.

The methodology implies the necessity of building *pdf*s and detecting QoS changes, which we have explained how to do in [4]. Essentially, the method we proposed consists in collecting an adequate number of measured durations (relative to a certain bound), use the latter to estimate an expected value $E(D)$ and a variance $V(D)$ and, finally, apply well known results from probability theory to determine time bounds that yield the desired coverage, using $E(D)$ and $V(D)$.

Finally, we address the effect of an individual timing failure. Again, there is room for ad-hoc solutions to this problem, but as shown by Theorem 3, an application possessing the **time-safety** property is guaranteed to be free from contamination when timing failures occur, regardless of the way it is implemented. The methodology was laid down in Section VI-C, and is simply based on:

- specifying applications such that safety and timeliness specifications are separated;
- having all applications be implemented in a modular fashion, such that the algorithmics related with the implementation of safety properties are time-free, that is, they do not generate timed actions;
- having the implementation of timeliness properties be

assisted by the TCB services when necessary.

C. Timing Error Masking

Finally, we discuss the issue of timing fault tolerance. This is an innovative idea which consists of using the **replication** and error processing principles of general fault tolerance, to timing faults (to keep the discussion clear, we now switch to system-level 'timing faults' and errors to designate component timing failures and their effect).

Several works have addressed the issue of fault tolerance in real-time systems[40], [37], [20], [41]. However, to our knowledge, the work presented in [42] was the first to address the issue of *timing* fault tolerance by software replication. Although designed around the quasi-synchronous system model, there is enough affinity with the work described here that it can very easily be explained under the light of the TCB framework. The methodology, that we detail below, is essentially the following:

- defining the replica set, choosing the replication degree to be greater than maximum expected number timing faults;
- defining timing parameters such that the TCB accurately detects faults;
- obtaining timely service by selecting timely replicas on a per service basis with the support of the TCB;
- applying the **time-safety** property to ensure no-contamination by late replicas
- applying the **time-elasticity** property to the whole replica set, to avoid total failure when the environment degrades for all replicas.

We give a brief explanation of how an application using the TCB can achieve timing fault tolerance. The issue is further treated in a recent paper [5]. The idea is to use a replication scheme for fault-tolerant components in order to mask independent timing errors affecting one or more of those components. The replication degree must be chosen in order to have more replicas than the maximum number of timing faults that can occur during a given protocol run. Up to this point, timing fault tolerance looks extremely simple. There are however a few subtle points that we discuss below.

A specific protocol is executed by the application to handle errors and to guarantee that the service will be timely executed. The simplest would be to use the first timely replica. However, timing error detection itself should be left to the TCB. Accurate **timing failure detection** and **time-safety** are important to ensure: selection of timely service by the timely replicas; no-contamination by the late replicas. Otherwise, upon the first fault, a replica would have to shut down, and the replication degree would quickly be lost. This is one of the subtle aspects of replication for timing fault tolerance, sometimes ignored in other works. On the other hand, a replica set should desirably preserve long-term coverage of the fault assumptions. A replica set is a good approach to mask transient timing errors in individual replicas. However, when the environment degrades for all replicas, timing fault rate increases, and this may reach a point where all replicas have faults in an execution. This is highly undesirable because it means the complete failure of the fault tolerance mechanisms.

The only way to counter this problem is by applying the **time-elasticity** property to the whole replica set. The reader will note that: either the application cannot withstand this increased delay, meaning it is not time-elastic, and then it must be shut-down; or, if it can, there must be a means to maintain both the spare coverage (number of operational replicas) and the individual coverage of each replica (probability of being timely). This is the second subtle issue to timing fault tolerance by replication. This obviously requires the application to switch to another long-term operational envelope, where the timing error being masked concerns a bound, which is longer than the previous one[40]. However, the coverage guarantees for replica set have been recovered.

VIII. Conclusion

We proposed a model to build dependable and timely applications exhibiting varying degrees of timing fault tolerance, under several synchrony models. In essence, our paper is an attempt to provide a unifying solution for a problem that has been addressed by several research teams: how to reconcile the need for synchrony, with the temporal uncertainty of the environment.

We have proposed an architectural construct that we have called Timely Computing Base (TCB), capable of executing timely functions, however asynchronous the rest of the system may be. Then, we postulated a few necessary services for the TCB to fulfil its role: timely execution, duration measurement, timing failure detection. The synchrony of the TCB does not mandatorily imply special hardware. Rather, the paradigm is based on the observation that synchronism is not a homogeneous property: it varies with time and space in a system. The quality of the synchrony of the TCB (speed, precision) is the only thing that may be improved by special components.

We introduced an innovative analysis of the effect of timing failures on application correctness. Besides the obvious effect of delay, we identified a long-term effect, of decreased coverage of assumptions, and an instantaneous effect, of contamination of other properties. Even when delays are allowed, any of these effects can lead to undesirable behavior of a system. Separating the mechanisms of timing failure into these three components— delay, uncoverage and contamination— has allowed us to introduce classes of applications that deal with combinations of the former, achieving varying degrees of dependability, when assisted by a TCB: fail-safe, which exhibits correct behavior or else stops in fail-safe state; time-elastic, which exhibits coverage stability; and time-safe, which exhibits no-contamination.

We showed that the computational model based in the TCB is generic enough to:

- support applications (algorithms, services, etc.) based on any synchrony of the payload system, from asynchronous to synchronous; from a system design viewpoint, this is the same as saying from non real-time to hard real-time;
- support several fault tolerance and performability adaptation techniques, such as fail-safe halting, error detection

and reconfiguration, QoS adaptation, or replication management;
• provide a suitable architectural framework for the common explanation of the several known models of partial synchrony.

The opportunities for future exploration are manifold. With regard to the first item, consider a hard real-time system for critical applications. The TCB might provide the generic and formal framework for the clear-cut separation between application and support system functionalities, which is sometimes hard to do in ad-hoc designs as we know them today: fail-safe shutdown, self-checking, watchdogs, etc. With regard to the last item, it remains to be seen whether the TCB, by a refinement of its failure detector structure, can provide the basis for a common explanation of what has up to now been represented by partial models such as asynchronous with failure detectors, timed asynchronous, or quasi-synchronous[43].

Acknowledgments

We wish to warmly acknowledge the contributions of Christof Fetzer and Rick Schlichting, who read earlier manuscripts and provided helpful comments. The comments of anonymous reviewers have considerably improved the paper.

Appendix

Informal proof of some results

Lemma 1: Given a history $\mathcal{H}(\mathcal{T})$ containing a finite initial number of executions n_0, the TCB can compute the probability density function *pdf* of the duration T bounded by $\mathcal{T}$ and there exists a p_{dev0} known and bounded, such that for any $P = pdf(T)$, and p the actual probability of T, it is $|p - P| \leq p_{dev0}$

Proof sketch:
From service **TCB** 2, the TCB is capable of measuring all durations $T(i)$ in any history $\mathcal{H}$. For a history $\mathcal{H}(\mathcal{T})$, $T(i)$ are the *observed durations* representing the *specified duration bound* $\mathcal{T}$. For any given actual distribution of T, a discrete probability distribution function *pdf* built with n_0 observations $T(i)$, subjected to the TCB duration measurement error $T_{DUR_{min}}$, represents T with an error of p_{dev0}[44]. □

Lemma 2: Given $pdf_{i-1}(T)$, of duration T in $\mathcal{H}(\mathcal{T})$, and given any immediately subsequent n executions, the TCB can compute $pdf_i(T)$ and there exists a p_{dev} known and bounded, such that for any $P = pdf_i(T)$, and p the actual probability of T, it is $|p - P| \leq p_{dev}$

Proof sketch:
From Lemma 1, the TCB is capable of recomputing $pdf(T)$ for any additional n (consider $n'_0 = n_0+n$). Then, *pdf* must preserve a bounded error, event in the presence of large deviations: for any T with actual probability of p, and any n, there is a subset n_{i-1} of the last executions of the original history $\mathcal{H}$, and an adequate function[31], such that $P = pdf_i(T)$ recomputed with the $n_{i-1} + n$ observations has a bounded error p_{dev} from p. □

Theorem 1: An application $A \in \mathcal{T}\epsilon$ using a TCB and the coverage-stabilization algorithm has coverage stability

Proof:
From Definition 2, we see that any $\mathcal{T}$ derived from any property $\mathcal{P}$ of A, is not an invariant, and as such, the correctness of A does not depend of the absolute value of any $\mathcal{T}$, but rather on a 'correct' value of the latter. In consequence, any $\mathcal{T}$ can be modified, namely with the intent of achieving coverage stability. From lemmata 1 and 2, and the discussion that followed about the coverage-stabilization algorithm, we see that for any property $\mathcal{P}$ with assumed coverage $P_{\mathcal{P}}$, any history $\mathcal{H}(\mathcal{T}_{\mathcal{P}})$ derived from $\mathcal{P}$ has coverage stability. By induction on every $\mathcal{P} \in \mathcal{P}_A$, and by Definition 1, application A has coverage stability. □

Theorem 2: An application $A \in \mathcal{F}\sigma$ using a TCB has no-contamination

Proof:
Consider a timing failure in $\mathcal{H}(\mathcal{T})$ derived from timeliness property $\mathcal{P} \in A$. By Lemmata 3 and 4, we see that for any bound $\mathcal{T}$ established as per the error cancellation rule, failures are not wrongly detected, and any failure of $X \in \mathcal{H}(\mathcal{T})$, is detected until $\mathcal{T}$ has elapsed. Consider this instant as corresponding to t_{FS} at the latest, when fail-safe switching is done, by Definition 4. Then, if up to instant t_{FS} (corresponding to detection threshold $\mathcal{T}$) there was no evidence of failure in $\mathcal{H}(\mathcal{T})$, the system was correct. Since the system halts at this point, the corresponding history $\mathcal{H}(\mathcal{T})$ has no-contamination, by the respective definition. The effect of the failure could not propagate, and in consequence, all other histories have no-contamination. In consequence, any safety property that was true before t_{FS}, will remain true after halting, by Definition 4. Then, by Definition 3, application A has no-contamination. □

Theorem 3: An application $A \in \mathcal{T}\sigma$ using a TCB has no-contamination

Proof:
If $A \in \mathcal{T}\sigma$, then no histories are derived from safety properties. In consequence, any history $\mathcal{H}(\mathcal{T})$ has to have been derived from a timeliness property. By definition of no-contamination, such histories have no-contamination. By Definition 3, application A has no-contamination. □

References

[1] M. Abrams, S. Jajodia, and H. Podell, Eds., *Information Security*, IEEE CS Press, 1995.
[2] P. Veríssimo and A. Casimiro, "The timely computing base," DI/FCUL TR 99–2, Department of Computer Science, University of Lisboa, Apr. 1999.
[3] P. Veríssimo, A. Casimiro, and C. Fetzer, "The timely computing base: Timely actions in the presence of uncertain timeliness," in *Proceedings of the International Conference on Dependable Systems and Networks*, New York City, USA, June 2000, pp. 533–542.
[4] A. Casimiro and P. Veríssimo, "Using the timely computing base for dependable qos adaptation," in *Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems*, New Orleans, USA, Oct. 2001, pp. 208–217.
[5] A. Casimiro and P. Veríssimo, "Generic timing fault tolerance using a timely computing base," in *Proceedings of the International Conference on Dependable Systems and Networks*, washington, D.C., USA, June 2002, To appear.
[6] A. Casimiro, P. Martins, and P. Veríssimo, "How to build a timely computing base using real-time linux," in *Proceedings of*

the 2000 IEEE International Workshop on Factory Communication Systems, Porto, Portugal, Sept. 2000, pp. 127–134.

[7] T. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM*, vol. 43, no. 2, pp. 225–267, Mar. 1996.

[8] F. Cristian and C. Fetzer, "The timed asynchronous system model," in *Proceedings of the 28th Annual International Symposium on Fault-Tolerant Computing*, Munich, Germany, June 1998, pp. 140–149.

[9] P. Veríssimo and C. Almeida, "Quasi-synchronism: a step away from the traditional fault-tolerant real-time system models," *Bulletin of the Technical Committee on Operating Systems and Application Environments (TCOS)*, vol. 7, no. 4, pp. 35–39, Winter 1995.

[10] D. Dolev, C. Dwork, and L. Stockmeyer, "On the minimal synchronism needed for distributed consensus," *24th Annual Symposium on Foundations of Computer Science*, Nov. 1983.

[11] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *Journal of the ACM*, vol. 35, no. 2, pp. 288–323, Apr. 1988.

[12] S. Mishra, C. Fetzer, and F. Cristian, "The timewheel group communication system," *IEEE Transactions on Computers, Special Issue on Asynchronous Real-Time Distributed Systems*, 2002.

[13] J.-F. Hermant and G. Le Lann, "Asynchronous uniform consensus in real-time distributed systems," *IEEE Transactions on Computers, Special Issue on Asynchronous Real-Time Distributed Systems*, 2002.

[14] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*, Springer, New York, 1992.

[15] R. Koymans, "Specifying real-time properties with metric temporal logic," *Journal of Real-Time-Systems*, vol. 2, no. 4, pp. 255–299, Nov. 1990.

[16] A. Burns and A. Wellings, *Real-Time Systems and Programming Languages*, International Computer Science Series. Addison-Wesley, 3rd edition, 2001.

[17] A. Burns, "A Framework for Building Real-time Responsive Systems," in *Proceedings of the 1st International Workshop on Responsive Computer Systems*, Golfe-Juan, France, Oct. 1991, ONR/INRIA, pp. 6–9.

[18] F. Jahanian, "Fault tolerance in embedded real-time systems," *LNCS*, vol. 774, pp. 237–249, 1994.

[19] E. Douglas Jensen and J. Duane Northcutt, "Alpha: A non-proprietary os for large, complex, distributed real-time systems," in *Proceedings of the IEEE Workshop on Experimental Distributed Systems*, Huntsville, Alabama, USA, Oct. 1990, pp. 35–41.

[20] H. Kopetz, R. Zainlinger, G. Fohler, H. Kantz, P. Puschner, and W. Schutz, "An engineering approach towards hard real-time system design," *Lecture Notes in Computer Science*, vol. 550, pp. 166–188, 1991.

[21] M. de Prycker, *Asynchronous Transfer Mode: Solution For Broadband ISDN*, Prentice-Hall, 3rd edition, 1995.

[22] R. Brand, "Iso-Ethernet: Bridging the gap from WAN to LAN," *Data Communications*, July 1995.

[23] R. Braden, Ed., L. Zhang, S. Berson, S. Herzog, and S. Jamin, "RFC 2205: Resource ReSerVation Protocol (RSVP) — version 1 functional specification," Sept. 1997.

[24] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, "RTP: A transport protocol for real-time applications," Tech. Rep. RFC 1889, Audio-Video Transport Working Group, Jan. 1996.

[25] P. Veríssimo, "Ordering and timeliness requirements of dependable real-time programs," *Journal of Real-Time Systems*, vol. 7, no. 2, pp. 105–128, Sept. 1994.

[26] W. Chen, S. Toueg, and M. Aguilera, "On the quality of service of failure detectors," in *Proceedings of the International Conference on Dependable Systems and Networks*, New York City, USA, June 2000, pp. 191–200.

[27] T. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost, "On the impossibility of group membership," in *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing*, Philadelphia, USA, May 1996, pp. 322–330.

[28] E. Anceaume, B. Charron-Bost, P. Minet, and S. Toueg, "On the formal specification of group membership services," Tech. Rep. RR-2695, INRIA, Rocquencourt, France, Nov. 1995.

[29] A. Gopal and S. Toueg, "Inconsistency and contamination (preliminary version)," in *Proceedings of the 10th Annual ACM Symposium on Principles of Distributed Computing*, Montreal, Québec, Canada, Aug. 1991, pp. 257–272.

[30] D. Powell, "Failure mode assumptions and assumption coverage," in *Digest of Papers, The 22nd International Symposium on Fault-Tolerant Computing Systems*, Boston, USA, July 1992, pp. 386–395.

[31] H. Wold, Ed., *Bibliography on Time Series and Stochastic Processes*, Oliver and Boyd, London, 1965.

[32] Lubaszewski and Courtois, "A reliable fail-safe system," *IEEE Transactions on Computers*, vol. 47, 1998.

[33] C. Fetzer and F. Cristian, "Fail-awareness in timed asynchronous systems," in *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing*, Philadefphia, USA, May 1996, pp. 314–321a.

[34] C. Fetzer and F. Cristian, "Fail-awareness: An approach to construct fail-safe applications," in *Proceedings of the 27th Annual International Fault-Tolerant Computing Symposium*, Seattle, Washington, USA, June 1997, pp. 282–291.

[35] T. Abdelzaher and K. Shin, "End-host architecture for qos-adaptive communication," in *Proceedings of the 4th IEEE Real-Time Technology and Applications Symposium*, Denver, Colorado, USA, June 1998.

[36] C. Almeida and P. Veríssimo, "Timing failure detection and real-time group communication in *quasi-synchronous* systems," in *Proceedings of the 8th Euromicro Workshop on Real-Time Systems*, L'Aquila, Italy, June 1996.

[37] M. Cukier, J. Ren, C. Sabnis, D. Henke, J. Pistole, W. H. Sanders, D. E. Bakken, M. E. Berman, D. A. Karr, and R. E. Schantz, "Aqua: An adaptive architecture that provides dependable distributed objects," in *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems (SRDS'98)*, West Lafayette, Indiana, USA, Oct. 1998.

[38] B. Li and K. Nahrstedt, "A control-based middleware framework for quality of service adaptations," *IEEE Journal of Selected Areas in Communications, Special Issue on Service Enabling Platforms*, vol. 17, no. 9, pp. 1632–1650, Sept. 1999.

[39] I. Foster, V. Sander, and A. Roy, "A quality of service architecture that combines resource reservation and application adaptation," in *Proceedings of the Eighth International Workshop on Quality of Service*, Westin William Penn, Pittsburgh, USA, June 2000, pp. 181–188.

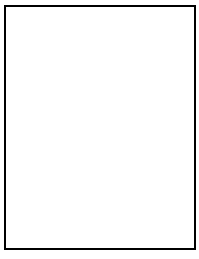
[40] P. Veríssimo, P. Barrett, P. Bond, A. Hilborne, L. Rodrigues, and D. Seaton, "The Extra Performance Architecture (XPA)," in *Delta-4 - A Generic Architecture for Dependable Distributed Computing*, D. Powell, Ed., ESPRIT Research Reports, pp. 211–266. Springer Verlag, Nov. 1991.

[41] H. Zou and F. Jahanian, "Real-time primary-backup (RTPB) replication with temporal consistency guarantees," Technical Report CSE-TR-356-98, University of Michigan Department of Electrical Engineering and Computer Science, Feb. 13, 1998.

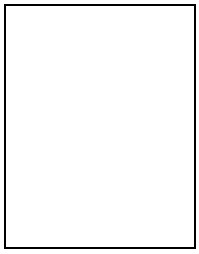
[42] C. Almeida and P. Veríssimo, "Using light-weight groups to handle timing failures in *quasi-synchronous* systems," in *Proceedings of the 19th IEEE Real-Time Systems Symposium*, Madrid, Spain, Dec. 1998.

[43] A. Casimiro, F. Cristian, C. Fetzer, and P. Veríssimo, "Private communications," Sept. 1998.

[44] W. Feller, *An Introduction to Probability Theory and its Applications*, John Wiley & Sons, 2 edition, 1971.



Paulo Veríssimo got his PhD from the Technical University of Lisboa, and is a Professor of the Department of Informatics, University of Lisboa Faculty of Sciences, where he leads the Navigators research group. He belongs to the Executive Board of the CaberNet European Network of Excellence, and is Vice-Chair of the IEEE T.C. on Fault Tolerance. He served as program co-chair of the IEEE DSN 2001 conference, and is an associate editor for the Telecommunications Systems Journal (Baltzer). He is author of more than 90 refereed publications in international scientific conferences and journals, and over a 100 technical reports. He is co-author of four books on distributed systems and dependability. He is currently interested in fault and intrusion tolerance, and dependable adaptation in real-time systems.



António Casimiro graduated in Electrical and Computer Engineering in 1991 at the Lisboa Higher Institute of Technology (IST, Instituto Superior Técnico) and got his Msc degree in 1995, also at IST. Since then he is a teaching assistant of the Department of Informatics, University of Lisboa Faculty of Sciences, where he is a member of the Navigators research group. He is currently finishing his PhD thesis, in the area of fault-tolerant and real-time distributed systems and he is particularly interested in dependable adaptation in real-time systems. He is a student member of IEEE.