The Timely Computing Base

Paulo Veríssimo
António Casimiro

DI–FCUL TR–99–2

May 1999

Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa
Campo Grande, 1700 Lisboa
Portugal

Technical reports are available at http://www.di.fc.ul.pt/biblioteca/tech-reports. The files are stored in PDF, with the report number as filename. Alternatively, reports are available by post from the above address.

The Timely Computing Base

Paulo Veríssimo
pjv@di.fc.ul.pt
FC / UL*

António Casimiro
casim@di.fc.ul.pt
FC / UL

Abstract

Real-time behavior is materialized by *timeliness* specifications, which in essence call for synchronous system models. However, systems many often rely on large-scale, unpredictable and unreliable infrastructures, that suggest the use of asynchronous models. Several models in between have addressed these antagonistic aims, each in its own way. We propose an architectural construct that addresses the problem in a generic way. We assume the existence of a component that is capable of executing timely functions, however asynchronous the rest of the system may be. This component can be used by other components to execute timely services. There is a certain analogy to the trusted computing base principle used in security. We call it the Timely Computing Base, TCB. In this paper, we show that a TCB can be used to build dependable and timely applications exhibiting varying degrees of timing fault tolerance, under several synchrony models.

1 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. A large number of the emerging services have interactivity or mission-criticality requirements, which are best translated into requirements for fault-tolerance and real-time. That is, service must be provided on time, either because of dependability constraints (e.g. air traffic control, telecommunication intelligent network architectures), or because of user-dictated quality-of-service requirements (e.g. network transaction servers, multimedia rendering, synchronized groupware).

This is a real-time systems behavior. Formally, it is materialized by *timeliness* specifications, which in essence call for a synchronous system model. Under this model there are known bounds for the above-mentioned 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. As examples, we mention reliable real-time communication, real-time scheduling and real-time distributed replication management.

*Faculdade de Ciências da Universidade de Lisboa. Bloco C5, Campo Grande, 1700 Lisboa - Portugal. Tel. +(351) 1 750 0087 (secretariat); +(351) 1 750 0103 (direct) (office). Fax +(351) 1 750 0084. Navigators Home Page: http://www.navigators.di.fc.ul.pt.

However, large-scale, 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 tolerated.

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 large-scale infrastructures, since they have poor 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 a framework that describes the problem in a generic way. We call it the **Timely Computing Base (TCB)** model. 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, if the system is distributed, the TCB provides such services in a distributed way.

Classically, a 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: decreased coverage; unexpected delay; contamination. 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[2], 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 obvious: the timely computing base can, to a given extent, ensure correct and timely operation of applications running on environments with untimely behavior.

The paper is organized as follows. In the next section we present a brief survey of related work. In section 3 we introduce the system model and failure mode assumptions. In Sections 4 and 5 we introduce the Timely Computing Base Model, and define the properties of the TCB services. Then, in Section 6, 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 prove that these properties can be secured when programming with a TCB. Finally, in Section 7 we explain how to use the TCB to achieve varying degrees of dependability vis-a-vis timing faults, from fail-safe halting to timing error masking. The paper concludes with some considerations about future work.

2 Related Work

This problem is extremely relevant for large-scale real-time systems, since reconciling timeliness expectations with the uncertainty of the environment is known to be a complex task. 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 'synchronous' for a long enough period, would be able to terminate[13]. 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'[15]. We have devised the quasi-synchronous model where parts of the system have enough synchronism to perform 'real-time actions' with a certain probability[36].

These works share a same observation: synchronism or asynchronism are not homogeneous properties of systems. That is, they vary with time, and they vary 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. Other two works have dealt with systems that are not completely asynchronous, i.e. exhibiting *partial synchrony* [17, 18]. They assumed a time-free liveness perspective, while studying the minimum guarantees for securing the safety properties of the system.

3 Failure Model

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, that is, if bounds exist for processing or communication delays, their magnitude may be uncertain or not known. 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— and 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 simple as possible:

- high-level system properties— which allow us to express functional issues, such as the type of agreement, 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 a system 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 $\mathcal{P}$ *will always be true*[30]. Informally, safety properties specify that wrong events never take place, whereas liveness properties specify that good events eventually take place. A particular class of safety property is a *timeliness property*, which specifies that a predicate $\mathcal{P}$ *will be true at infinitely many instants of real time*. 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: "task T must execute with a period of Tp"; "any message is delivered within a delay Td"; "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[27]. An appropriate *time operator* to define timeliness properties in this context is based on real time **durations**: P *within* T *from* t_0 (real time instant of reference). 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 will become true at some point of the interval. A negative value of the duration is useful to define a set-up interval, before the reference instant. On the other hand, the negation of a formula with the within / from operator is useful to enforce a "not before" or *liveline* condition for a predicate[37]. Unlike simpler operators such as "terminate at", this operator captures all relevant notions of time-related operations without ambiguity.

In this paper, we will distinguish between what we call logical safety properties, described by formulas containing logic and temporal operators, and what we call timeliness safety properties, containing time operators. For simplicity, we will call the former safety properties, and the latter timeliness properties.

Timed Actions

A timeliness property is specified as a predicate, expressed by a within / from operator, which holds at infinitely many instants of real time. Timeliness properties belong to the class of properties where, in order to verify their correctness, we need to observe every individual execution[30]. In order to be able to verify whether a predicate holds or not, or in other words, to detect timing failures, we need to follow the runtime (real-time) behavior that derives from the implementation of that property.

To that purpose, we introduce *timed action*, which we define as the execution of some operation within a known bounded time T from its start. 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 to any process is delivered within T from the time of the send request" 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, into a timed action: *execute the delivery of* M_i *at* p **within** T **from** t_s.

Note that in a distributed system some operations may start on one node and end on another, such as message delivery. Furthermore, several factors may contribute to the delay budget of a real-time activity[1]. Runtime enforcement of bounds on these delays obeys to known techniques in distributed scheduling and real-time communication[33, 10]. 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.

Take again the example of message delivery to a process p with bounded delay T: the termination event (`delivery`) must take place at p by a real time instant t_e that is a priori definable for each execution, upon its triggering (`send`), and which in the example is $t_e = t_s + T$. Generalizing, a timed action can be defined as follows:

Timed Action - *Given process p, event e, and real time instant t_e, a timed action $X(p, e, t_e)$ is the execution of some operation, such that its termination event e takes place at p, at a real time instant $t \leq t_e$*

The time-domain correctness of the execution of a timed action may be observed 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 - *There is a timing failure at p, iff given the execution of a timed action $X(p, e, t_e)$, e takes place at a real time instant $t_e^{'}$, $t_e < t_e^{'} \leq \infty$. The lateness degree is the delay of occurrence of e, $Ld = t_e^{'} - t_e$*

Wrapping up

The assumption that the system's components *only* have timing failures means that timeliness properties may, under certain circumstances, by 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 a computation in general (e.g. a consensus protocol, a replication management algorithm, a multimedia rendering application). Any application implements a set of properties $\mathcal{P}_A$, of which some are (logical) safety ($\mathcal{P}_S$) and others are timeliness (safety) ($\mathcal{P}_T$) properties. An implementation may require an activity to be performed within a bounded duration $\mathcal{T}$, in consequence of which a component may perform one, several or infinitely many *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

[1]The overall delay of message delivery is composed of the sum of several delay terms, such as: send set-up; network access; network propagation; reception. Of course, we might decompose the delivery delay in several timed actions, but this would lead us to a recursive argument, so we just consider a timed action as the smallest unit of execution for the sake of observing timeliness.

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 implementation of 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 such 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 $\mathcal{T}$. 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 otherwise, according to the definition of timing failure.
- We denote $\mathcal{H}(\mathcal{T})$ as a history where $\forall X(p, e, t_e) \in \mathcal{H}(\mathcal{T})$, the value of t_e is related to $\mathcal{T}$. The existence of the relation is important for the results of this paper. The exact relation is only relevant for the implementation.
- We denote $\mathcal{T}_\mathcal{P}$ as a duration bound derived from property $\mathcal{P}$. That is, a new $\mathcal{T}$ is derived whenever a bound is established in the course of implementing a property (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*, $\not\dashrightarrow$, that is, $\mathcal{T} \not\dashrightarrow \mathcal{P}$.
- 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 that is derived from property $\mathcal{P}$. We also represent this fact by $\mathcal{H} \not\dashrightarrow \mathcal{P}$.

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

4 The Timely Computing Base Model

Given the above introductory sections, the fundamental problem is how to do distributed computations in bounded time. Its hardness consists in achieving reliable and timely processing and communication in a distributed system over an infrastructure with uncertain timeliness.

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. These objectives require a form of: computing in bounded delay; measuring durations; detecting failures. The apparent incompatibility of these objectives with the uncertainty of the environment may be solved if processes in the system have access to a timely computing base, that is, a component that performs these functions on their behalf.

In the following sections, we start by defining what is a *Timely Computing Base, TCB*, and introduce a set of services to be provided by the TCB. Next, we define the necessary properties of a generic *timely computing model*. We motivate the need for those properties by referring to the reasons why systems fail in the presence of uncertain timeliness. Then, we prove that the TCB, with the help of the above-mentioned services, secures the desired properties.

There is one local Timely Computing Base at every site. Inspired by the trusted computing base work[2], we define the architectural principles that the TCB must fulfil, 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 in default of the TCB (e.g. by detouring it)

Shielding - the TCB construction is such that it is itself protected from faults affecting timeliness (e.g. delays in the outside system affecting TCB computations)

Validation - the TCB functionality is such that it allows the implementation of verifiable mechanisms w.r.t. timeliness

We now define the fault and synchronism model of the TCB. 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. This comes from the Interposition principle. We proceed by defining informally a few synchronism properties that should be enjoyed by the TCB local to a site.

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 the TCB. 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.

We need to extend the operation of the TCB to distributed systems. In consequence, we define a distributed TCB as the collection of all local TCBs in a system, interconnected by inter-TCB communication channels by which local TCBs exchange messages. The construction of the distributed TCB must obviously satisfy the interposition, shielding and validation principles. Likewise, communication must possess the same synchronism as the rest of the TCB functions. Furthermore, we assume that the inter-TCB channels provide reliable delivery, in the sense that no messages addressed to correct TCB modules are lost. Property **Ps** 3 completes the synchronism properties, referring to the determinism in the time to exchange messages amongTCB modules:

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

The achievement of reliable (unicast or multicast) delivery is not treated in this paper, since it will 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. In consequence, we also assume nothing about how many local TCBs can fail, since under a fail-silent model this is irrelevant to the correctness of operation of the distributed 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.

Due to lack of space, we will not go very far into discussing how to implement and validate a TCB. There is a body of research on real-time operating systems and networks that has contributed to this subject[9, 24, 25, 26]. 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. 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.

These principles also postulate the control of the TCB over a timely inter-TCB communication channel. Note that this channel, that we call *control* channel, may or may not be based on a physically different network from the one supporting the channel used for normal communication, that we call *payload* channel. 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 that is required of the control channel is much smaller than that of the payload channel: 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[32, 8, 7, 34]. 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.

5 Services of the TCB

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. We start with timely execution and duration measurement.

Timely Execution

TCB 1 Eager Execution: *Given any function F with an execution time bounded by a known constant $T_{XEQ_{max}}$, for any eager execution of the function triggered at real time t_{start}, the TCB terminates F within $T_{XEQ_{max}}$ from t_{start}*

TCB 2 Deferred Execution: *Given any function F, and a delay time lower-bounded by a known constant $T_{XEQ_{min}}$, for any deferred execution of the function triggered at real time t_{start}, the TCB does not start the execution of F within $T_{XEQ_{min}}$ from t_{start}*

Eager Execution allows the TCB to execute arbitrary functions deterministically, given a feasible $T_{XEQ_{max}}$. Deferred Execution allows the TCB to execute delayed functions, such as those resulting from timeouts ($T_{XEQ_{min}}$). Informally, the former ensures that something finishes before a deadline, whereas the latter ensures that something does not start before a liveline[37]. Another useful service in some applications, the deterministic (time bounded) execution of a function after a delay, can be achieved by compounding **TCB** 2 and **TCB** 1 in this order.

Duration Measurement

TCB 3 *There exists $T_{DUR_{min}}$ such that given any two events e_s and e_e separated by an interval bounded by a known constant $T_{SEP_{max}}$, and 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 $t_e - t_s - T_{DUR_{min}} \leq T_{se} \leq t_e - t_s + T_{DUR_{min}}$*

The TCB can measure distributed durations, with an error $T_{DUR_{min}}$. Obviously, if the events are separated by less than $T_{DUR_{min}}$, the measurement is not significant. This is treated further in [35]. The error $T_{DUR_{min}}$ depends on $T_{SEP_{max}}$, because the local clocks skew from real-time (Property **Ps** 2).

Another crucial service of the TCB is failure detection. We define a *Perfect Timing Failure Detector (pTFD)*, using an adaptation of the terminology of Chandra[13].

Timing Failure Detection

TCB 4 Timed Strong Completeness: *There exists $T_{TFD_{max}}$ such that given a timing failure at p in any timed action $X(p, e, t_e)$, the TCB detects it within $T_{TFD_{max}}$ from t_e*

TCB 5 Timed Strong Accuracy: *There exists $T_{TFD_{min}}$ such that given any timely timed action $X(p, e, t_e)$ by p that does not occur within $-T_{TFD_{min}}$ from t_e, the TCB considers p timely*

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 $T_{TFD_{max}}$ of its occurrence. In essence, it specifies the detection latency of the pTFD.

Timed Strong Accuracy can also 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 $T_{TFD_{min}}$ before the detection threshold (the specified bound). In essence, it specifies the detection accuracy of the pTFD.

The majority of detectors known are *crash* failure detectors. Crash failures are particular cases of timing failures, where the process responsible for executing a specification produces infinitely many timing failures with infinite lateness degree. In consequence, the TCB is also capable of detecting crash failures. In fact, there is a transformation from the TFD to a Crash Failure Detector that we do not introduce here, due to lack of space, and since it is not fundamental for the results of this paper.

6 Dependable and Timely Computing with a TCB

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* channel, the system admits the construction of say, a *control* part, made of local TCB modules, interconnected by some form of medium, the *control* channel. The medium may be a virtual channel over the available physical network or a network in its own right. Processes p execute on the several sites, making use of the TCB whenever appropriate. The TCB subsystem, dashed in the figure, preserves, by construction, properties **Ps** 1 to **Ps** 3. The nature of the modules and the interconnection medium is outside the scope of this paper, but these are problems with known solutions, and we have given a few hints on how to implement a TCB in Section 4.

How applications interact with the TCB is also another important problem. While this is an issue we will tackle in another paper, it deserves at least a brief discussion, so we leave here the intuition behind our approach. The first remark is that applications 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. The second is that the TCB always detects a late event as a timing failure, within the terms of the definition of timing failure detector. Finally, although the TCB detects timing failures, nothing obliges an application to become aware of such failures. 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 past steps before proceeding to the next step.

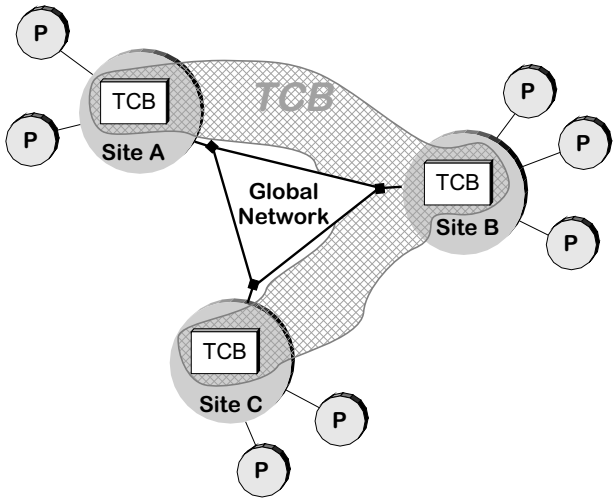

Figure 1: The TCB Architecture

How can the TCB help design dependable and timely applications? 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.

We recall that an application is a computation in general, that any application is 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:

- decreased coverage
- unexpected delay
- contamination

When we make assumptions about the absence of timing failures, we have in mind a certain coverage, which is the 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 hard real-time systems, and may be somewhat relaxed for any other realistic real-time system, like mission-critical or soft real-time. However, in a system with uncertain timeliness, this correspondence varies during system life. If the environment conditions start degrading to states worse than assumed, the coverage incrementally decreases, that is, the probability of timing failure increases. On the other hand, if coverage is better than assumed, we are not taking full advantage from what the environment gives. This is a generic problem for any class of system relying on the assumption / coverage binomial[31]: 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. This handles the first of the problems we enumerated as a consequence of timing failures: decreased coverage.

Nevertheless, even if long term coverage stability is ensured, instantaneously the system can still have timing failures. The immediate effect of timing failures may be twofold: delay and / or incorrectness by contamination. We define unexpected **delay** as the violation of a timeliness property. That can sometimes be accepted, as we discuss in the next section. 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. For example, aggressive timeouts, and failure detection, when embedded in the protocols, create scenarios that bring the FLP problem inexorably back, as reported in [13, 12]. In fact, these problems 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, and are thus prone to timing failures, without taking measures to counter them (because they were not

supposed to exist!); (b) in consequence, by not ensuring error confinement[28] they sometimes let timing failures contaminate (logical) safety properties[2]. The violation of safety properties, such as having processes choose different values in consensus, because of erroneous timeout-based error detection, is well exemplified in [5, 12]. The contamination problem was first described in the context of ordered broadcast protocols in [22]. 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 previous steps were timely. Now observe that "timely" means that the detection latency is bounded. In consequence, if processing steps wait more than the detection latency, absence of failure indication means "no failure", and thus they 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 is an extra delay. 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 $\mathcal{H}(\mathcal{I}_{\mathcal{P}})$ 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[13], 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. 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, 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.

The next step is to prove that the TCB helps applications to secure coverage stability and no-contamination, despite the uncertainty of the environment. In all the proofs that follow, timely execution is secured by services **TCB** 1 and **TCB** 2.

6.1 Enforcing Coverage Stability with the TCB

We start with a definition of coverage stability for an application.

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

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

[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.

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]

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

In practical terms, $\mathcal{T}\epsilon$ applications are those whose bounds can be increased or decreased dynamically, such as QoS-driven applications. We now prove that a $\mathcal{T}\epsilon$ application achieves coverage stability under certain conditions. We start by giving informal proofs of two lemmata about the capability of the TCB to make histograms of the distribution of durations (service **TCB** 3) and thus gathering evidence about coverage.

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

PROOF SKETCH:
From service **TCB** 3, 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 $\mathcal{T}$, there is a minimum n_0 such that a discrete probability distribution function pdf, built with n_0 observations $T(i)$ subjected to the TCB duration measurement error, $T_{DUR_{min}}$, represents $\mathcal{T}$ with an error of p_{dev0}[19]. □

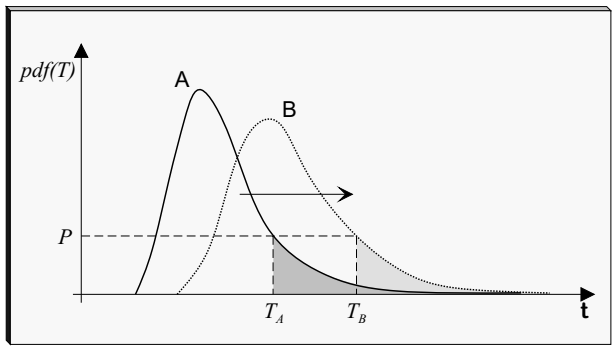

Figure 2: Example variation of distribution $pdf(\mathcal{T})$ with the changing environment

In systems of uncertain timeliness, the pdf of a duration varies with time. Normally, if a sufficiently large interval 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 curves A and B. For the results of this paper, it is enough to know that p_{dev} remains bounded, as we state in Lemma 2 below. However, one wishes to keep the error p_{dev} small in order to predict the probability of any T accurately, even with periods of timeliness instability. As future work, we plan to study efficient functions to keep p_{dev} small, in the presence of large deviations [38].

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

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

PROOF SKETCH:
From Lemma 1, the TCB is capable of recomputing $pdf(\mathcal{T})$ 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[38], such that $P = pdf_i(T)$ recomputed with the $n_{i-1} + n$ observations has a bounded error p_{dev} from p. □

Next we prove that every $\mathcal{H}$ has coverage stability 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 *timeliness-tuning* algorithm, and it is given in Figure 3.

For each function f of an application A having assumed duration $\mathcal{T}_f$, with coverage P_f

```
01  / / pdf(T_f) exists and is kept updated by the TCB
02  / / pdf^-1 is the "inverse" of pdf
03  / / T_f is initialized to some value
04  / / h is the hysteresis of triggering of the algorithm
05  / / TCB_setTo sets T_f to pdf^-1(P_f)
06
07  foreach f do
08        when |pdf^-1(P_f) - T_f| > h do
09              TCB_setTo(T_f, pdf^-1(P_f))
10        od
11  od
```

Figure 3: Timeliness-Tuning Algorithm

Consider any function of an application A depending on a specified duration $\mathcal{T}_f$, assumed to hold with a probability P_f. An execution of a timed action X is triggered whenever the application takes an action conditioned to $\mathcal{T}_f$. Typically, this action can be the sending of a message (which generates a remote event) or the execution of a local function. In either case the TCB follows the execution of this action, and in consequence the timeliness-tuning algorithm assumes that the TCB has created $pdf(\mathcal{T}_f)$ and keeps it updated.

Given pdf and P_f such that $P_f = pdf(T)$, it is easy to extract an approximation of value T. In the algorithm of Figure 3, we denote this operation by pdf^{-1}, $pdf^{-1}(P_f) \simeq T$ being the current observed duration that holds with the desired probability P_f (it may have deviated from the specified duration $\mathcal{T}_f$).

For example, consider that curve A as depicted in Figure 2 represents the steady-state initial operational environment for $\mathcal{T}_f$: if $P = P_f$, then $\mathcal{T}_f \simeq T_A$. Whenever the TCB detects a significant variation in the environment (line 8), that is, when $pdf^{-1}(P_f)$ exceeds an interval h around $\mathcal{T}_f$, $\mathcal{T}_f$ is set to T. We represent this update in the algorithm (line 9) by a function $TCB\_setTo()$. The semantics of $TCB\_setTo()$ may be tuned to specific applications. However, most of the times, it can be as simple as "set $\mathcal{T}_f = pdf^{-1}(P_f)$ in all processes". Please note that this new value corresponds to what the TCB considers the

adequate value to obtain the desired probability P_f of bound $\mathcal{T}_f$ holding. To understand why, let us look again at Figure 2: the environment got slower, and there was a shift of pdf to the right. 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 $\mathcal{T}_f$, the application maintains the degree of coverage. Note that these remarks are also true when the environment gets faster: the bound should get back to its lower value as soon as possible. $\mathcal{T}_f$ is only changed again when the measured bound deviates from $\mathcal{T}_f$ more than h, the hysteresis of triggering of the algorithm.

Lemma 3 *Given property $\mathcal{P}$ with assumed coverage $P_{\mathcal{P}}$, and any history $\mathcal{H}(\mathcal{T}_{\mathcal{P}})$ derived from $\mathcal{P}$, built using a TCB and the timeliness-tuning algorithm, $\mathcal{H}$ has coverage stability*

PROOF:
Immediate from Lemmata 1 and 2, and the discussion of the algorithm □

Now we have the final theorem.

Theorem 1 *An application $A \in \mathcal{T}\epsilon$ using a TCB and the timeliness-tuning 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 $\mathcal{T}$. In consequence, any $\mathcal{T}$ can be modified, namely with the intent of achieving coverage stability. From Lemma 3, we see that any bound $\mathcal{T}$ can be continuously adjusted with the timeliness-tuning algorithm, so that the history $\mathcal{H}(\mathcal{T})$ has coverage stability. By induction on every $\mathcal{P} \in \mathcal{P}_A$, and by Definition 1, application A has coverage stability. □

6.2 Avoiding Contamination with the TCB

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

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

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

Firstly, we prove some lemmata about the capability of failure detection of the TCB (services **TCB** 4 and **TCB** 5).

Lemma 4 *Given any $\mathcal{H}(\mathcal{T}')$, and any $X \in \mathcal{H}$, if the execution of X fails the TCB detects it by $\mathcal{T} = \mathcal{T}' + T_{TFD_{max}}$*

PROOF
This comes directly from the TFD, from service **TCB** 4. □

Lemma 5 *Given any $\mathcal{H}(\mathcal{T}')$, and any $X \in \mathcal{H}$, the TCB never detects the execution of X as failed if it takes place by $\mathcal{T}'' = \mathcal{T}' - T_{TFD_{min}}$*

PROOF
This comes directly from the TFD, from service **TCB** 5. □

These two lemmata reveal a very interesting implementation result.

Rule 1 *Given $\mathcal{P}$ and $\mathcal{T} \not\xrightarrow{d} \mathcal{P}$, implemented such that:*

- *$\mathcal{T}$ is the bound as seen by the application;*
- *$\mathcal{T}' = \mathcal{T} - T_{TFD_{max}}$ is the bound as seen by the TCB;*
- *$\mathcal{T}'' = \mathcal{T} - T_{TFD_{max}} - T_{TFD_{min}}$ is the bound as achieved by the environment (the "real" bound);*

then:

- *any timing failure (delay beyond $\mathcal{T}'$) is detected by the TCB by $\mathcal{T}$*
- *any timely execution (terminating by $\mathcal{T}''$) is never detected as failed*

That is, when implementing property $\mathcal{P}$ in classical systems, one would base specifications on bound $\mathcal{T}$. $\mathcal{T}$ would be based on the matching between what was desired for the application and what the support environment (e.g. a network) would yield. Timing failure detection would be based on guaranteeing that $\mathcal{T}$ were met. In fact, **Rule** 1 proposes to use different values for each of the three above-mentioned aspects of the implementation of $\mathcal{P}$ (specification, implementation, error detection): given $\mathcal{T}''$, the possible bound yielded by the support environment, we use $\mathcal{T}' = \mathcal{T}'' + T_{TFD_{min}}$ as the failure detection threshold, and $\mathcal{T} = \mathcal{T}' + T_{TFD_{max}}$ as the bound visible to the application (which the failure detector will secure).

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 in absence of failures the system's safety properties remain correct. We now prove the first theorem about the reliability of applications based on a TCB: that an $\mathcal{F}\sigma$ application achieves no-contamination under certain conditions.

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 4 and 5, we see that for any bound $\mathcal{T}$ established as per **Rule** 1, 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 at the latest to t_{FS}, 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 correct after halting, by Definition 4. Then, by Definition 3, application A 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 $\mathcal{P}_A$*

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

In practical terms, $\mathcal{T}\sigma$ applications are those: in whose properties timeliness is neatly separated from logical safety; and where the implementation of safety properties does not depend on time (e.g. timeouts). We now prove that a $\mathcal{T}\sigma$ application achieves no-contamination under certain conditions, in a second theorem.

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. □

7 Example Applications of the TCB Model

Timing failures, in a fault-tolerance sense, can be handled in one of the following ways: 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 are mostly ensured by the TCB and its services. The last two are secured indirectly, by the Coverage Stability and No-Contamination properties. Furthermore, our model is able to distinguish between several mechanisms of timing failure:

- decreased coverage;
- unexpected delay;
- contamination.

In consequence, one should be able to program real-time applications which have several degrees of dependability, despite the occurrence of timing failures: by detecting

timing failures and still being able to recover by reconfiguration (e.g. by postponing a decision, by increasing the deadline, etc.); by detecting irrecoverable timing failures and doing a fail-safe shutdown; by masking independent timing failures 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 apppropriate, we refer the reader to some works under several models, which separately approximate some of the solutions we now propose in a unified way under the TCB model.

7.1 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:

- define all important timing parameters so that timing failures can be detected by the TCB;
- **Rule** 1 (Section 6.2) should be followed, so that failure detection can be done before any harm happens, and if nothing else can be done, at least the application fails-safe before getting incorrect;
- moreover, since all processes can be informed by the TCB of failure occurrences, switching to a fail-safe state 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[29, 20, 21]. In the examples described in [20] and [21], 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. Timely switching to the fail-safe state can be ensured by the TCB, no matter how asynchronous the payload system.

7.2 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.

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[1, 3, 14, 6, 16, 23]. 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 was laid down in Section 6.1:

- defining all important timing variables;
- building a pdf for each of those variables with the support of the TCB;
- applying the timeliness-tuning algorithm (Fig. 3) to relax or tighten the bounds dynamically.

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.

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 6.2, and is simply based on:

- specifying applications such that safety and timeliness specifications are neatly 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 (eager and deferred execution, duration measurement and obviously, timing failure detection) when necessary;

7.3 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[37, 16, 26, 39]. However, to our knowledge, the work presented in [4] was the first to address the issue of *timing* fault tolerance by 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 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 the faults and to guarantee that the service will be timely executed. The simplest would be to use the first timely replica. However, timing fault detection itself should be left to the TCB. Accurate **timing fault detection** and **time-safety** are important to ensure: selection of timely service by the timely replicas; no-contamination of 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. 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[37]. However, the coverage guarantees for replica set have been recovered.

8 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[11].

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.

References

[1] Tarek F. Abdelzaher and Kang G. 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. IEEE Computer Society Press.

[2] M. Abrams, S. Jajodia, and H. Podell, editors. *Information Security*. IEEE CS Press, 1995.

[3] Carlos Almeida and Paulo 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.

[4] Carlos Almeida and Paulo 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, December 1998.

[5] E. Anceaume, B. Charron-Bost, P. Minet, and S. Toueg. On the formal specification of group membership services. Technical Report RR-2695, INRIA, Domaine de Voluceau, Rocquencourt, B.P. 105, 78153 LE CHESNAY Cedex, France, November 1995.

[6] S. Bagchi, K. Whisnant, Z. Kalbarczyk, and R. K. Iyer. Chameleon: A software infrastructure for adaptive fault tolerance. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, Pardue University, West Lafayette, IN, USA, October 1998.

[7] R. Braden, Ed., L. Zhang, S. Berson, S. Herzog, and S. Jamin. RFC 2205: Resource ReSerVation Protocol (RSVP) — version 1 functional specification, September 1997. Status: PROPOSED STANDARD.

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

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

[10] A. Burns and A. Wellings. Real-time distributed computing. In *Proceedings of the 5th Workshop on Future Trends of Distributed Computing Systems*, pages 34–40, Cheju Island, Korea, August 1995.

[11] A. Casimiro, F. Cristian, C. Fetzer, and P. Veríssimo. Private communications, September 1998.

[12] Tushar Chandra, Vassos Hadzilacos, Sam Toueg, and Bernadette Charron-Bost. On the impossibility of group membership. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing*, pages 322–330, Philadelphia, USA, May 1996. ACM.

[13] Tushar Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[14] François J. N. Cosquer, Luís Rodrigues, and Paulo Veríssimo. Using Tailored Failure Suspectors to Support Distributed Cooperative Applications. In *Proceedings of the 7th International Conference on Parallel and Distributed Computing and Systems*, pages 352–356. IASTED, October 1995.

[15] Flaviu Cristian and Christof Fetzer. The timed asynchronous system model. In *Proceedings of the 28th Annual International Symposium on Fault-Tolerant Computing*, pages 140–149, Munich, Germany, June 1998. IEEE Computer Society Press.

[16] 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, October 1998. IEEE Computer Society Press.

[17] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *24th Annual Symposium on Foundations of Computer Science*, November 1983.

[18] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

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

[20] Christof Fetzer and Flaviu Cristian. Fail-awareness in timed asynchronous systems. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing*, pages 314–321a, Philadefphia, USA, May 1996. ACM.

[21] Christof Fetzer and Flaviu Cristian. Fail-awareness: An approach to construct fail-safe applications. In *Proceedings of the 27th Annual International Fault-Tolerant Computing Symposium*, pages 282–291, Seattle, Washington, USA, June 1997. IEEE Computer Society Press.

[22] A. Gopal and S. Toueg. Inconsistency and contamination (preliminary version). In *Proceedings of the 10th Annual ACM Symposium on Principles of Distributed Computing*, pages 257–272, Montreal, Québec, Canada, August 1991. ACM Press.

[23] M. Hiltunen, R. Schlichting, X. Han, M. Cardozo, and R. Das. Real-time dependable channels: Customizing qos attributes for distributed systems. Technical Report TR 99-xx, University of Arizona, Department of Computer Science, Tucson, AZ, February 1999.

[24] Farnam Jahanian. Fault tolerance in embedded real-time systems. *Lecture Notes in Computer Science*, 774:237–249, 1994.

[25] 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*, pages 35–41, Huntsville, Alabama, USA, October 1990. IEEE Computer Society Press.

[26] 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*, 550:166–188, 1991.

[27] R. Koymans. Specifying real-time properties with metric temporal logic. *Journal of Real-Time-Systems*, 2(4):255–299, November 1990.

[28] J. C. Laprie. Dependability: A Unifying Concept for Reliable Computing and Fault-Tolerance. In *Resilient Computing Systems*, volume 2. Collins and Wiley, 1987.

[29] Lubaszewski and Courtois. A reliable fail-safe system. *IEEETC: IEEE Transactions on Computers*, 47, 1998.

[30] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer, New York, 1992.

[31] David Powell. Failure mode assumptions and assumption coverage. In *Digest of Papers, The 22nd International Symposium on Fault-Tolerant Computing Systems*, pages 386–395, Boston, USA, July 1992. IEEE Computer Society Press.

[32] Martin de Prycker. *Asynchronous Transfer Mode: Solution For Broadband ISDN*. Prentice-Hall, third edition edition, 1995.

[33] Krithi Ramamritham, Stankovic, and Wei Zhao. Distributed sheduling of tasks with deadlines and resource requirements. *IEEE Transactions Computers*, 38(8):1110–1123, August 1989.

[34] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A transport protocol for real-time applications. Status: PROPOSED STANDARD RFC 1889, Audio-Video Transport Working Group, January 1996.

[35] Paulo Veríssimo. Ordering and timeliness requirements of dependable real-time programs. *Journal of Real-Time Systems*, 7(2):105–128, September 1994.

[36] Paulo Veríssimo and Carlos 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)*, 7(4):35–39, Winter 1995.

[37] Paulo Veríssimo, P. Barrett, P. Bond, A. Hilborne, L. Rodrigues, and D. Seaton. The Extra Performance Architecture (XPA). In D. Powell, editor, *Delta-4 - A Generic Architecture for Dependable Distributed Computing*, ESPRIT Research Reports, pages 211–266. Springer Verlag, November 1991.

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

[39] Hengming Zou and Farnam 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, February 13, 1998.