The Timely Computing Base: Timely Actions in the Presence of Uncertain Timeliness

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

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

Christof Fetzer
christof@research.att.com
AT&T[†]

Abstract

Real-time behavior is specified in compliance with timeliness *requirements, which in essence calls for synchronous system models. However, systems often rely on unpredictable and unreliable infrastructures, that suggest the use of asynchronous models. Several models have been proposed to address this issue. We propose an architectural construct that takes a generic approach to the problem of programming in the presence of uncertain timeliness. We assume the existence of a component, capable of executing timely functions, which helps applications with varying degrees of synchrony to behave reliably despite the occurrence of timing failures. We call this component the Timely Computing Base, TCB. This paper describes the TCB architecture and model, and discusses the application programming interface for accessing the TCB services. The implementation of the TCB services uses fail-awareness techniques to increase the coverage of TCB properties.*

1. Introduction and Motivation

A large number of the emerging services have response 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).

Real-time behavior is specified in compliance with *timeliness* requirements, which in essence calls for synchronous system models. Under this model there are known bounds for all relevant timing variables.

However, unpredictable and unreliable infrastructures are not adequate environments for synchronous models, since it is difficult to enforce timeliness assumptions. Violation of assumptions might cause incorrect system behavior. In contrast, an asynchronous model is a well-studied framework, appropriate for these environments. Informally, 'asynchronous' means that there are no bounds on timing variables, such as processing speed or communication delay. In summary, fully asynchronous models do not satisfy our needs, because they do not allow the specification nor the enforcement of timeliness specifications. On the other hand, enforcing the properties of fully synchronous models is very difficult to achieve in infrastructures with poor baseline timeliness properties. The issue that has to be addressed is: *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 can rely on services provided by a synchronous module, the TCB.

The proposed TCB is just a small part of the system and thus its properties can be implemented with high coverage. We describe an architecture that takes reliability concerns into account, enforcing the coverage of TCB synchronism properties. We show how the TCB services can be implemented and how they can be used by applications.

The paper is organized as follows. In the next section we present a brief survey of related work. Section 3 provides the definition of timing failure. The Timely Computing Base Model is introduced in Section 4, after which we present the services and application programming interface of the TCB, in Section 5. Sections 6 and 7 discuss how to implement the TCB services and improve their coverage through self-checking mechanisms. The paper concludes with some considerations about future work.

2. Related Work

Several previous papers have proposed models for systems with uncertain temporal behavior. Chandra & Toueg have studied the minimal strengthening of the time- free model [14] such that the consensus and atomic broadcast

[*] Faculdade de Ciências da Universidade de Lisboa. Bloco C5, Campo Grande, 1700 Lisboa - Portugal. Tel. +(351) 21 750 0087 (secretariat); +(351) 21 750 0103 (direct) (office). Fax +(351) 21 750 0084. Navigators Home Page: http://www.navigators.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 project IST-1999-11583 (MAFTIA).

[†] AT&T Labs. 180 Park Ave, Florham-Park, NJ07932, USA.

problems become solvable in the presence of crash failures: they give a failure detector which will eventually stop suspecting at least one correct process and that will eventually suspect all crashed processes[6]. The present authors, in separate teams, have developed models of partial synchrony that can be seen as precursors of the present work: the timed-asynchronous model, where the system alternates between synchronous and asynchronous behavior, and where hardware clocks provide sufficient synchronism to make decisions such as ’detection of timing failures’ or ’fail-safe shutdown’[8]; the quasi-synchronous model, where parts of the system have enough synchronism to perform ’real-time actions’ with a certain probability[20]. All 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 works have studied the minimum guarantees for securing the safety properties of the system, assuming a time-free liveness perspective [9, 10].

3. Timing Failures

In our opinion, a general model can be devised that encompasses the entire spectrum of what is sometimes called *partial synchrony*. The common denominator of systems belonging to that realm is that *they can exhibit timing failures*, denoted as the violation of timeliness properties. Informally, timeliness properties concern the specification of timed actions, such as: P **within** T **from** t_0 (T- duration; t_0- instant of reference). Examples of timed actions are the release of tasks with deadlines, the sending of messages with delivery delay bounds, and so forth. A full discussion can be found in [22].

The bounds specified for a timed action may be violated, in which case a timing failure occurs. We base our approach on the observability of the termination event of a timed action, regardless of where it originated. If a timed action does not incur in a timing failure, the action is *timely*.

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, $Ld = t_e^{'} - t_e$, is the lateness degree*

This brings us to the central issue of this paper: the Timely Computing Base (TCB) model and its implementation and programming interface, as a paradigm for achieving timely actions in the presence of uncertain timeliness.

4. The Timely Computing Base Model

The architecture of a system with a Timely Computing Base (TCB) is suggested by Figure 1. The first relevant aspect is that the heterogeneity in system synchronism was cast into the system architecture. There is a generic or *payload* system, over a global network or *payload* channel. This prefigures what is normally ’the system’ in homogeneous architectures, that is, where applications run. Additionally, there is 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 an alternative network in its own right. The second relevant aspect is that the TCB has well-defined synchronism properties. The TCB provides simple support services, such as the ability to detect failures, to measure durations, and to execute *timely* timed actions. For certain types of less critical applications, it is not necessary that all sites have TCBs. However, note that this decreases the capability of these sites to exhibit guaranteed synchronous behavior, which may be a nuisance in, e.g., fail-safe or real-time systems. So, for simplicity, and to show the virtues of the model, in this paper we assume that every site has a TCB.

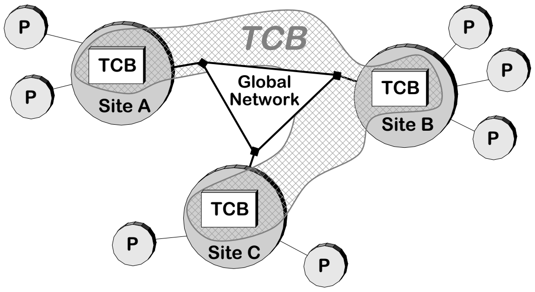


Figure 1. The TCB Architecture

4.1. Payload System Properties

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 (we use both designations interchangeably). 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 drift rate. We assume the system to follow an omissive failure model, that is, components *only have late timing failures*— and omission and crash, since they are subsets of timing failures— no value failures occur. The system model has 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.

4.2. Timely Computing Base Properties

We now define the fault and synchronism model of the TCB. There is one local Timely Computing Base at every site. 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. The TCB subsystem, shown dashed in the figure, preserves, by construction, the properties of a synchronous system:

Ps 1 *There exists a known upper bound $T_{D^1_{max}}$ on TCB processing delays*

Ps 2 *There exists a known upper bound $T_{D^2_{max}}$ on the drift rate of local TCB clocks*

Ps 3 *There exists a known upper bound $T_{D^3_{max}}$, on the delivery delay of messages exchanged between TCB moduleFs*

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. Property **Ps** 3 completes the synchronism properties, referring to the upper bound on the time to exchange messages among TCB modules. We assume that the inter-TCB channels provide reliable delivery, in the sense that no messages addressed to correct TCB modules are lost. The distributed TCB is the collection of all local TCBs in a system, together with the inter-TCB communication channels (see Figure 1). 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. We 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.

Note that having a computing base with these properties opens very interesting perspectives, in terms of turning the TCB into an oracle for applications (even asynchronous) to solve their time-related problems. Accomplishing this raises three orders of problems:

- defining the minimal services— the TCB must be kept simple;
- defining the payload-to-TCB interface— to allow potentially asynchronous applications to dialogue with a synchronous component may prove delicate;
- implementing the TCB services with the necessary coverage.

The minimal services required to satisfy a wide range of applications with timeliness requirements have essentially to do with: ability to measure distributed durations with bounded accuracy; complete and accurate detection of timing failures; ability to execute well-defined functions in bounded time.

The way applications interact with the TCB is also another important problem. 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. However, when timing failures occur, the TCB can help implement contingency plans, such as timely fail-safe shutdown. Finally, although the TCB detects timing failures, nothing obliges an application to become aware of such failures. In consequence, applications take advantage of the TCB by construction, typically using it as a pacemaker, letting it assess (explicitly or implicitly) the correctness of past steps before proceeding to the next step. The crux of this style of operation is the application programming interface, which we discuss in Section 5, together with the TCB services.

Finally, the TCB implementation, which we discuss in Section 6. One may suggest that we are only hiding in the TCB the main problem that other systems have: achieving coverage of synchrony assumptions. We stress the fundamental architectural principle that makes the approach different from previous models— by design, there are: a larger-scale, more complex, asynchronous at the limit, payload part, where we can run the logic part of our algorithms, which can even be time-free; and a small-scale, simple, synchronous, control part, where we can run the time-related part of our algorithms, with the support of the TCB services. In consequence, the coverage of the implementation of synchronous services on the TCB can be made comparably much higher than one would achieve for the same services implemented on the payload system. In generic terms, the TCB can be seen as a *coverage amplifier* for the execution of the time-critical functions of any system (the TCB services). Still, the TCB can still fail with nasty consequences for the safety of applications no matter how small and simple it may be. In Section 7, we show how to improve the coverage of the TCB through very simple self-checking mechanisms.

5. Dependable Programming on the TCB

How can the TCB help design dependable and timely applications? We begin this section with the presentation of a simple set of services to be provided by the TCB. Then, we introduce and explain the application programming interface between payload applications and the TCB. Note that the interface must ensure correct interaction between the essentially asynchronous world of the payload applications, and the synchronous world of the TCB.

5.1. 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. The properties are defined as seen at the TCB interface. 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_{Xmax}, for any eager execution of the function triggered at real time t_{start}, the TCB terminates F within T_{Xmax} from t_{start}*

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

Eager Execution allows the TCB to execute arbitrary functions deterministically, given a feasible T_{Xmax}. Deferred Execution allows the TCB to execute delayed functions, such as those resulting from timeouts (T_{Xmin}).

Duration Measurement

TCB 3 *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: $T_{INT}T_{D^2_{max}} + T_{Dmin}$. 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 ϵ apart from real-time. In systems with external clock synchronization, the measurement error is bounded by 2ϵ. Note that internal clock synchronization for the matter would not help here. Although given properties **Ps** 1–**Ps** 3 one could implement global time, explicitly synchronized clocks would just improve some variables quantitatively, but they would not increase the power of the model. To minimize the assumptions of the model, we refrain from requiring synchronized clocks.

Timing Failure Detection

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[6].

TCB 4 Timed Strong Completeness: *There exists T_{TFDmax} 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_{TFDmax} from t_e*

TCB 5 Timed Strong Accuracy: *There exists T_{TFDmin} such that any timely timed action $X(p, e, T_A, t_A)$ that does not terminate within $-T_{TFDmin}$ from t_e is considered timely by the TCB if the local TCB does not crash until $t_e + T_{TFDmax}$*

The majority of detectors known are *crash* failure detectors. We introduce timing failure detectors. 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_{TFDmax} 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_{TFDmin} before the detection threshold t_e. In essence, it specifies the detection accuracy of the pTFD.

5.2. Application Programming Interface

Given the baseline asynchronism of a payload application, there are no guarantees about the actual time of invocation of a TCB service by the former. In fact, the latency of service invocation may not be bounded. The same can be said of the actual time that responses or notifications from the TCB arrive at the application buffer. When considering the interface definition this is perhaps the most important problem. The interface presented in this section makes the bridge between a synchronous environment and a potentially asynchronous one.

Duration Measurement

The most basic function we have to provide is obviously one that allows applications to read a clock:

```
timestamp ← getTimestamp()
```

The function returns a timestamp generated inside the TCB. Since the application runs in the payload part of the system, when it uses a single timestamp there are no guarantees about how accurately this timestamp reflects the current time. However, a difference between two timestamps represents an upper bound for a time interval, if the interval took place between the two timestamps. For instance, just by using this function an application is able to obtain an upper bound on the time it has needed to execute a computation step: it would suffice to request two timestamps, one before the execution and another after it. If this execution is a timed action, then the knowledge of this upper bound is also sufficient to detect a timing failure, should it occur. The TCB recognizes the importance of measuring local durations and explicitly provides interface functions to do this:

```
tag ← startMeasurement(start_ev)
end_ev,procdelay ← stopMeasurement(tag)
```

When the `startMeasurement` function is called, the application has to provide a timestamp to mark the start event. It gets a request `tag`. When it wants to mark the end event, and obtain the measured duration, it calls `stopMeasurement` for `tag`. The service gets a timestamp for the end event, and the difference between the two timestamps yields the duration. A very simple example of the usage of these functions is depicted in Figure 2a. Here, an application has to execute some computation in a bounded time interval (T_{spec}), on a best-effort basis. If this is achieved, the computation results can be accepted. Otherwise they are rejected. Possibly there will be subsequent (adjacent) computations also with timeliness requirements. In that case, the end event of a computation is used as the start event of the next one in order to cancel the time spent to verify the execution timeliness (shown in the far right of the figure, with `startMeasurement(B)`).

Timely Execution

The timely execution service allows the construction of applications where strict timeliness guarantees are sporadically required. In essence, timely execution means the possibility of guaranteeing that something will be executed before a deadline (eager), or that something will not be executed before a liveline[21] (deferred). This maps onto the following interface function:

```
end_ev ←
 startExec(start_ev,delay,t_exec_max,func)
```

When this is called, `func` will be executed by the TCB. The specification of an execution deadline is done through the `start_ev` and `t_exec_max` parameters. The former is a timestamp that marks a reference point from where the execution time should be measured. The latter, a duration, indicates the desired maximum termination time counted from `start_ev`. On return, the `end_ev` parameter contains a timestamp that marks the termination instant. The `delay` parameter is the deferral term, counted from `start_ev`. If it is zero, it is a pure eager execution function. The feasibility of timely execution of each function must be analyzed, for instance, through the calculation of the worst-case execution time (WCET) and schedulability analysis (*see* Section 6).

Not all eager execution requests are feasible. Depending on the specified parameters and on the instant the request is processed, the TCB may not be able to execute it and, in that case, an error status reporting this fact will be returned and made available in the interface.

The example of figure 2b illustrates the utility of the eager execution service. Suppose the application had to execute the computation of the Figure 2a with such strict timeliness requirements (instead of on a best-effort basis) that it would delegate it to the timely execution service of the TCB. The computation had a WCET and was short enough to be schedulable by the TCB. Then, instead of issuing a `startMeasurement` request, the application would call `startExec` with the appropriate `func`. `t_exec_max` would be T_{spec}. The request would always succeed, unless the delay between the `getTimestamp` call and the execution start was so large that the execution was no longer schedulable. The application would just need to check the `startExec` return status.

Timing Failure Detection

We now present the API of the timing failure detection (TFD) service and give two short examples that illustrate how it should be used to solve concrete problems.

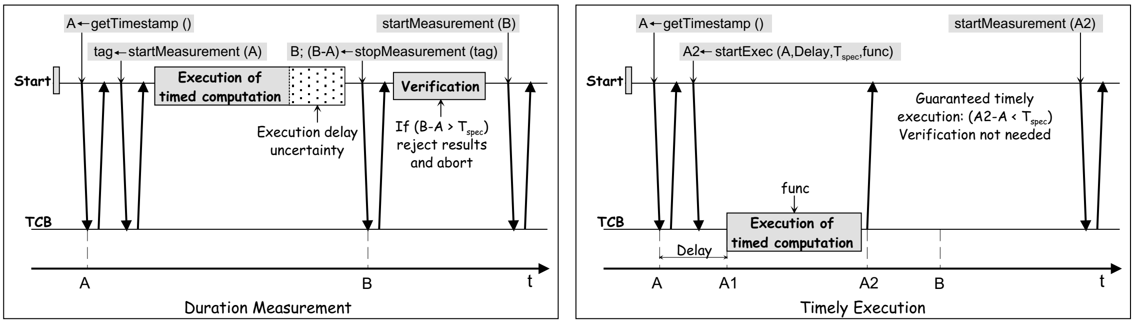
As introduced in section 5.1, there is logically only one TFD service, with the properties **TCB**4 and **TCB**5. However, in practice it is wise to make a distinction between the detection of timing failures in local timed actions and in distributed timed actions. This distinction is important in terms of interface, because in one case the failures are only important to one process (the one performing the action) while in the other they are important to many (all those affected by the distributed action). Therefore, the API described here has two sets of functions: for local and for distributed timing failure detection. The following two functions provide for all that is necessary concerning local timing failure detection:

```
tag ← startLocal(start_ev,spec,handler)
end_ev,procdelay,faulty ← endLocal(tag)
```

With `startLocal` an application requests the service to observe the timeliness of some execution. The TFD service takes `start_ev` as the start instant of the observed execution, and `spec` as the specified execution duration. Each request receives a unique `tag` so that it is possible to handle several concurrent requests. Since the service has to timely detect timing failures, it does not accept requests to observe executions that have already failed. The timely reaction to a timing failure can be delegated on the TCB, using the `handler` parameter. This parameter identifies one of the built-in functions of the TCB, executed as soon as the failure is detected (e.g., an orderly fail-safe shutdown procedure). Note that there would be no guarantees about the timeliness of the reaction if it were done in the payload part of the system.

When the execution finishes the application has to call the `end_local` function, with the identifier `tag`, in order to disable the detection for this action and receive information about the execution: when it finished, its duration and whether it was timely.

The relevance of this service and the importance of timely reaction to failures can be better explained with the following example. Consider a distributed system composed of a controller, a sensor and an actuator processes (Figure 3a). The system has a TCB and the payload part is asynchronous. Since processes are in different nodes, they can only communicate by message passing. The sensor process periodically reads a temperature sensor and sends the value to the controller process. When the controller receives a new reading, it compares it with the set point, and computes the new value to send to the burner, in order to keep the temperature within the allowed error interval. It then sends a command to the actuator process. The system has three classes of critical requirements: (a) the temperature



A←getTimestamp ()
tag←startMeasurement (A)
B; (B-A)←stopMeasurement (tag)
startMeasurement (B)
Start
Execution of timed computation
Verification
Execution delay uncertainty
If (B-A > T_spec) reject results and abort
TCB
A
B
t
Duration Measurement
A←getTimestamp ()
A2← startExec (A,Delay,T_spec,func)
startMeasurement (A2)
Start
Guaranteed timely execution: (A2-A < T_spec) Verification not needed
func
TCB
Execution of timed computation
Delay
A
A1
A2
B
t
Timely Execution

Figure 2. Using TCB services: (a) Duration Measurement; (b) Timely Execution

must remain within $\pm\varepsilon$ of a set point; (b) the control loop must be executed frequently enough (the controller must receive a valid temperature reading every $D2$ time units); (c) once the sensor value read, the actuation value must be computed and sent fast enough to the actuator to achieve accurate control. Let us neglect, for simplicity, the delay in sending the temperature reading from sensor to controller, and consider that the actuation must be acknowledged in $D1$ time units after the reading has been received. A solution for detecting delayed temperature readings is presented in [13].

Requirement (a) can be ensured by the application logic residing in the payload part. Requirements (b) and (c) can be controlled by the TCB on behalf of the application.

System safety is compromised if $D1$ or $D2$ are violated. In this case the system has to switch to a safe state. We assume the controller node to have full control of the heating device power switch and thus able to turn it off, putting the system in a safe state.

Since the system is asynchronous but has timeliness requirements, it has to rely on the TCB. In figure 3b it is possible to observe how the TFD service is used to detect local timing failures. The controller receives a new temperature reading at instant $t1$ (measured by the TCB). From this moment on the bounds $D1$ and $D2$ must be checked, and so it is necessary to invoke the `startLocal` function twice. Note the call `endLocal`(id_0) **after** the other two: the call disables detection for the previous period ($D2$ specification, not shown), since a new message arrived. The controller then sends a command to the actuator and, when it receives the acknowledgment, `endLocal` is called again, this time to terminate the execution of $id1$. Normally, neither $D1$ nor $D2$ expire. If the computation takes so long that $D1$ expires, or if a message is not received from the sensor before $D2$ expires (as depicted), the handler is immediately executed by the TCB. The handler function passed as argument can be a very simple function that is executed by the TCB, issuing a command to the actuator that turns off the heating device.

A distributed execution requires at least one message to be sent between two processes. Thus, the action to be observed for timing failure detection, in addition to local actions, is message delivery. Since a delivery delay is bounded by a send and a receive event, the TFD service just has to intercept message transmissions. The described interface provides not only the required functionality but also allows message interceptions to be done in a very simple and intuitive manner. In the following functions we only present the TFD service-specific parameters (we omit normal parameters such as addresses, etc.).

```
tag ← send(send_ev,spec,handler)
tag,deliv_ev ← receive()
```

The meaning of the `send` function parameters is similar to the ones of the `startLocal` function. We assume it is possible to multicast a message to a set of destination processes using this `send` function. The `receive` function blocks the application until a message is received. On return, the function provides a message `tag` and a timestamp for the delivery event. The information relative to timing failures is queried by means of another function:

```
info₁ ··· infoₙ ← waitInfo(tag)
info=(delivdelay,faulty)
```

When `waitInfo` is called, the application will block until all information is available, but never more than the maximum timing failure detection latency (T_{TFDmax}). `waitInfo` returns the delivery delay and the failure result for each receiver process of message `tag`.

Using this service in the example of figure 3, we may now extend the response time control ($D1$) back to the sensor reading moment, to enforce the freshness of sensor readings. In order to achieve that, the sensor message would be sent using this interface, by specifying some maximum delivery delay. Upon message reception the `waitInfo` function could be used to detect a timing failure. There are other examples of the utility of timing failure detection in application construction [1, 12, 11].

6. Implementing TCB Services

This section provides the basic principles and guidelines to implement the TCB services. We show that despite the importance of the services, their construction can be quite simple. Likewise, their overhead on system execution is kept at a low level.

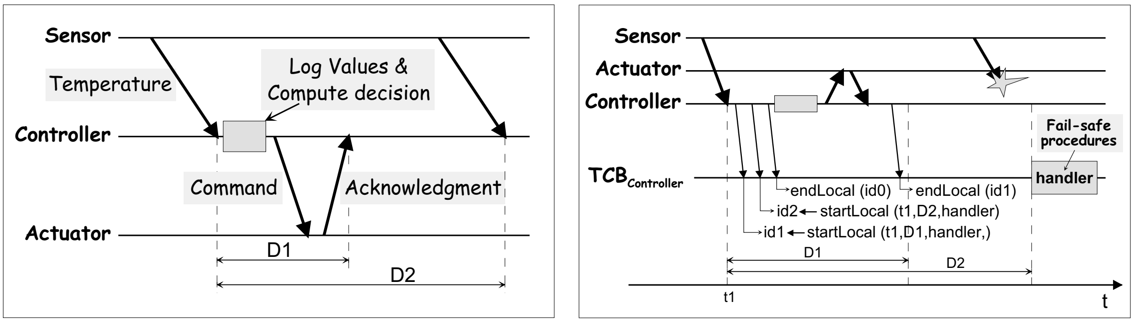


Figure 3. Using Local Timing Failure Detection: (a) Example Scenario and Timing Specifications; (b) Detecting and Handling Timing Failures with the TCB

6.1. Timely Execution Service

There are two essential functions that must be provided by the timely execution service, in order to enforce properties **TCB**1 and **TCB**2 introduced in Section 5.1: eager execution and deferred execution. They can be combined, as found in some real-time schedulers[4].

By definition, the timely execution service is designed for the execution of short-lasting time-critical application functions. These functions should not reside in the application address space. Functions have guaranteed behavior when directly called by the TCB (for instance, they can not be swapped out, otherwise it would be impossible to compute execution times). Inside the TCB, we have to assume that a number of measures have been taken a priori, such as the calculation of the worst-case execution time and the schedulability analysis of the critical albeit simple set of functions submitted to the TCB by each application. There is a body of research on the schedulability of real-time operating systems and networks that contributes to the subject of building the system support of a TCB[4, 15, 16, 17]. A more detailed discussion about the engineering principles behind the construction of a TCB can be found in [22].

Timely function execution can be explicitly triggered by an application, but can also be *implicitly triggered* as a response to some failure detected by the timing failure detection service. The former method is intended for applications where specific actions have to be executed in a bounded amount of time. The latter is mostly useful for the implementation of fail-safe orderly shutdown procedures that must be timely executed upon the occurrence of irrecoverable failures.

6.2. Duration Measurement Service

The local TCB module is trusted to provide accurate readings of time intervals, enforcing property **TCB**3 (see Section 5.1). Thus, when applications in the payload part have to determine durations of executions or transmissions, they delegate this task on the TCB, with the interface discussed in Section 5.

The availability of a local clock with bounded drift rate is sufficient for the measurement of *local durations*. The service has to know which are the two events that bound the time interval, so that it can determine their timestamps. The difference between the two timestamps yields the measured duration. Another way for applications to keep track of their timeliness is by using the timing failure detection service interface. This has the advantage of detecting whether a duration threshold is exceeded, besides measuring the duration.

Measurement of *distributed durations* is done differently. Given that we do not assume the existence of synchronized clocks, the methodology for measuring a distributed duration, that is, for relating timestamps of events in different sites, relies on the well-known round-trip duration measurement technique[7], which is for the matter an implicit clock synchronization action. Distributed durations are often associated to the measurement of message delivery delays. A send and a deliver event bound the measured duration. This technique produces an error associated to each measurement whose value is dictated by several factors, including the separation between the two events[13], as discussed in Section 5.1.

6.3. Timing Failure Detection Service

To construct the timing failure detection (TFD) service it is necessary to employ more elaborate algorithms than for the previously described services. The problem we have to solve is how to build a timing failure detector which satisfies properties **TCB**4 and **TCB**5 (see section 5.1). Timing failure detection of local actions is dealt with at the end of this section. Timing failure detection of distributed actions requires a protocol to be executed by all TFD modules on top of the control channel. Recall that we are talking about actions *in the payload system*.

For lack of space, in this paper we only provide the intuition behind this protocol. However, the protocol is available in a technical report[5]. The protocol executes in rounds, during which each TFD instance broadcasts all relevant information for the detection of timing failures. Distributed common knowledge of failure occurrences is achieved by disseminating information about locally detected failures. When an application sends a message, the TFD service uses the distributed duration measurement service to measure its delivery delay. Therefore, TFD instances in all destination sites will collect the measured value. The error associated to this measurement yields the value of T_{TFDmin} specified in property **TCB**5 (see [5] for a proof). The TCB of the sender site logs the timed action specification of the sent message for later use. Each send request is tagged with a unique identifier, known to the TFD service and to the application. The information concerning each sent message (identifier and bound specification) is disseminated to the relevant sites during the next round.

On each receiver site, the TFD service will eventually learn the delivery delay and the specified bound of each received payload message. The bound on the timing failure detection latency, expressed by T_{TFDmax} in property **TCB**4, is enforced by timely executing a decision function after a given amount of time.

Applications that want to perform a timed action provide a timestamp for the start event (the measurement may start before the action actually starts, but that only depends on the application's own timeliness). The TFD service performs an admission test before accepting to control the action. This test is required to guarantee that timely detection of failure is achievable. In certain conditions (in the limit, if the start timestamp is already older than $t_{now} - T_{duration}$) timing failure detection is infeasible and so the request fails. For the application, this equals a timing failure.

Local actions are a subset of distributed actions, and they can be checked locally. To detect timing failures of local actions it is necessary to pre-register those actions, specifying the duration and getting a timestamp for the start instant. Again, there is an admission test and the request may be denied. The TFD service uses the timestamp, the current time and the specified duration to set a timer that counts the remaining time interval. Thereafter, either the TFD service receives an indication that the action has ended or the timer expires. The TFD service delivers the result, success in the former case, failure in the latter.

As we mentioned above, it is possible to trigger the eager execution of some function when a failure is detected. This is the way to guarantee timely reaction to failures and thus maintain correctness of the system (even when that means the orderly execution of a shutdown procedure).

7. Enforcing Synchronism Properties of the TCB

The TCB *can* be built in any way that enforces the TCB synchronism properties **Ps** 1, **Ps** 3 and **Ps** 2 stated in Section 4. The TCB *should* be built in a way that secures the above-mentioned properties with ⟨*bound*,*coverage*⟩ pairs that are commensurate with the time-scales and criticality of the application. In consequence, the local TCB can either be a special hardware module, or an auxiliary firmware-based microcomputer board, or a software-based kernel on a plain desktop machine such as a PC or workstation. Likewise, the distributed TCB assumes the existence of a timely inter-TCB communication channel. This channel can assume several forms that exhibit different ⟨*bound*,*coverage*⟩ values for the message delivery delay ($T_{D^3_{max}}$). It may or not be based on a physically different network from the one supporting the *payload* channel. Virtual channels with predictable timing characteristics coexisting with essentially asynchronous channels are feasible in some of the current networks, even in Internet [19]. Observe that the bandwidth required of the control channel is much smaller than that of the payload channel: local TCBs only exchange control messages. In a number of local area networks, switched networks, and even wider area networks, it is possible to give guarantees for high priority messages[18, 3, 2]. In more demanding scenarios, one may resort to alternative networks (real-time LAN, ISDN connection, GSM Short Message Service, Low Earth Orbit satellite communication).

In fact, a TCB can be built out of normal hardware, and this is the scenario that we consider here, as the most adequate to show the feasibility of the model. As shown in Figure 4, the TCB is set-up on a real-time kernel that sits directly on the hardware, so that it controls all time-critical resources (e.g., clock, scheduler, network interface). The regular operating system (e.g., Linux) is layered on top of the kernel. The placement of the TCB between the O.S. and the resources allows the TCB to monitor application calls and protect the kernel activity with regard to timeliness. The TCB offers a TCB- specific application programming interface (API)— presented in Section 5— which is provided to the payload applications together with the regular O.S. and system libraries interface. The API offers access to the timely execution, duration measurement and timing failure detection services. Note however, that applications not using the TCB need not be aware of the existence of the latter.

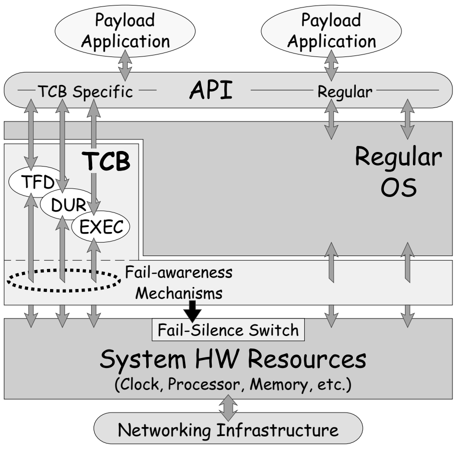


Figure 4. Block Diagram of a System with TCB

We are assuming the TCB to be fully synchronous. We are considering that the macroscopic services (timely execution, duration measurement and timing failure detection) always execute correctly. As such, we are also considering that the microscopic operations on the resources— code module executions and message transmissions— on which their correctness depends, are perfectly timely, as per the **Ps** properties. These are the foundations of a hard real-time device, which enforces the desired timeliness properties of the TCB services, namely by testing schedulability and actually scheduling computations and communication adequately.

However, there is always a risk that deadlines may be missed, mainly if sporadic or event-triggered computations take place [4]. We implement a few measures to amplify the coverage of the **Ps** properties, which consist in transforming unexpected timing failures into crash failures: we enforce fail-silence upon the first timing failure of a unit. We can afford to do that because these failures will be rare events, which could however compromise the safety of the system. Note that we *are not* allowing timing failures inside the TCB, so these are not tolerance measures, but safety measures: with this transformation, we bring some unexpected failures back into the assumed failure mode universe (crash). Note that there may be other, more sophisticated approaches to improve coverage of a TCB, and there may be distributed algorithmic approaches to a fault-tolerant TCB. Again, we wish to persuade the reader that a baseline implementation of this model can lead to very robust systems with simple mechanisms. Here is the approach and its validation:

- we only attempt to deal with unexpected failures in time (not in the value) domain;
- we assume property **Ps** 2 (clocks) to be always valid; we further assume that the same reliance can be put on clock-based operations in general, such as reading the local clock, and have the kernel set up alarms (watchdogs) that trip at a given time on the local clock;
- as such, we are concerned with the coverage of the more fragile properties: **Ps** 1 (processing) and **Ps** 3 (communication);
- we monitor the termination instants of local computations submitted to the kernel and compare them with the respective deadlines;
- we monitor the delivery instants of messages submitted to the kernel for transmission and compare them with the respective deadlines;
- should any deadline be missed, we enforce fail-silence of the unit observing the failure.

As depicted in Figure 4, the lower interface of the TCB with the system resources is under the surveillance of the monitoring mechanisms, based on fail-awareness techniques, i.e., techniques that allow the component to realize it has suffered a timing failure. As suggested in the figure, these mechanisms are hooked to a *fail-silence switch*, an abstraction whose implementation has the effect of immediately halting the whole site.

Improving the Coverage of Timely Processing

To improve the coverage of **Ps** 1 we use the local clock to measure local kernel execution durations. Recall that we assume that we can place more reliance on the timeliness of an alarm (watchdog), than on task scheduling. The kernel logs the start timestamp (T_s) of a time-critical computation with maximum termination time T_A, and sets an alarm for the desired deadline ($t_{dead} = T_s + T_A$). Either the computation ends in time, that is, until the deadline ($T_e \leq t_{dead}$) and the alarm does not trip, or else the alarm trips and causes the immediate activation of the fail-silence switch, crashing the whole site.

Improving the Coverage of Timely Communication

A similar principle can be used to improve the coverage of the bounded message delivery delay property (**Ps** 3). Message delivery delays are measured and compared to previously specified bounds. Round-trip duration measurement is used, since we are in the presence of a distributed duration. From a structural point of view, the idea is to apply the fail-awareness concept[12] to build a fail-aware broadcast as the basic kernel communication primitive to serve the TCB control channel. If a message is not delivered on time, an exception is raised that causes the immediate activation of the fail-silence switch, crashing the whole site.

Note that we cannot be as aggressive as with processing: we can only act on delivery of the message and not at the deadline instant. If we acted at the deadline point, we might be acting on either a crash or a timing failure. Whatever we did might not be appropriate in the case of crash, since we would be interfering with an assumed failure mode. For example, since we would crash a TCB that failed to receive a message until the maximum delivery time, the crash of a sending TCB (a normal event as per the assumptions) would cause all recipient TCBs to commit suicide, crashing the entire system. This would be inappropriate, since the crash of a TCB causes no safety problems, so we only crash TCBs after they receive a late message. On the other hand, with our technique, if a sending TCB or the network would cause all of the TCBs to receive a late message, then all the TCBs would crash as well. However, from a safety viewpoint this would be appropriate, in order to avoid contamination.

8. Conclusion

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. Such systems are characterized by having timeliness assumptions that may be violated, producing timing failures in components. An analysis of the effect of timing failures on application correctness shows that besides the obvious effect of delay, there are undesirable side effects: a long-term one, of decreased coverage of assumptions; and an instantaneous one, of contamination of other properties. Even when delays are allowed (e.g. soft real-time systems), any of these effects can lead to undesirable behavior of a system. Dealing with them requires some capability of acting timely at critical moments.

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. Special hardware is not mandatory to achieve synchrony of the TCB. The quality of that synchrony (speed, precision) is the only thing that may be improved by special components. By implementing only a small and simple part of the system, the TCB can affordably implement stronger properties. It acts in fact as a coverage amplifier, for the execution of certain functions where high assurance is desired.

We introduced a computational model based on the TCB, 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. Namely: we proposed a few services for the TCB to fulfill its role— timely execution, duration measurement, timing failure detection; and we devised an application programming interface allowing to propagate the notion of time from the TCB to payload applications. Finally, we discussed the implementation of the TCB services and the enforcement of the synchronism properties of the TCB platform, by using fail-awareness techniques.

We are currently developing an experimental prototype of a TCB. The infrastructure is composed of normal Pentium PCs, running Real-Time Linux, and communicating over a LAN. We expect to be able to publish the results of our experiments in the near future.

References

[1] 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.

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

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

[4] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages*. International Computer Science Series. Addison-Wesley publishers Ltd., 1996.

[5] A. Casimiro and P. Veríssimo. Timing failure detection with a timely computing base. In *Third European Research Seminar on Advances in Distributed Systems*, Madeira Island, Portugal, May 1999. Available as Tech. Report, Department of Informatics, University of Lisboa, DI/FCUL TR-99-8.

[6] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, Mar. 1996.

[7] F. Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3(3):146–158, 1989.

[8] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, pages 642–657, Jun 1999.

[9] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronization needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, Jan. 1987.

[10] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, Apr. 1988.

[11] D. Essamé, J. Arlat, and D. Powell. PADRE: A Protocol for Asymmetric Duplex REdundancy. In *Proceedings of the Seventh IFIP International Working Conference on Dependable Computing for Critical Applications*, pages 213–232, San Jose, California, USA, Jan. 1999.

[12] 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*, pages 282–291, Seattle, Washington, USA, June 1997. IEEE Computer Society Press.

[13] C. Fetzer and F. Cristian. A fail-aware datagram service. *IEE Proceedings - Software Engineering*, pages 58–74, April 1999.

[14] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, Apr. 1985.

[15] F. Jahanian. Fault tolerance in embedded real-time systems. *LNCS*, 774:237–249, 1994.

[16] E. D. Jensen and J. D. 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, Oct. 1990. IEEE Computer Society Press.

[17] H. Kopetz, R. Zainlinger, G. Fohler, H. Kantz, P. Puschner, and W. Schutz. An engineering approach towards hard real-time system design. *LNCS*, 550:166–188, 1991.

[18] M. d. Prycker. *Asynchronous Transfer Mode: Solution For Broadband ISDN*. Prentice-Hall, third edition edition, 1995.

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

[20] 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)*, 7(4):35–39, Winter 1995.

[21] P. 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, Nov. 1991.

[22] P. Veríssimo and A. Casimiro. The timely computing base. DI/FCUL TR 99–2, Department of Computer Science, University of Lisboa, Apr. 1999. Short version appeared in the Digest of Fast Abstracts, The 29th IEEE Intl. Symposium on Fault-Tolerant Computing, Madison, USA, June 1999.