Timing Failure Detection with a Timely Computing Base

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

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

Abstract

In a recent report we proposed an architectural construct to address the problem of dealing with *timeliness* specifications in a generic way. We called it the Timely Computing Base, TCB. The TCB defines a set of services available to applications, including timely execution, duration measurement and timing failure detection. We showed how these services could be used to build dependable and timely applications. In this paper we further extend the description of the TCB, namely by presenting a protocol for its Timing Failure Detection (TFD) service. We discuss the essential aspects of providing such a service under the TCB framework and make some considerations relative to the service interface.

1 Introduction

In the past few years we have been witness of distributed systems expansion and of the appearance of more demanding applications. While many of these applications can take advantage of growing system capabilities like processing speed, storage size or memory size, other have requirements, like real-time or fault-tolerance, that do not depend exclusively on hardware capabilities.

For instance, the implementation of services with high interactivity or mission-criticality requirements must be based on solid and adequate system models and correct software protocols. This kind of services are usually demanding in terms of *timeliness*, 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). An intuitive approach to cope with such timeliness needs is to use a synchronous system model. However, large-scale, unpredictable and unreliable infrastructures cause synchronous system models to work incorrectly. On the other hand, asynchronous models do not satisfy our needs because they do not allow timeliness specifications.

In order to clarify the problem and create a generic framework to deal with this problem, we introduced the **Timely Computing Base (TCB)** model[13]. It assumes that any system, regardless of its synchrony properties, can rely on services provided by a special module, the TCB, which is timely, that is, synchronous. In this paper we further extend the description of the TCB, namely by proposing a protocol for its Timing Failure Detection (TFD) service. We show that synchronized clocks are not required to implement such a TFD service. We also analyze the problem of node crashes – how this affects the TFD service – and propose a solution to deal with it. To keep the protocol generic we do not assume any specific environment or computational platform.

The rest of the paper is organized as follows. Next section presents some related work. Section 3 describes the TCB model and its services. Section 4 is dedicated to the presentation and discussion of the TFD service. In section 5 we discuss some issues related to the service interface. Finally, we conclude with a summary of what has been done and highlight some topics for future work.

2 Related Work

The problem of failure detection is strictly related to system assumptions. In the past few years, several authors have addressed this problem under different perspectives and assuming varying degrees of synchronism properties. One of the first known results, derived for fully asynchronous systems, describes the impossibility of distributed consensus in the presence of failures [11]. In such a time-free model the specification of a useful failure detector (one that could allow some progress) turns out to be impossible.

Chandra & Toueg proposed a classification model for failure detectors [5]. The merit of their work lies in the formal way the problem was treated, and how they managed to isolate and specify the exact properties that determine the possibility or impossibility of solving various distributed system problems like consensus, atomic broadcast or leader election.

The timed asynchronous model adds some synchronism to the system by assuming the existence of local hardware clocks with bounded drift rate [7]. This allows processes to measure the passage of time and use timeouts. In timed systems it is possible to construct a special failure detector, a fail-aware failure detector [8], which can be implemented if some additional progress assumptions are made. In [10], Fetzer proposes an ap-

*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. The Navigators Group Web page: http://www.navigators.di.fc.ul.pt/.

proach to calculate upper bounds for the transmission delay of messages inspired on the round-trip clock reading method [6]. Our work uses these results as a building block to construct the timing failure detection service.

The quasi-synchronous model introduces the notion of timing failure detectors [12]. In [2], Almeida describes a TFD service for the quasi-synchronous model which assumes local clocks are synchronized and is used as a specific tool to implement a group communication protocol providing total temporal order. In a more recent work, it was shown that the timing failure detector has such properties that allow a timely failure detection, so that, despite the eventual failure of synchrony assumptions, protocols can adapt to timing failures and allow the system to remain correct [3]. The TFD service we propose in this paper also has these properties. However, its construction is ruled by other objectives. First, we show that it is possible to timely detect timing failures without synchronized clocks. Second, we propose a self-contained service with a clearly defined objective, which is solely the provision of timeliness information about events occurring in the system. As opposed to the service presented in [2], no application related information can be sent through this service. Finally, we envisage a TFD service that gives more than just timeliness related information since it may keep, and manage, historical information about the system behavior.

3 The TCB Model

The assumed system model is composed of participants or processes (both designations are used interchangeably) which exchange messages, and may exist in several sites or nodes of the system. A communication network interconnects sites. 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.

In terms of fault assumptions, the system is assumed to follow an omissive failure model, that is, components *only do timing failures*— and of course, omission and crash, since they are subsets of timing failures— no value failures occur.

Given the above assumptions, systems have to face the problem of uncertain timeliness (bounds may be violated) while still being dependable with regard to time (timely executing certain functions). This can be achieved if processes in the system have access to a *Timely Computing Base*, that is, a component that performs the following functions on their behalf: timely execution, duration measurement, timing failure detection. In this paper we deal with the last of these functions, and define the protocols for a Timing Failure Detection service.

There is one local TCB at every site, fulfilling the following architectural principles:

Interposition - the TCB position is such that no direct access to resources vital to timeliness can be made in default of the TCB

Shielding - the TCB construction is such that it itself is protected from faults affecting timeliness

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

TCB components are assumed to fail only by crashing, i.e. they are fail-silent. Furthermore, the failure of a local TCB module implies the failure of that site. Each local TCB enjoys the following synchronism properties:

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 not have clocks.

A distributed TCB is the collection of all local TCBs in a system, interconnected by a communication means by which local TCBs exchange messages. The interposition, shielding and validation requirements must also be satisfied by the distributed TCB such as the communication among local TCBs, which must still be synchronous. Property **Ps** 3 refers to the determinism in the time to exchange messages among participants via the TCB:

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

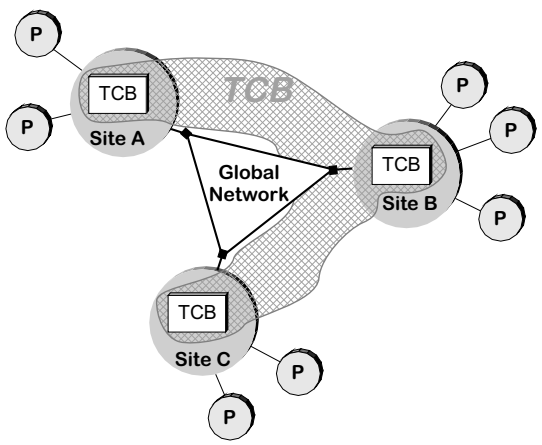

Figure 1: The TCB Architecture

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, fulfills the *interposition*, *shielding* and *validation* principles. Altogether, it preserves properties **Ps** 1 to **Ps** 3. The nature of the modules and the interconnection medium is outside the scope of this paper. The interested reader can refer to [13] where a few hints on how to implement a TCB are given.

4 The TFD Service

In this section we present and discuss a protocol for the TCB Timing Failure Detection service. We first introduce the formal definitions of *timed action* and *timing failure*, that are necessary to subsequently understand how the TCB, and more specifically the TFD service, handles application timeliness requirements. Given that, we introduce the properties required for the TFD service and briefly overview what they imply in terms of system model. We then propose a generic protocol and prove it satisfies those properties.

4.1 Timed Actions and Timing Failures

The TCB model makes a clear distinction between two classes of safety properties: logical safety properties, described by formulas containing logic and temporal operators, and timeliness safety properties, containing only time operators. For simplicity the former are called safety properties and the latter timeliness properties. A *timeliness property* 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. The TCB model formally defines the *within/from* operator, which is based on durations and is the most appropriate *time operator* to capture all relevant notions of time-related operations without ambiguity.

Whatever the underlying specification, the runtime enforcement of timely behavior can be expressed in terms of what we call a *timed action*, which we define as an event to be produced by a process and the latest real time instant when the event must take place:

Timed Action - Given process p, event e, and real time instant t_e, a timed action is: $X(p, e, t_e) \equiv$ *p produces e at* $t \leq t_e$

The correctness of the execution of a timed action may be affected by timing failures:

Timing Failure - given the execution of a timed action $X(p, e, t_e)$, there is a timing failure by p at t_e, iff 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, $Ld = t_e^{'} - t_e$

In consequence of an implementation dictated by a given property, a component must perform one, several or infinitely many *timed actions*. If a timed action does not incur in a timing failure, the action is *timely*.

4.2 TFD properties

As specified in the TCB model, the Timing Failure Detector service must have the following properties:

TFD 1 Timed Strong Completeness: *There exists* $T_{TFD_{max}}$ *such that given a timing failure of* p *in any timed action* $X(p, e, t_e)$*, the TCB detects it within* $T_{TFD_{max}}$ *from* t_e

TFD 2 Timed Strong Accuracy: *There exists* $T_{TFD_{min}}$ *such that given any 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*

These quite strong properties, defining a *Perfect Timing Failure Detector (pTFD)*, can only be guaranteed if certain conditions are met. The underlying system synchrony is the crucial factor dictating whether or not these properties can be satisfied. Remember we are now considering the environment under which the TCB and its services are run. In totally asynchronous, or *time-free* systems, it is obviously impossible to construct such a timing failure detection service since by definition the notion of time (thus of timing failures) is absent. Adding a short amount of synchrony, namely by allowing processes to access a local clock with bounded drift rate, it becomes possible to tackle problems with timeliness specifications. In particular, it is possible to detect *late events* and to construct fail-aware services [9]. However, achieving simultaneously the two properties required for perfect timing failure detection is still not possible [4]. That is only possible, in fact, if the model over which the TFD service is constructed is, at least in part, synchronous. This has implications both in terms of the communication medium and at the operating system level. There must exist a synchronous *control* channel interconnecting every local TCB module and the system must be scheduled in order to ensure that TCB tasks are hard real-time tasks, immune to timing faults in the other tasks.

4.3 TFD protocol

The problem we have to solve is how to build a timing failure detector which satisfies properties **TFD** 1 and **TFD** 2. This requires a protocol to be executed by all TFD modules on top of the above-mentioned synchronous control channel. To better understand the intuition behind the protocol we will proceed by steps and discuss some aspects we consider relevant.

Any timed action is related to an event that must take place at a specific time instant. We make a clear distinction between an *event* and its corresponding *timed action*. An event may occur at any given instant independently of what is specified in the timed action. The timed action

only defines the allowed interval of timely occurrence of the event.

We further consider the existence of two kinds of events. Those that occur locally to the TCB where the timed action was issued, and those that occur in a remote site. Then, we also distinguish timed actions relative to such events: the former are described by *local timed actions*, and the later by *remote timed actions*. Formally, a timed action $X(p, e, t_e)$ is local if it is generated by processor p and remote otherwise (when generated by some $q \neq p$).

In practice, local timed actions refer to local function executions and the timed event corresponds to the function termination. Remote timed actions are issued when a message is sent to another site. In this case the timed event is the remote receive event.

The timing failure of an event at a processor p can only be detected if that processor knows the deadline for the event, i.e. it must be aware of the timed action specification. For local events this is not a problem since the specification is locally available. However, for remote ones this requires the timed action specification to be delivered to the appropriate TCB module. This, in part, justifies the necessity of a protocol that allows TCB modules to exchange and share information. Another reason is due to the requirement for timely detection. In fact, timely detection can only be achieved if there is a protocol which forces the TFD to make a decision in bounded time. Simply waiting for an event to occur is certainly not sufficient.

Making decisions about timeliness of events is based on time values. According to the formal specification of *timing failure*, a TFD module detects a timing failure if the event occurs at a real time instant greater than the deadline instant specified in the timed action. Since the TCB model only assumes the existence of local clocks (property **Ps** 2) and does not even require clocks to be synchronized, reasoning in terms of a global time frame is not possible. Another methodology must be chosen. Using local clock values to specify deadlines is a simple and acceptable solution for local events. But for remote events absolute time values must be replaced by relative ones, that is, the arrival deadline on the remote local clock must be specified in terms of a duration related with the message send time. This requires some form of relating both clocks. The round-trip duration measurement technique described in [6] can be used for this purpose.

Following we present and describe the TFD protocol. Since local and remote timed actions can be treated differently, we first deal with failure detection of remote events and only then present a brief description of an algorithm to handle local timing failures.

Remote timing failure detection

The protocol that implements the "remote" part of the TFD service is presented in figure 2. It is executed in rounds, during which each TFD instance broadcasts all information relative to new (remote) timed actions and to timed events evaluated during last interval (since last round). The protocol uses three tables to store this information: a *Timed Actions Table* (TATable), an *Event Table* (ETable) and a *Log Table* (LTable). The TATable holds information about timed actions that must be delivered to remote sites during the next round. The ETable maintains information about receive events and their deadlines (specified in the timed actions), which will be used to make decisions about timeliness. The last table is where timing failure decisions are output and is also used to keep timeliness information of past events.

For each TFD_p instance

```
01 // T_send is the duration of send actions
02 // r is round number
03 // Π is the period of a TFD round
04 // C(t) returns the local clock value at real-time t
05 // R_TAT is the set of all records in TATable
06 // R_ET is the set of all complete records the in ETable
07
08 when user requests to send ⟨m⟩ to D do
09     mid := get_uniqId();
10     timed-send(⟨m, mid⟩, D); // to payload channel
11     insert (mid, D, T_send) in TATable;
12 od
13 when C(t) = rΠ do
14     broadcast (⟨R_TAT, R_ET⟩); // to control channel
15     r := r + 1;
16 od
17 when timed-receive(⟨m, mid⟩, q, T_mid, T_rec) do
18     if ∃R ∈ ETable : R.mid = mid then
19         R.T_mid := T_mid;
20         R.q := q;
21         if R.Complete = False then
22             stop (timer_⟨mid⟩);
23             R.Complete := True;
24         fi
25     else
26         insert (mid, T_mid, q, ⊥, False) in ETable;
27     fi
28     deliver (⟨m⟩, mid, T_rec, q) to user ;
29 od
30 when message ⟨R_TAT, R_ET⟩ received from q do
31     foreach (mid, D, T_send) ∈ R_TAT : p ∈ D do
32         if ∃R ∈ ETable : R.mid = mid then
33             R.T_send := T_send;
34             R.Complete := True;
35         else
36             insert (mid, ⊥, ⊥, T_send, False) in ETable;
37             start (timer_⟨mid⟩, T_send);
38         fi
39     od
40     foreach (mid, q, T_mid, T_send) ∈ R_ET do
41         if T_mid = ⊥ then
42             Failed := True;
43         else if T_mid > T_send then
44             Failed := True;
45         else
46             Failed := False;
47         fi
48         insert (mid, q, T_mid, T_send, Failed) in LTable;
49     od
50 od
51 when timer_⟨mid⟩ expires do
52     search R ∈ ETable : R.mid = mid;
53     R.Complete := True;
54 od
```

Figure 2: Timing Failure Detection protocol.

Activity within the TCB is triggered by a user request to send a message (line 8). We assume the TCB is capable of intercepting send requests, since it occupies a privileged position in the system. How this is done is an implementation issue out of the scope of this paper. Upon intercepting a message, a unique message identifier *mid* is generated (using some function *get_uniqId()*) and assigned to both the message and the timed action (lines 9-11). This identifier makes the association between a message and a timed action and it must be unique within the (distributed) TCB to avoid wrong associations.

The intercepted message is then sent to the payload channel using a special *timed-send* service, which inserts additional timestamping information in the message. This is required to allow the computation, at the receiver, of an upper bound for the effective message transmission delay. A detailed description of this technique can be found in [10]. The *timed-receive()* function, counterpart of *timed-send()*, delivers the measured transmission delay (T_{mid}) and the receive timestamp (T_{rec}) values (line 17).

After sending the message a new record is added to the Timed Actions Table (line 11). Each record contains the following items: the unique message identifier *mid*, the set of destination processes D and the specified duration for the send action, T_{send}. The value of T_{send} is kept by the TCB but may be changed at execution time. For instance, a Timeliness-Tuning Algorithm, as explained in [13] may do this. It is worthwhile to point out the generic and innovative perspective of assuming a dynamic system evolution in terms of timeliness parameters. In essence, this dynamic behavior allows a certain class of applications to adapt to environment changes and achieve *coverage stability*, as described in [13].

As said earlier, each TFD instance periodically disseminates new information concerning timed actions and event executions. The period Π depends on several factors, including the control channel bandwidth, the number of processes and the maximum amount of information sent in each round. Ideally, the value of Π should be the lowest possible to minimize the timing failure detection latency (see section 4.3). The TFD service wakes up, timely, when the local clock indicates it is time for a new round (line 13). The contents of the TATable and the *complete* records in the ETable are then broadcast on the control channel. A record is considered *complete* (and marked accordingly) when all the information necessary to make a decision has been collected, or when this decision can be made solely with the timed action information or if a special failure situation is detected (see section 4.3). The *Complete* field is not included as part of the message.

Synchronization among TFD instances is not enforced. Therefore, dissemination rounds of all instances may be spread in an interval of duration Π. However, since we assume bounded delays for TCB tasks (property **Ps** 1) and a synchronous control channel (property **Ps** 3), the inter-arrival interval of control information from a given instance is bounded. This knowledge can be used, as we will see, to detect the crash of a TCB module.

A message arriving from the payload channel is received with the already mentioned *timed-receive()* function (line 17). This function calculates the message transmission delay and it does so using send and receive timestamps of a round-trip message pair. Since the exact transmission delay cannot be determined, its upper bound is used instead. The error associated to this upper bound yields the value of $T_{TFD_{min}}$ specified in property **TFD** 2.

When the message arrives three situations are possible: (a) there is yet no information about the timed action and thus no entry in ETable for the message, (b) there is an entry which is not complete or (c) there is an entry marked as complete. In case (a) a new record is created for the message (line 26, $\perp$ denotes absence of value). In cases (b) and (c) the duration (T_{send}) for message *mid* is already known and an entry for the message exists so T_{mid} and q (the sender process identification) are added to that record (lines 18-20). If the record is not marked complete, this means the TFD was still waiting for the message to arrive and so $timer_{\langle mid \rangle}$ is stopped and $Complete$ is set to $True$ (lines 21-23). Otherwise, it means the message arrived late and a positive failure decision was, or will be made. Whichever is the case, the message is always delivered to the user along with the (TFD internal) message identification and the receive timestamp (line 28). Note that the aim of the TFD service is just to provide information about timing failures and thus no filtering of any kind is done to messages. Applications are free to handle the information provided by the TFD service in a manner that makes sense at their level of abstraction. But we will come back to this interesting interface problem in section 5.

Each message received from the control channel provides two kinds of information: timed actions records and complete event records. For a certain process p, the relevant timed actions are those of messages delivered to p (line 31). Durations of messages not yet received are inserted in ETable and a timer is started to allow a timely failure detection (lines 36-37). If the timer expires before the message arrives, the message will never be considered timely. Then, since we have to preserve property **TFD** 2 assuring that timely messages are never considered late, the smallest timeout value we can use is T_{send}. This value is obtained assuming that the duration in a timed action can be remotely known as soon as it is generated. If a more pessimistic, although realistic assumption was made, the timeout value could be relaxed to a lower value[1]. Nevertheless, this would not improve the maximum latency time for failure detection in the general case.

Complete event records are treated next. The TFD service compares the specified delivery delay with the measured one and assert a Boolean value to the variable $Failed$ (lines 41-47). If T_{mid} is empty ($\perp$) this means the message is late ($timer_{\langle mid \rangle}$ expired) and there is a timing

[1] At least, the minimum message delivery delay for the control channel could be taken into account.

failure. Each record in LTable contains the (TFD) message identifier, the sender, the specified and measured durations and the $Failed$ flag.

In the presented protocol we intentionally omitted the problem of table size and possible memory exhaustion to simplify the problem. Although it is simple to devise a solution to clean table records after the TFD has made the decisions, it may be useful to keep an history of timed action executions, and this raises the problem of choosing an adequate criterion to make the deletions. Solutions to this problem can only be dealt taking into account the possible uses of the information, and these depend on the application.

Local timing failure detection

As we said earlier, detection of local timing failures can be done more easily than remote ones. In fact, since all events in a same site can be timestamped using the same local clock it is easy to measure time intervals between events. Hence, it is simple for the TCB to measure the duration of any executed function.

For each TFD instance

```
01 // T_f is the duration of function f
02 // C(t) returns the local clock value at real-time t
03 // R_f is a local record for function f
04 // The R_f.T field keeps the specified duration
05 // The R_f.Start field stores the start timestamp
06 // The R_f.Run field stores the measured duration
07 // The R_f.Failed field indicates the failure decision
08
09 when user calls function f do
10      start (timer_f, T_f);
11      R_f.T := T_f; R_f.Run := ⊥; R_f.Failed := ⊥;
12      R_f.Start := C(t);
13 od
14 when function f terminates do
15      R_f.Run := C(t) − R_f.Start;
16      if R_f.Failed = ⊥ then
17           stop (timer_f);
18           R_f.Failed := False;
19      fi
20 od
21 when timer_f expires do
22      R_f.Failed := True;
23 od
```

Figure 3: Algorithm for local timing failure detection.

In figure 3 we present an algorithm to keep track of local timing failures. We assume that the TCB can intercept function calls and that recursive function calls are not allowed (an extended version of the algorithm could be devised to cope with this). We also assume that the specified duration T_f is known within the TFD service and is initialized to some value which can be changed by the application later on. Again we are not too worried with the interface. The interesting feature is that we can have a service that measures timeliness of functions executed by the application. Whether the application uses this service, and how, is another problem that we will tackle in a future paper.

Perfect timing failure detection

We are now able to state the following theorem.

Theorem 1 *The algorithms of figures 2 and 3 satisfy properties* **TFD** *1 and* **TFD** *2.*

The proof is obvious from the discussion of the protocols. Nevertheless we explain how the values of $T_{TFD_{max}}$ (**TFD** 1) and $T_{TFD_{min}}$ (**TFD** 2) are obtained. We only discuss the case for remote timed actions, since this is the harder one.

Consider the example depicted in figure 4. It illustrates a situation where a process p sends a message m to a process q with a specified duration of zero. Obviously, since no message can be sent instantaneously, a timing failure will occur as soon as the message is sent. Clearly, no timing failure can occur sooner than this. At worst, the TFD of processor p wakes up and broadcast the information about m into the control channel Π time units after the timing failure. This information is delivered at most Δ time units later to the TFD of processor q. It is inserted in the Event Table and the record for message m is marked as complete. It is so independently of m having arrived, since the timeout of $timer_{\langle mid \rangle}$ is set to zero (the value of T_{send}) and hence will expire immediately. Then, it may be possible to wait another $\Pi + \Delta$ time units until TFD_q disseminates the complete record is to all TFD instances. Only then the decision about the timed action will be made, that is, at most $2(\Pi + \Delta)$ after the timing failure.

The value of $T_{TFD_{max}}$ is then $2(\Pi + \Delta)$.

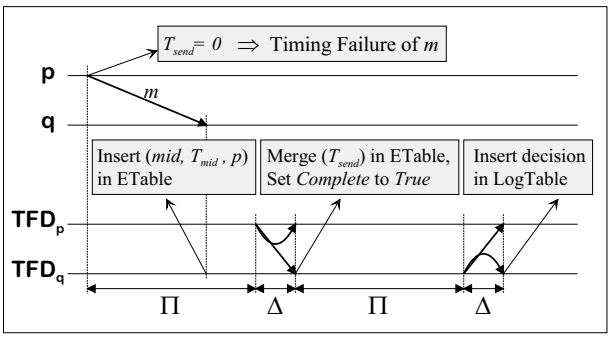

Figure 4: Example of earliest timing failure and maximum detection latency.

The value of $T_{TFD_{min}}$ derives from the error of the delivery delay measurement. Since the exact value of this delay is unknown, the higher bound is used to assure that a late event is never considered timely. In our protocol, the message delivery delay is measured by the *timed-send* service, which delivers the upper bound value. The associated error depends on the drift rate of local clocks (ρ_p, ρ_q), on the maximum drift rate (ρ), on the send and receive timestamps of a round-trip message pair $\langle m1, m2 \rangle$ $(S_{m1}, R_{m1}, S_{m2}, R_{m2})$, on the minimum message delivery delay (δ_{min}) and on the measured delivery delay of message $m1$ (T_{m1}). Assuming that $m1$ is first sent from q

to p and then $m2$ from p to q, the error associated to the transmission delay of $m2$ can be expressed as follows [10]:

$$e(m2) = (R_{m2} - S_{m1})(\rho + \rho_q) + (S_{m2} - R_{m1})(\rho - \rho_p) + (T_{m1} - \delta_{min})$$

The value of $T_{TFD_{min}}$ is then $e(m)$.

Impact of crash failures

The discussion of possible implications of a TFD instance crash was left to this point since it raises some model related questions that, if presented earlier, could confuse the protocol explanation. We also did so because we are convinced that the cases described below do not compromise what has been said until now.

There are two situations in which the crash of a TFD instance must be carefully analyzed to prevent the misbehavior of remaining instances or, even worse, incorrect information to be output. The key issue is the loss of information that, in this case, concerns timed action durations and complete event results.

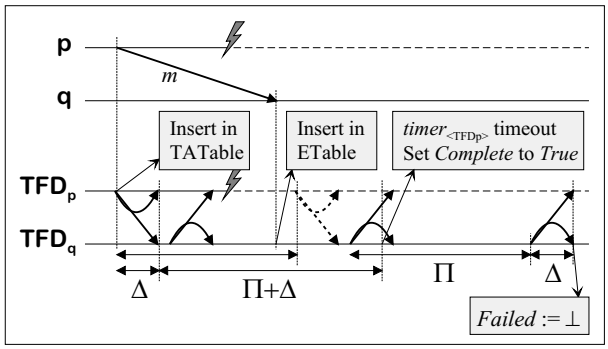

Figure 5: Example of crash failure before timed action durations are sent to control channel.

Figure 5 illustrates a situation in which the information contained in the TATable is lost. When process p sends m to q, TFD_p stores the timing information of m in TATable. Normally, this information would then be delivered to TFD_q and inserted in ETable. However, if TFD_p crashes before sending the control message, the information will be lost and therefore it will be impossible to decide about the timeliness of m. At first glance, this impossibility may seem a violation of properties **TFD** 1 and **TFD** 2. However, the fact that a crash failure occurred can be detected and this allows a special decision to be taken without knowing the specified duration for m. Thus, while the TFD service cannot say if message m was timely or late, it indicates that a crash of processor p occurred and that no decision can be made. It is then up to the application to handle this information and act upon it.

A generic solution to the problem of crash detection was discussed by Almeida in [1]. In what follows we present a specific extension to the TFD protocol to handle TFD crashes, and then finalize the discussion of figure 5.

For each TFD_p instance

```
01 // Π is the period of a TFD round
02 // Δ is the Max delivery delay in control channel
03 // C is the set of crashed processes
04
05 when message ⟨R_TAT, R_ET⟩ received from q do
06     (re)start (timer_⟨TFD_q⟩, Π + Δ);
07     if q ∈ C then remove q from C fi
08     (... treat R_TAT as before ...)
09     foreach (mid, r, T_mid, T_send) ∈ R_ET do
10         if T_send = ⊥ then
11             Failed := ⊥;
12         else
13             (... decide as before ...)
14         fi
15         insert (mid, q, T_mid, T_send, Failed) in LTable;
16     od
17 od
18 when timer_⟨TFD_q⟩ expires do
19     add q to C;
20     foreach R ∈ ETable : R.q = q do
21         R.Complete := True;
22     od
23 od
24 when timed-receive(⟨m, mid⟩, q, T_mid, T_rec) do
25     if q ∈ C then
26         insert (mid, T_mid, q, ⊥, True) in ETable;
27     fi
28     deliver (⟨m⟩, mid, T_rec, q) to user ;
29 od
```

Figure 6: Extension of the TFD protocol to handle crash failures.

The extension protocol uses a timer for each active process, which is restarted every time new control information is received from that process (line 6). Since the channel is synchronous, the interval between the reception of successive control messages from a given process is bounded by $\Pi + \Delta$. Hence, if the timeout is set to this value, the timer will expire only if that process has crashed. In the figure it is possible to observe when TFD_q detects the crash of p. Since a decision has to be delivered, all pending events are marked as complete (lines 20-22). A set containing all crashed processes is maintained so that future decisions (for messages still in transit) can be taken (lines 25-26). At decision time, if the field containing the value of the specified duration is empty ($\perp$), the special empty value $\perp$ will also be assigned to $Failed$ in the Log Table, meaning that it was not possible to make a decision (lines 10-11).

Having considered the possibility of crash failures, the value previously obtained for $T_{TFD_{max}}$ must be reconsidered. In fact, since the detection of a process crash may take longer than the reception of control information, we get a value slightly higher than before. Observe figure 5 and suppose TFD_q only sends the control message Π time units after detecting the crash. Consider also that for the purpose of calculating $T_{TFD_{max}}$ we may admit that m had a timing failure as soon as it was sent. Then, the new value will be $2\Pi + 3\Delta$.

The other potential problem is due to the loss of information in the Event Table. Since this table contains results of event executions, an intuitive approach would

make them immediately available (in LTable) at the local site. However, if a crash occurred before the broadcast of those results, this would mean a decision had been taken in one site and not in the others. To prevent this, we assure that decisions are only made upon reception of ETable records, and so either all TFD instances or none will output timeliness decisions.

5 TFD Service Interface

We have seen that remote timed actions are generated upon interception of send requests and that local function calls are also intercepted by the TFD. This mode of operation was intentionally devised to obtain a transparent service invocation. This means that applications do not have to be modified to explicitly request a timeliness evaluation of each action they perform. Instead, they are allowed to configure timeliness parameters, namely by defining the required duration for a certain kind of action, and then only have to query the service to collect the results. This transparency can be useful to deal with different kind of applications, specifically in terms of their synchrony assumptions.

The main source of information output by the TFD service is the Log Table, where timeliness information about events is kept. The idea to access this information is to have some kind of event identifier that is used to query the TFD service. This is why messages delivered to applications are accompanied by the identifier *mid* and by the receive timestamp. The application may not use these values, but if it wants it may use the timestamp to know when will the timeliness information be available (remember the $T_{TFD_{max}}$ constraint) and the *mid* to obtain it.

For a certain kind of applications it may be useful to know the activity in the payload channel. For instance, it may be interesting to know if there are any messages sent by some processor, which are supposed to be received. We know that the TFD service may be able to provide this information since it has a description of timed actions registered in the Event Table. So it may be convenient to provide an interface to access this particular table.

6 Conclusions

The TCB model provides a framework to deal with application timeliness requirements. It defines a number of services to be required including a Timing Failure Detection service. This paper has discussed several important aspects related to the provision of this service and a protocol with perfect timing failure detection properties has been presented. We have shown that clock synchronization is not an essential requirement to be able to timely detect timing failures. Some general issues relative to the TFD service interface have finally been discussed.

To conclude, we consider the implementation of a fully-fledged TCB prototype to be a long-term goal. We are currently investigating the requirements for an adequate and generic interface between control and payload modules exhibiting any degree of synchrony.

Acknowledgments

The authors are grateful to Christof Fetzer and Flaviu Cristian for a lot of discussions on these issues.

References

[1] Carlos Almeida. *Comunication in Quasi-Synchronous Systems: Providing Support a for Dynamic Real-Time Applications.* PhD thesis, Instituto Superior Técnico, January 1998. (in Portuguese).

[2] 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.

[3] 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*, pages 430–439, Madrid, Spain, December 1998. IEEE Computer Society Press.

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

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

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

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

[8] Christof Fetzer and Flaviu Cristian. Fail-aware failure detectors. Technical Report CS96-475, University of California, San Diego, October 1996.

[9] 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.

[10] Christof Fetzer and Flaviu Cristian. A fail-aware datagram service. In *Proceedings of the 2nd Annual Workshop on Fault-Tolerant Parallel and Distributed Systems*, Geneva, Switzerland, April 1997.

[11] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the Association for Computing Machinery*, 32(2):374–382, April 1985.

[12] 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.

[13] Paulo Veríssimo and António Casimiro. The timely computing base, February 1999. Submitted to EDCC-3, Third European Dependable Computing Conference.