Generic Timing Fault Tolerance using a Timely Computing Base

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

Paulo Veríssimo
pjv@di.fc.ul.pt
FC/UL

Abstract

Designing applications with timeliness requirements in environments of uncertain synchrony is known to be a difficult problem. In this paper, we follow the perspective of timing fault tolerance: timing errors occur, and they are processed using redundancy, e.g., component replication, to recover and deliver timely service. We introduce a paradigm for generic timing fault tolerance with replicated state machines. The paradigm is based on the existence of Timing Failure Detection with timed completeness and accuracy properties.

Generic timing fault tolerance implies the ability to dependably observe the system and to timely notify timing failures, which we discuss in the paper. On the other hand, it ensures replica determinism with respect to time (temporal consistency), and safety in case of spare exhaustion. We show that the paradigm can be addressed and realized in the framework of the Timely Computing Base (TCB) model and architecture. Furthermore, we illustrate the generality of our approach by reviewing previous existing solutions and by showing that in contrast with ours, they only secure a restricted semantics, or simply provide ad-hoc solutions.

1 Introduction

Dealing with applications with timeliness requirements in environments with poor baseline synchrony is known to be a difficult problem. During their execution these applications have to face the possible occurrence of timing failures, when they do not perform the specified timed actions within the specified deadlines. This raises a number of possible consequences, such as instantaneous delays, decreased coverage over the long term or contamination of logical safety properties [23]. A general approach to timing fault tolerance has to deal with all these aspects and must provide adequate solutions or techniques to handle each of the possible effects of timing failures.

Choosing a distributed systems model relevant to the provision of the stipulated services and coverage to a chosen fault model is of fundamental importance. However, well known models like the fully asynchronous, also called time-free model, or the synchronous model, are not the most appropriate. The former does not allow to handle timeliness requirements and the latter does not faithfully characterize these uncertain environments and thus may lead to incorrect behaviors. For example, an application whose safety depends on meeting a bound will be unsafe if the bound, being assumed, cannot be guaranteed due to the uncertainty of the environment. A better way to deal with the problem is to use partial synchrony models. For instance, the Timed Asynchronous Model [10], which allows the construction of fail-aware applications [12], and the Quasi-Synchronous Model [22], which assumes the existence of synchronous components that allow certain real-time actions to be executed. Our approach has been to propose a more generic model, encompassing the whole spectrum of synchrony. We observed that synchrony is not a homogeneous property of systems– it varies with time or with space– and this led to the definition of the Timely Computing Base (TCB) model. Applications designed under the TCB model can rely on the services of a synchronous component, a TCB module, to achieve a timely and safe behavior.

We have been systematically detailing how the effects of timing failures can be handled, with the help of a TCB, to implement certain classes of applications. In this paper we focus on the problem of timing fault tolerance as a means to increase the availability of applications with timeliness requirements. We show that the problem can be handled in a generic way in the context of the TCB model, using redundancy techniques. For that purpose we introduce a paradigm for generic timing fault tolerance using a replicated state machine based server. The paradigm assumes the existence of a timing failure detector with timed completeness and accuracy properties, which is used to ensure the temporal consistency of the server state.

Using replication in the design of fault tolerant systems is not a novel idea [2]. However, there is not much work

*Faculdade de Ciências da Universidade de Lisboa. Bloco C5, Campo Grande, 1749-016 Lisboa, Portugal. Navigators Home Page: http://www.navigators.di.fc.ul.pt. This work was partially supported by the EC, through project IST-2000-26031 (CORTEX), and by the FCT, through the Large-Scale Informatic Systems Laboratory (LASIGE).

dealing with replication for *timing fault tolerance*. Among the exceptions, the work in [1] proposes a solution for timing fault tolerance based on the replication of system components and on the definition of light-weight groups on top of a quasi-synchronous network. Our approach is more generic in the sense that it is suitable for any timed service executing as a state machine. Furthermore, we reason in terms of Quality of Service (QoS) specifications to express timeliness and reliability requirements. Torres-Rojas et al. [20] propose a model of *timed consistency*, which requires the effects of a write operation to be observed in all the sites of a distributed system within a known and bounded amount of time. This work uses time to explore the semantical aspects of the problem, without being concerned with system synchrony or timeliness. Another work that deals with server replication for improved availability has been presented in [16]. It follows the perspective that client applications specify their timeliness requirements through a QoS specification and proposes a protocol that takes into account this specification to select an adequate set of server replicas to which requests should be sent. This work has now been extended in [17] to address the tradeoff between timeliness and consistency requirements when performing read and write operations on server replicas.

We offer a system model and a paradigm for generic timing fault tolerance. We provide a detailed discussion of the potential problems for the timely and consistent behavior of a replicated server caused by timing failures, and we enumerate the basic requirements that must be fulfilled in order to avoid these problems. Then we show that the TCB model provides an adequate framework to address the problem. The rest of the paper is organized as follows. In Section 2 we describe the system model and we review some of the facets of timing fault tolerance. Then, in Section 3 we introduce the generic paradigm for timing fault tolerance using a replicated state machine. In Section 4 we briefly present an overview of the TCB model, before we describe, in Section 5, how this model can provide an adequate framework to address problem of timing fault tolerance using the paradigm introduced in Section 3. In Section 6 we discuss relevant related work in the context of the proposed paradigm for generic timing fault tolerance. Section 7 concludes the paper.

2 System Model

We consider applications with timeliness requirements, executing in an environment where it is not possible to always guarantee time bounds for fundamental variables, such as message delivery delay or processing time. Let us consider a **timed action** as one which must terminate within a certain interval from a reference instant. Because of the above, applications are subject to **timing failures**, which occur when the execution of a timed action does not terminate within the specified time bounds. Furthermore we consider these failures to occur sporadically, and independently. Throughout the paper we use the term "timing fault tolerance" meaning that individual components or actions are affected by timing failures, whose system-level effect (a "timing fault") we wish to tolerate.

The effects of timing failures can be handled in different ways, depending on the properties of the particular application. For example, *fail-safe* applications can handle individual timing failures by switching to a fail-safe state as soon as the timing failure occurs and is detected [24]. Other applications can remain correct even when occasional timing failures occur, but they expect a certain coverage or probability of success relative to an assumed time bound. These applications must be *time-elastic*, which means that they can handle the decreased coverage effect caused by too many timing failures by adapting to new time bounds [7].

Another way of dealing with timing failures is by replicating the components affected by these failures. In this case, which we address here, it is necessary to ensure that applications remain *time-safe*, that is, that they remain consistent as they progress despite the occurrence of timing failures [23]. We consider a client-server operation model in which the server is replicated and part of it can be implemented as a deterministic state machine [19]. Before introducing the paradigm for generic timing fault tolerance, let us describe the correctness criteria that we follow and some basic assumptions relative to the interaction and to the QoS models considered in this paper.

2.1 Correctness Criteria

The correctness criteria for our paradigm rely on two premisses: a) ensuring the value consistency of the updates to the server replicas state; b) securing the *temporal consistency* of *time-value entities*. Any generic solution for timing fault tolerance must ensure that these requirements are fulfilled despite timing failures.

Since we are considering, for the case of write interactions, that the replicated server operates as deterministic state machine, we must ensure that write requests are processed in the same order by all the replicas in the system, so that the value of their state remains consistent.

Beside this value consistency requirement, it is necessary to ensure that both write and read operations respect timeliness requirements. Such requirements are very important in several areas of computing, namely in real-time control, in real-time databases, or in clock synchronization. We explain below the relevant correctness criterion[25]:

- A time-value entity is such that there are actions on it whose time-domain and value-domain correctness are inter-dependent (e.g., the position of a crankshaft, the temperature of an oven). Real-time problems can, almost without exception, be formulated in terms of

reading, computing, and updating the state of time-value entities. For the correct operation of systems using these entities two problems must be solved: a) ensuring the timely observation of the entity; b) ensuring the timely use of the observation afterwards.

- This implies the establishment of validity constraints for the computer representations of time-value entities (which make up the server state), and bounded delays for their manipulation (reading, writing, computing).

- Furthermore, the fundamental consistency criterion between concurrent operations, causality, must be equated in terms of time, which again can be guaranteed by the establishment of known and bounded intervals between events [21].

These requirements are captured by the notion of temporal consistency. Consider the value of a time-value entity at instant T_i, $E_i(T_i)$. Assume bound $\mathcal{V}_a$ for the maximum acceptable error accumulated by the observation (a computer variable) of a time-value entity made at T_i, over time. Then, we say that the observation is **temporally consistent at** $t_a \geq T_i$, if and only if the value of the time-value entity at t_a differs less than $\mathcal{V}_a$ from the observation ($|E_i(t_a) - E_i(T_i)| \leq \mathcal{V}_a$). That is, we are defining the faithfulness of an observation.

Therefore, when using such observed values to update the state of a replicated state machine, and later when reading them to perform real-time tasks, it is necessary to guarantee that the requests will be processed within bounded intervals. This means that it is possible to secure temporal consistency if there is an interval $\mathcal{T}_a$ such that the variation of the value of the time-value entity within that interval is at most $\mathcal{V}_a$. This interval, also called *temporal accuracy interval* for control [14], constitutes the relevant variable which determines the bounds that must be observed during execution in order to detect timing failures.

2.2 Interaction Model

We assume a distributed system composed of several clients that communicate through a network with a server. Client processes can execute read or write interactions (by sending read or write requests to the server) to respectively retrieve or modify (parts of) the server state. Furthermore, we assume that read interactions are independent from write interactions, which means that the server does not have to enforce any ordering criteria between read and write requests. It also means that read requests, which are the most frequent, may be served more rapidly and more efficiently, thus achieving an overall performance improvement.

We assume that there is a process in the server responsible for handling write requests, which executes as a deterministic state machine. Since we replicate the server for timing fault tolerance and to implicitly increase its availability, the process handling write requests is also replicated. Therefore, when considering write requests the server can be seen as a replicated state machine. On the other hand, read requests are made to one or more servers, and served immediately they are done, concurrently with write requests, internal concurrency control (e.g., critical sections) not withstanding. This allows the server to process the two types of requests in an independent manner, for instance using concurrent threads of execution, one for the writes and the others for the reads. This independence is particularly relevant if we consider that the frequency of read requests is much higher that that of write requests (e.g., interactive web servers, real-time databases).

2.3 QoS Model

Timing specifications are usually derived from application timeliness requirements. In the current context we follow an end-to-end approach, in which we take into account the delay of interactions (messages transmitted and/or received from the server) in addition to the response time of the server. These are the relevant timeliness requirements for the clients interacting with the server.

Since we are not considering an environment of guaranteed behavior, time bounds may be violated during execution. An adequate approach for dependable operation is to consider that time bounds have an associated measure of the probability that they will hold during an interval of execution. This measure is called **coverage** (of the assumption) [18]. Therefore, we assume that QoS requirements can be specified through ⟨bound, coverage⟩ pairs, that is, by defining a bound that should be secured with a given coverage. A generic timing fault tolerance approach must ensure that despite the occurrence of timing failures of read or write interactions, the system (including clients and the replicated server) will remain correct and timely, and QoS specifications of the form ⟨bound, coverage⟩ will be secured. Incidentally, we should point out that this approach may be applied in soft, as well as in mission-critical real-time systems design. Timeliness of execution is guaranteed with a certain coverage (the QoS specification) and, should a QoS failure be detected, safety measures (e.g., fail-safe shutdown), or real-time adaptation (e.g., reducing the system requirement), or QoS renegotiation can be undertaken, depending on the application characteristics [25].

3 A Paradigm for Timing Fault Tolerance

To analyze the effects of timing failures on the correctness of the replicated server and to derive the paradigm that will allow us to achieve timing fault tolerance with a replicated state machine, we will address read and write interactions separately.

3.1 Read interactions

When considering read interactions, the server can be seen as a simple information server that replies to client queries. The key issue we have to deal with is that these queries have timeliness requirements. The relevant timed action is the query operation, which includes the time to send the query request, the time to process it and the time to send the reply. A timing failure occurs if the reply is not delivered to the client within the specified time bound. By replicating the server, and by sending the query to multiple replicas, the probability of receiving a timely reply (from one of the replicas) can be increased. Note that any reply can be used, provided that it has been sent by a correct (which implies timely) replica.

The effect of individual timing failures in the case of read interactions can simply be masked by replicating the server, assuming that at least one of the replies is timely. But there is always a chance that multiple timing failures occur, causing the replicated query operation to fail. The idea is to reason in terms of the QoS specification, that is, in terms of the coverage required for the timeliness of the query. An adequate number of replicas– not too few, to ensure that the coverage is secured, not too many, to avoid unnecessarily increasing the load of the system– should be used. Finding an adequate number of replicas raises two issues:

- First, it is necessary to observe the behavior of individual replicas and accurately estimate the coverage that can be expected for each of them.
- Second, since some replicas may be timelier than others, it is necessary to find a convenient set (providing the desired coverage), preferably containing as few replicas as possible.

Observing the execution of generic timed actions requires the measurement of both local and distributed **durations**. Although the overall duration of query operations can be measured on a local basis, sometimes it may be convenient to measure the partial terms that contribute to this overall duration. This implies measuring distributed durations such as those relative to message transmission delays. The possibility to measure durations of individual interactions, allows to differentiate the coverage associated with each replica. In [7] we describe a methodology to estimate with a bounded error the duration distribution function and the expected coverage of a timed action, based on the history of measured durations.

To address the second issue, we propose to devise an algorithm that takes the ⟨bound, coverage⟩ QoS specification and uses the distribution function to select the set of replicas that should be used [23]. The work presented in [16], specifically concerned with this issue, proposes a solution that uses local duration measurements to calculate response time distribution functions of query operations, which are then used to select the convenient replica set.

If the number of available replicas is not sufficient to ensure the desired coverage, then the QoS specification must be dynamically modified, increasing the requested time bound or reducing the target coverage.

3.2 Write interactions

Dealing with interactions that modify the state of the replicated state machine raises several additional problems. We consider that for the case of write interactions the interesting timed action consists on the transmission of an update message and its consequent processing by the receiving replica. The timed action terminates when the state machine has modified its internal state.

Similarly to what we have mentioned earlier for the case of read interactions, we could also decompose a write interaction into two basic timed actions, which could be observed individually: the transmission of the write request through the network and the actual write operation in the replica. However, for the purpose of showing that the state machine behaves correctly and timely, we simply need to observe the global timed action. We do not care about the precise instant at which a write request is received, provided that the replica is able to process the request and update the local state before the requisite global deadline.

Since we need to ensure that updates are delivered to replicas in total order (to obtain a deterministic replicated state machine) it is necessary to use a message delivery service that provides this semantics. For instance, this can be done using a protocol designed for timed models [9].

The difficult problem to be solved consists in ensuring temporal consistency in the presence of timing failures. When a timing failure occurs, that is, when one of the replicas does not update its state in a timely fashion, it will become inconsistent with the other replicas. If nothing is done to prevent this situation, then an undesirable **contamination** effect can be observed. The effect is illustrated in Figure 1.

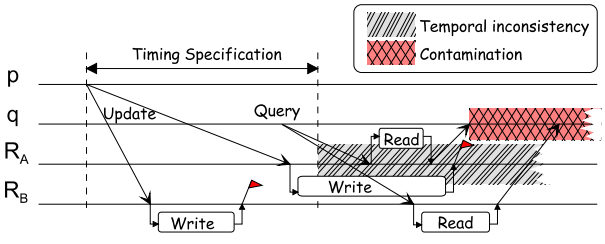

Figure 1. The contamination effect.

In the example, some process p sends an update to the replicated state machine, which is received and processed by replicas R_A and R_B. There is a timing specification to

express the required timeliness of the write interaction. The arrows representing the transmission of the update include the time to achieve a total order delivery. The write operation, executed after the request is received, must be terminated within the specified duration. Otherwise there will be a timing failure and the late replica will become temporally inconsistent. Process q sends a query to replicas R_A and R_B, and waits for the first reply. Since we assumed that the server may use independent (but properly synchronized) threads to concurrently serve read and write requests, the first reply may be received from the temporally inconsistent replica (R_A), and process q may use a temporally incorrect value and become contaminated.

To avoid contamination, something must be done in order to prevent the affected replica to propagate its inconsistent state to other elements in the system. For instance, one can remove the affected replica from the set of available replicas. This translates into the following two basic requirements. It is necessary to a) detect timing failures in a timely manner; b) guarantee a timely reaction after failure detection.

The first requirement implies a form of **timing failure detection**. As mentioned in the beginning of this section, timing failures occur when some timing specification is violated. Therefore, any timing failure detection service must observe the execution of timed actions and must be aware of the relevant deadlines. Furthermore, for the detection to be useful it must be done in a bounded amount of time. In Section 5 we explain how the bounded detection latency is crucial to enforce application timeliness requirements. The second requirement implies a form of **timely executing** certain actions. Solutions to this problem must take into account existing results in the realm of real-time systems.

Since the solution for the contamination problem relies on the information provided by a timing failure detection service, the properties of this failure detector are mostly relevant. Informally, the timing failure detector should exhibit the following properties:

Completeness: All timing failures must be detected, that is, no faulty timed action should ever be considered timely. This property is required to guarantee that faulty behaviors are never allowed.

Accuracy: Timely timed actions should not be wrongly detected as timing failures. Sometimes the failure detector may make (false positives) mistakes, but it must be possible to establish a bound to limit the situations in which these mistakes can take place (*see* Section 5). This property is required to establish the usefulness of the failure detector.

Another important issue is related with the scope of failure detection. Failure notifications should be delivered to all interested participants in order to allow the execution of coordinated actions upon failure events. For instance, it may be desirable to switch in a coordinated way to a degraded mode of operation after a replica is disabled. Therefore, in addition to the above properties, an useful timing failure detector must ensure that timing failures are detected by **all processes** in the system. Since all processes in the system (clients and server replicas) may have the same view about timing failures, late replicas can be removed from the set of available replicas everywhere.

Reducing the number of available replicas will also reduce the probability of subsequent read or write interactions to be executed on time. Let us now reason about the coverage of write interactions and its relation with the number of available replicas. Remember that given a certain QoS specification of the form $\langle$bound, coverage$\rangle$, the objective is to ensure that the required coverage can be achieved for that bound.

Since a write interaction must be performed using all available replicas (to ensure strong consistency), its overall coverage depends on the number of available replicas and on the coverage that can be expected from each of them. It is obviously much higher than it would be with a non-replicated solution and typically higher than the coverage required by the QoS specification. When timing failures occur and replicas are removed the decreased coverage effect can be observed, that is, the overall achievable coverage is reduced. This is acceptable in write interactions, and nothing has to be done, provided that the achievable coverage is still higher than the desired one. However, it may be possible that too many replicas fail, bringing down the achievable coverage to a level lower than the required one. In this case the system can react in two ways: a) by adapting the timeliness requirements, in order to maintain the desired coverage; b) by starting new replicas, in order to increase the availability and, consequently, the achievable coverage. The former solution requires the system to be adaptable and does not solve the problem of decreased replication level. The latter is a good long term solution, but may require too many system resources and time to start some new replicas. Since we are talking about timing failures, and not crash failures, in some cases it may be possible to simply wait for removed (late) replicas to be re-enabled, and in that way reestablish the replication level. This approach is discussed with more detail in Section 6.

It should be clear at this point that the use of replication can improve the availability of a system with timeliness requirements. But this is only achievable if the correct measures to prevent the negative effects of timing failures are taken. In summary, any effective solution based on the paradigm for generic timing fault tolerance using a replicated state machine must be able to: 1) measure local and distributed durations; 2) timely detect timing failures, in a complete, accurate and distributed manner; 3) guarantee a timely reaction after the failures are detected.

4 TCB Overview

4.1 TCB model

The Timely Computing Base model has been devised to provide a generic framework for systems of partial synchrony, such as the system we assume in this paper. Therefore, it is our goal to show that the TCB model can be used to achieve timing fault tolerance with a replicated state machine. Although in this paper we just present a short description of the model, a complete description can be found in [23] and [24].

A system with a Timely Computing Base is divided into two well-defined parts: a *payload* and a *control* part. The generic or *payload* part constitutes what is normally 'the system' in homogeneous architectures. It exists over a payload network and is where applications run and communicate. The *control* part is made of local TCB modules, interconnected by some form of medium, the *control* network. Processes execute on several sites, making use of the TCB whenever appropriate. We assume only crash failures for the TCB components, i.e. that they are fail-silent. Furthermore, we assume that the failure of a local TCB module implies the failure of that site, as seen from the other sites.

The payload part can have any degree of synchronism, and the control part (the TCB) is assumed to be a synchronous component exhibiting known upper bounds on processing and message delivery delays, and on the rate of drift of local clocks. Note that these synchrony properties are only required for a very small part of the system, and thus can be secured much more easily than if we were considering the overall system. A discussion about TCB implementation issues can be found in [4]. A TCB is a tiny but effective subsystem providing time-related services to applications or middleware components executing in the payload part of the system.

4.2 TCB services

In order to keep the TCB simple, which is fundamental to ensure the required synchrony properties, only the services considered essential to satisfy a wide range of applications with timeliness requirements have been defined. These services satisfy the requirements enumerated in the end of Section 3. They include a **Duration Measurement** service, a **Timely Execution** service and a **Timing Failure Detection (TFD)** service. The duration measurement service allows the measurement of arbitrary durations with a known bounded error. The timely execution service allows the deterministic execution of some function given a feasible bound T, with the possibility of specifying an execution delay, as those resulting from timeouts. Finally, the TFD service has *Timed Strong Completeness* and *Timed Strong Accuracy*, which describe the properties that a *perfect Timing Failure Detector (pTFD)* should exhibit. We use an adaptation of the terminology of Chandra [8] for the timed versions of the completeness and accuracy properties.

5 Using the TCB for Timing Fault Tolerance

In this section we explain how the TCB services can be used in order to handle timeliness requirements and ensure a correct behavior of the replicated state machine. We focus on the TCB interface functions that are relevant for this paper, namely those related with the TFD service.

5.1 Duration Measurement

The duration measurement service can be used in the construction of the system to measure the duration of read and write interactions. However, since it is necessary to detect timing failures, this service alone does not fulfill all our needs. The adequate approach, in this case, is to use the TFD service interface, which also delivers information about measured durations. In fact, the duration measurement service is used as a building block for the TFD service. Note that we are not concerned with the particular implementation of the duration measurement service, provided that it allows to bound the measurement errors. Example implementations can be found in [5] and [11].

5.2 Timely Execution

The timely execution service plays an extremely important role in the context of this work. In fact, it can be used to guarantee that some appropriate function is timely executed when a timing failure is detected in one of the replicas. The service allows arbitrary functions to be executed by the TCB, provided that certain requirements are met. There is an admission control layer which evaluates the feasibility of the request, given a specified worst case execution time (WCET) and taking into account the existing resources. The actual implementation details are beyond the scope of this paper. Nevertheless, there is a body of research on real-time operating systems and networks that has contributed to this subject [3, 13, 15].

Similarly to the duration measurement service, the timely execution service constitutes a building block for the TFD service. Therefore, instead of using the basic TCB interface function for timely execution, we will again use the TFD service interface.

5.3 Timing Failure Detection

There are two important aspects that we address in what follows. First, we explain how the TCB TFD service interface can be used to observe the duration and to detect timing

failures of read and write interactions on the replicated state machine. Second, we discuss the properties of the TFD service in order to show that they are sufficient to solve the contamination problem and enforce a correct behavior of the replicated state machine and of the system.

The TCB interface functions on which we focus in this paper are presented in Table 1. For the sake of simplicity, we only show the parameters that are relevant for our discussion.

Local Timing Failure Detection

```
id ← startLocal (start_ts, t_spec, handler)
end_ts,duration,faulty ← endLocal(id)
```

Remote Timing Failure Detection

```
id ← sendWRemoteTFD (start_ts, t_spec, handler)
id ← receive ()
end_ts ← endDistAction (id)
id,dur1,faulty1 ··· durn,faultyn ← waitInfo()
```

Table 1. TCB API to handle timing failures.

Read interactions can be observed through local timing failure detection functions. The function `startLocal()` is used to indicate that a new local action, starting at `start_ts` and with a specified maximum duration `t_spec`, should be observed by the TCB. `endLocal()` is called to indicate that the action has terminated. If the action does not terminate within the specified bound, then a `handler` function (if specified) will be executed by the TCB as soon as the timing failure is detected. On return, `endLocal()` provides the measured duration, as well as the end timestamp and a failure indicator. Each measurement has a unique `id`.

In the case of write interactions, the measurement can be accomplished using the `sendWRemoteTFD()` and the `endDistAction()` functions, which mark the start and end events, respectively. The identifier of the timed action, `id`, is obtained when the update message is received by the replica (through some `receive()` function) and is later used to indicate that the action has ended. To obtain the measured duration it is necessary to call `waitInfo()`. This function will block the calling process until the TCB is able to provide all the information relative to the completion of the timed action identified by `id`. This includes measured durations and failure status relative to all replicas addressed by the update operation (the timed action).

It is important to note that the above functions allow client processes to obtain information about the measured durations, which are required to correctly estimate the achievable coverage for the timed interactions. How this is done is another issue, which we discuss in detail in [7].

When requesting the TCB to observe a new timed action it is possible to specify a `handler` function to be timely executed as soon as a timing failure is detected. Note that there would be no guarantees about the timeliness of the reaction if it were done by the replica, in the payload part of the system. Deciding which local TCB modules should execute the failure handler depends on the particular application. Since in our case we want the handler to be executed in the replica that suffered the timing failure, we use the `sendWRemoteTFD()` function instead of a simple `send()` (as described in [24]), in which the handler would be executed by the client side TCB.

Given the above discussion about the relevant TCB interface functions, we now focus on the properties of the TFD service. We start by defining these properties, assuming that a timed action $TA(p, e, T_A, t_A)$ is the execution of some operation, such that its termination event e takes place at p, within time interval T_A from instant t_A:

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

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

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

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

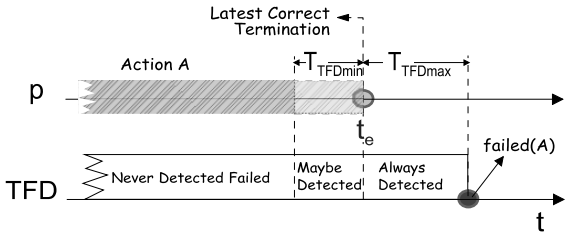

Figure 2. Execution of a perfect TFD.

The timing aspects of the failure detector are illustrated in Figure 2. When the execution of a timed action terminates before $t_e - T_{TFDmin}$ it is always considered correct. If it terminates between $t_e - T_{TFDmin}$ and t_e, it may either be considered timely or not. When it terminates after t_e, a

timing failure is always detected. The detection instant will occur no latter than $t_e + T_{TFDmax}$.

The detection latency is seen at the TCB interface as follows. It is guaranteed that the `handler` function will be executed no later than `start_ts` + `t_spec` + T_{TFDmax}. Furthermore, since timing failures are detected by every correct TCB, the failure handler can be timely executed, if needed, in each of them.

Next we show that this TFD service can be used in order to avoid the contamination effect. We first introduce an **error-cancellation** design rule, which must be followed in the design of the application.

5.4 Error Cancellation

Given bound $\mathcal{T}$ to be enforced with the help of the TCB:

1. *let the application logic work with the actual bound* $\mathcal{T}_{APP} = \mathcal{T}$ — this means that $\mathcal{T}$ corresponds to the timing specification that is used in the application, on which the correctness and temporal consistency depend
2. *set the TCB failure detection logic to trigger at* $\mathcal{T}_{TFD} = \mathcal{T} - T_{TFDmax}$ — this means that when using the TFD services, for instance through `sendWRemoteTFD()`, the value of `t_spec` must be set to $\mathcal{T} - T_{TFDmax}$
3. *design the environment in order to secure* $\mathcal{T}_{ENV} = \mathcal{T} - T_{TFDmax} - T_{TFDmin}$ — this means that to avoid timing fault detection, the environment should guarantee a bound equal to $\mathcal{T} - T_{TFDmax} - T_{TFDmin}$

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

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

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

Using this error-cancellation design rule in association with the services provided by the TCB, it is possible to design the replicated state machine in a safe way, by guaranteeing that no inconsistent replica may possibly contaminate the rest of the system. In Figure 3 we use the same scenario of Figure 1, but we add the TCBs of a client and a server node to illustrate how the contamination effect is avoided.

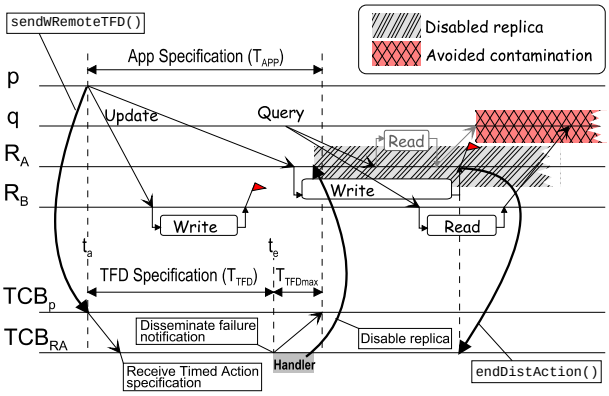

Figure 3. Avoiding the contamination effect.

When the update is initiated by process p, at t_a, the TCB of p is informed of a new timed action that must terminate within a certain T_{TFD} interval, to guarantee the timing specification T_{APP} (as per the error-cancellation rule). The TCB of p disseminates the T_{TFD} specification among all other TCBs. At the same time, replicas R_A and R_B receive the update request and process it. When R_B finishes the write operation it issues a `endDistAction()` (not shown in the figure), still before the deadline. The handler in R_B will not be executed. But replica R_A does not finish within the specified bound, yielding a timing failure at t_e. The timed completeness property guarantees that this failure will be detected by all TCBs within T_{TFDmax} of t_e, which means within the interval T_{APP} assumed by the application.

The guaranteed and timely execution of the handler is the last step required to timely disable the inconsistent replica R_A. Note that, without loss of generality, we can include the time necessary to execute the handler in T_{TFDmax}. The handler can do something as simple as changing the state of some variable in the server, which disables queries to be replied. When the query of process q is received by replica R_A, it will no longer be replied. Only one reply, from the correct replica R_B, will arrive at q.

Before we conclude this section, we must mention that in this paper we explicitly avoided to focus on implementation issues, or on the description of concrete protocols to be used in the TCB, since this is not required to understand the generic approach for timing fault tolerance that we introduced. Nevertheless, the interested reader can refer to a number of papers where these practical or concrete issues are dealt [6, 4, 5].

5.5 Reintegration of Failed Replicas

To prevent the possible contamination of the system, a late replica has to be removed from the set of available replicas. However, provided that the faulty write interaction finally terminates, the state of the affected replica will be consistent, at some point, with the state of the other replicas [1]. This requires disabled replicas to continue receiving and processing updates, until they become temporally consistent again.

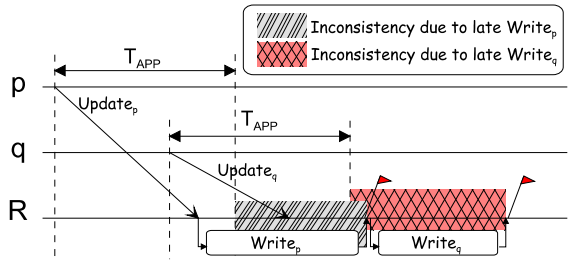

Figure 4. Generic replica management using the TFD service interface.

We argue that this reintegration procedure can be generalized with the availability of a timing failure detector. In fact, what triggers the passage from a temporally consistent to an inconsistent state is the detection of a timing failure. This is illustrated in Figure 4, where it is possible to observe the execution of two timed actions, corresponding to write interactions performed by two processes, p and q, on a replica R. In this example both actions incur in timing failures (detected before T_{APP}), which imply the existence of two periods of inconsistency until the actions finally terminate. Note that the second write operation ($Write_q$) is delayed by the first one, which produces a cascading inconsistency effect. What triggers the passage back to a consistent state is the indication of termination of the action (for instance, with the TCB `endDistAction()` TFD interface function).

6 Reviewing Related Work

There exist a number of works dealing with the several facets of timing fault tolerance. We make a clear distinction between the use of techniques to prevent or avoid timing failures, and approaches which let these failures happen and provide the means to detect and possibly tolerate them. The former have mostly to do with resource availability, scheduling, allocation and execution control strategies, which fall in the domain of synchronous system models. However, our interest is in the latter, which require alternative partially synchronous models to be considered. In the remainder of this section we essentially focus on the works that we are aware of, which follow this line of reasoning.

From a system model perspective the problem of timing fault tolerance requires timing failures to be explicitly considered in the fault model, so that proper solutions can be devised thereof. The fail-awareness concept [12] introduced in the framework of the Timed Asynchronous system model [10], is useful for the design of timed services that are at all times aware of their timeliness, and thus can switch to a safe or a degraded mode when a timing failure occurs. In this framework, the actual means to enforce this timely detection is by extending the model with a synchronous device, such as an hardware whatchdog, on which the applications can rely. Applying the proposed paradigm in the context of this model would raise some potentially difficult, or even unsolvable problems. In fact, and besides the problem of timely reaction upon failure detection, it is not obvious whether it would be possible to construct any useful timing failure detector, for the purpose of enforcing temporal consistency and coverage stability for the overall replica set. We consider this an open problem.

To the best of our knowledge, the first work that has explicitly focused on the problem of timing fault tolerance using a replication technique has been presented in [1]. This work proposes a specific solution in the context of the Quasi-Synchronous system model [22]. It is based on the existence of a TFD service highly integrated with a group management system. The infrastructure supporting the TFD and the group management system is assumed to be synchronous. The TFD service is designed to observe messages transmitted among group members and to detect timing failures occurring during the transmission of these messages. The group management system receives timely information from the TFD when it is necessary to remove a faulty replica from the group of active replicas. This integrated approach, although not being generic, presents some advantages over a generic solution. In fact, since the knowledge of the required communication semantics can be embedded in the TFD/group management protocols, it is possible to construct more efficient timed solutions.

In contrast with the work of Almeida, where the problem of decreased coverage is not explicitly addressed in the solution, the work presented in [16] is specifically concerned with achieving an adequate coverage for assumed timing bounds, using a replicated server. While the solution in [16] does not tackle the problem of handling server updates, the work now presented in [17] already considers that case, allowing consistency requirements to be specified when performing the operation. However, the solution still does not consider the possibility of specifying timeliness constraints for update operations. In this sense our work is more generic since we address the timeliness problem for both read and write interactions and we show how to maintain temporally consistent replicas despite timing failures.

7 Conclusions

This paper contributes with a paradigm for generic timing fault tolerance with a replicated state machine. Existing related work only deal with the problem in a partial way or using ad-hoc techniques. In contrast, we reason in generic terms and we identify the fundamental requirements that must be fulfilled by any complete solution to the problem.

We show how the TCB model can provide an adequate framework to deal with timing fault tolerance issues, and we provide a description of the interface functions that may be used to solve the problem of contamination and coverage stability. A particularly relevant aspect of the work we present here is the definition of timed completeness and accuracy properties, which are used to characterize in a generic way the TCB timing failure detector.

We believe that our work may constitute an important reference to other works in the area of timing fault tolerance, namely to works requiring the use of timing failure detectors.

References

[1] C. Almeida and P. Veríssimo. Using light-weight groups to handle timing failures in *quasi-synchronous* systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 430–439, Madrid, Spain, Dec. 1998.

[2] K. P. Birman. Replication and fault tolerance in the ISIS system. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 79–86, Dec. 1985.

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

[4] A. Casimiro, P. Martins, and P. Veríssimo. How to Build a Timely Computing Base using Real-Time Linux. In *Proc. of the 2000 IEEE Workshop on Factory Communication Systems*, pages 127–134, Porto, Portugal, Sept. 2000.

[5] A. Casimiro, P. Martins, P. Veríssimo, and L. Rodrigues. Measuring distributed durations with stable errors. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, pages 310–319, London, UK, Dec. 2001.

[6] A. Casimiro and P. Veríssimo. Timing failure detection with a Timely Computing Base. In *Proceedings of the European Research Seminar on Advances in Distributed Systems*, Madeira Island, Portugal, Apr. 1999.

[7] A. Casimiro and P. Veríssimo. Using the Timely Computing Base for dependable QoS adaptation. In *Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems*, pages 208–217, New Orleans, USA, Oct. 2001.

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

[9] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to Byzantine agreement. *Information and Computation*, 118(1):158–179, Apr. 1995.

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

[11] C. Fetzer and F. Cristian. A fail-aware datagram service. In *Proc. of the 2nd Workshop on Fault-Tolerant Parallel and Distributed Systems*, Geneva, Switzerland, Apr. 1997.

[12] C. Fetzer and F. Cristian. Fail-awareness: An approach to construct fail-safe applications. In *Digest of Papers, The 27th Annual International Symposium on Fault-Tolerant Computing*, pages 282–291, Seattle, Washington, USA, June 1997.

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

[14] H. Kopetz. *Real-Time Systems*. Kluwer Academic Publishers, 1997.

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

[16] S. Krishnamurthy, W. Sanders, and M. Cukier. A dynamic replica selection algorithm for tolerating time faults in a replicated service. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 107–116, Goteborg, Sweden, June 2001.

[17] S. Krishnamurthy, W. Sanders, and M. Cukier. An adaptive framework for tunable consistency and timeliness using replication. In *Proceedings of the International Conference on Dependable Systems and Networks*, Washington D.C., USA, June 2002.

[18] D. Powell. Failure mode assumptions and assumption coverage. In *Digest of Papers, The 22nd Annual International Symposium on Fault-Tolerant Computing*, pages 386–395, Boston, USA, July 1992.

[19] F. B. Schneider. The state machine approach: A tutorial. In *Fault–Tolerant Distributed Computing*, Lecture Notes in Computer Science, pages 18–41, 1987.

[20] F. Torres-Rojas, M. Ahamad, and M. Raynal. Timed consistency for shared distributed objects. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing*, pages 163–172, Atlanta, Georgia, USA, 1999.

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

[22] P. Veríssimo and C. Almeida. Quasi-synchronism: a step away from the traditional fault-tolerant real-time system models. *Bulletin of the TCOS*, 7(4):35–39, Winter 1995.

[23] P. Veríssimo and A. Casimiro. The Timely Computing Base model and architecture. *Transactions on Computers - Special Issue on Asynchronous Real-Time Systems*, 2002, To appear. A preliminary version of this document appeared as Technical Report DI/FCUL TR 99-2, Department of Computer Science, University of Lisboa, Apr 1999.

[24] P. Veríssimo, A. Casimiro, and C. Fetzer. The Timely Computing Base: Timely actions in the presence of uncertain timeliness. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 533–542, New York, USA, June 2000.

[25] P. Veríssimo and L. Rodrigues. *Distributed Systems for System Architects*. Kluwer Academic Publishers, 2001.