Using the Timely Computing Base for Dependable QoS Adaptation

António Casimiro
Paulo Veríssimo

DI–FCUL TR–01–3

July 2001

Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa
Campo Grande, 1749-016 Lisboa
Portugal

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

Using the Timely Computing Base for Dependable QoS Adaptation

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

Paulo Veríssimo
pjv@di.fc.ul.pt
FC/UL

Abstract

In open and heterogeneous environments, where an unpredictable number of applications compete for a limited amount of resources, executions can be affected by also unpredictable delays, which may not even be bounded. Since many of these applications have timeliness requirements, they can only be implemented if they are able to adapt to the existing conditions. Adaptation can be done by several ways, taking into account many different factors, but an obvious factor of success is knowing what they have to adapt to. In this paper we present a novel approach, called *Dependable QoS adaptation*, which can only be achieved if the environment is accurately and reliably observed.

Dependable QoS adaptation is based on the *Timely Computing Base (TCB)* model. The TCB model is a partial synchrony model that adequately characterizes environments of uncertain synchrony and allows, at the same time, the specification and verification of timeliness requirements. We introduce the *coverage stability* property and show that adaptive applications can use the TCB to dependably adapt and enjoy this property. We describe the characteristics and the interface of a *QoS coverage service* and discuss its implementation details.

1 Introduction

It is a striking reality that an increasing number of applications with timeliness or real-time requirements is being used in open, unpredictable or unreliable environments, like the internet. This is the case of multimedia applications, e-commerce or transaction based applications and applications for remote process control. For a systems' architect this may appear to be a contradiction, since it is well known that no real-time guarantees can be provided using intrinsically asynchronous platforms or environments. What happens in practice is that many of these applications are simply best-effort applications (no timeliness is expected), or are designed assuming a synchronous system model, postulating artificial bounds for relevant timing variables (such as those used in timeouts), and possibly implementing strategies to overcome the problems that might occur when those assumptions are violated. In the latter, it may appear that a guaranteed (synchronous) behavior is achievable but in fact timeliness requirements can only be partially fulfilled, only until the first occurrence of a *timing failure*.

Any systematic approach to the problem of implementing timeliness requirements in environments with unpredictable behavior has to take into account the effects of timing failures on the correctness of applications. However, this is not what happens with most of the known system models. For instance, in asynchronous, or time-free [17] systems, there is not even a notion of time and in synchronous systems the fault model includes crash and omission failures, but no timing

*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 Laboratoty (LASIGE) and projects Praxis/P/EEI/12160/1998 (MICRA) and Praxis/P/EEI/14187/1998 (DEAR-COTS).

failures. The timed asynchronous system model [12], on the contrary, allows the definition of timed services and "knows" that timing or performance failures can occur. But its basic asynchronous nature does not allow timing failures to be detected within known time bounds. So their effects can only be partially handled.

The lack of a generic model able to deal with the partial synchrony problem in a systematic way was one of the reasons that motivated our work around the definition of a new model, which we called the **Timely Computing Base (TCB)** model [27]. It assumes that systems, however asynchronous they may be, and whatever their scale, can rely on services provided by a special module, the TCB, which is timely, that is, synchronous. Under the TCB framework we define different classes of applications according to the properties that they enjoy, and we explain how to handle the effects of timing failures, with the help of TCB services, for each of these application classes (or combinations thereof).

In this paper we concentrate on the particular effect of *decreased coverage* resulting from timing failures, and show that there is a class of QoS adaptive applications, to which we refer as *time-elastic* applications, that may benefit from the TCB to avoid the decreased coverage problem. An important aspect of our work is that adaptation to environment changes relies on rigorous observations and mathematical analysis, which allows to dependably adapt the QoS (expressed as timing variables) to maintain the coverage of timeliness assumptions. We describe the QoS coverage service as an entity that can indeed be used by applications to dependably adapt.

The rest of this paper is organized as follows. Section 2 presents a brief overview of related work. Then, in section 3 the TCB model is presented. The problem of dependable adaptation is discussed in section 4, focusing on the role of the TCB in the global system architecture. Section 5 presents the QoS coverage service and in section 6 we discuss a few implementation issues. Finally, in section 7 we present our conclusions.

2 Related Work

The provision of quality of service (QoS) guarantees in open environments, such as the internet, is currently an active field of research. In fact, although there is a lot of work dealing with the problem of QoS provision in environments where resources are known and can be controlled [28, 24, 31, 18, 6], no systematic solution has been proposed for environments where there is no knowledge about the amount of available resources. In particular, and as far as we know, there is not a generic system model that adequately characterizes these kind of unpredictable environments, which can be used to derive solutions for applications with QoS needs. This paper presents the Timely Computing Base (TCB) model as a basic system model that can be used to design QoS adaptive architectures.

Most of the works that deal with QoS provision assume that resource reservation is possible. They are fundamentally concerned with resource management mechanisms and models, with QoS specification and mapping and with dynamic adaptation. For instance, [23] and [11] use benefit functions specified by the application as a way to optimize resource management. Several middleware architectures can be found. For instance, they deal with heterogeneous environments [24], they propose control-based solutions [19] or they use resource brokers to manage system resources [31]. A discussion about the main issues concerning QoS constrained communication can be found in [30] and a comprehensive survey about end-to-end QoS architectures can be found in [3].

We should also mention the IntServ [5] and DiffServ [4] architectures that were proposed to specifically address the problem of handling QoS requirements and differentiated service guarantees in the Internet. Detailed discussions about multimedia applications over the Internet can be found in [15] and [29].

Unlike the majority of the work dealing with QoS provision, we are concerned with environ-

ments where resource reservation and QoS enforcement may not be possible. Obviously, not all applications can be implemented on these environments. They need to be *adaptive* or more particularly *time-elastic*, that is, they must be able to adapt their timing expectations to the actual conditions of the environment, possibly sacrificing the quality of other (non time-related) parameters to avoid an increase of failures and a consequent coverage degradation. The success of an adaptive system has to do essentially with two factors: 1) the monitoring framework, which dictates the accuracy of the observations that drive adaptation and 2) the adaptation framework, which determines how the adaptation will be realized.

Monitoring of local resources and processes is widely used, but it does not provide a global view of the environment [21, 18]. Some works propose adaptation based on network monitoring and on information exchange among hosts (using specific protocols like RTP [7] or estimating delays [8]) but they do not reason in terms of the *confidence about the observations*, which is essential for dependable adaptation. In contrast, we focus on providing a global view of the environment (consistent to all participating entities) and on ensuring the correctness of that view.

Relatively to adaptation strategies, we mention the work in [1], that proposes adaptation among a fixed number of accepted QoS levels, and the work in [20], that uses control theory to enhance adaptation decisions and fuzzy logic to map adaptation values into application-specific control actions. Just like ours, both these works assume that the environment can be unpredictable and focus on providing middleware algorithms and mechanisms to improve the adaptation, instead of providing resource management architectures and services. However, since these works are not specially focused on *dependability* aspects, they have no dependability concerns relative to the adaptation, neither they reason in terms of a *generic model of partial synchrony* for the distributed system. Therefore, we believe our work can complement these other works.

The AQuA architecture was designed for dependable operation and provides adaptation mechanisms to respond to system faults in order to maintain certain (dependability related) QoS levels [13]. However, our dependability concerns are related with the adaptation mechanism itself, and not particularly with application dependability requirements.

The study of partial synchrony models has been the subject of some previous work. Chandra and Toueg have shown how to solve consensus in asynchronous systems with failure detectors [10]. Cristian and Fetzer have devised the timed-asynchronous model, where the system alternates between synchronous and asynchronous behavior, and where hardware clocks provide sufficient synchronism to make decisions such as 'detection of timing failures' or 'fail-safe shutdown' [12]. They have studied the problem of QoS adaptation, namely by proposing a methodology based on the timed-asynchronous model to design adaptive real-time applications [16]. The quasi-synchronous model, developed by one of the present authors, assumes that parts of the system have enough synchronism to perform 'real-time actions' with a certain probability [26]. These works were in general precursors of the TCB model [25], which provides a more generic framework to address asymmetries (both in space and time) of the system synchrony and is therefore adequate to handle a vast range of problems.

3 Timely Computing

This section provides an overview of the Timely Computing Base model. Its properties and engineering principles are firstly described, followed by the basic services essential for timely and dependable computing. Lastly, we discuss the most relevant issues related to the application programming interface. A complete and detailed description of these issues appears in [27] and [25].

3.1 The Timely Computing Base Model

A system with a Timely Computing Base (TCB) is divided into two well-defined parts: a *payload* and a *control* part. The generic or *payload* part prefigures what is normally 'the system' in homogeneous architectures. It exists over a payload network and is where applications run and communicate. In particular, all middleware services dedicated to QoS provisioning, monitoring or management are constructed in the payload part of the system. The *control* part is made of local TCB modules, interconnected by some form of medium, the *control* network. Figure 1 illustrates the architecture of a system with a TCB.

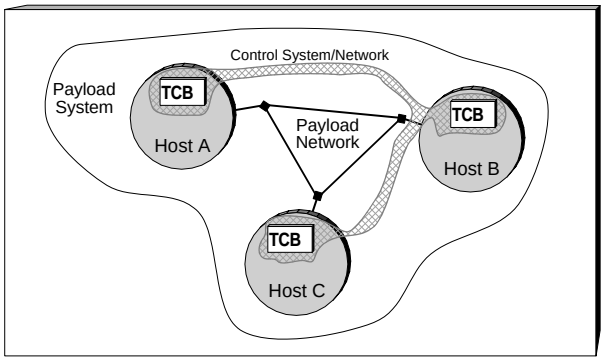

Figure 1: The TCB Architecture.

Concerning the payload part, the important property is that 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. 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.

In the control part, there is one local TCB at every node, fulfilling the following construction 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

An extended discussion of these principles and their impact in the implementation of a TCB can be found in [9].

TCB modules are assumed to be fail-silent, that is, they only fail by crashing. Moreover, it is assumed that the failure of a local TCB module implies the failure of that node. The TCB subsystem enjoys the following synchrony properties:

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

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

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

Property **Ps 1** refers to the determinism in the execution time of code elements by the TCB. Property **Ps 2** refers to the existence of a local clock in each TCB whose individual drift is bounded. This allows measuring local durations, that is, the interval between two local events. These clocks are internal to the TCB. Property **Ps 3** completes the synchronism properties, referring to the determinism in the time to exchange messages among TCB modules. It is assumed that inter-TCB channels provide reliable delivery, that is, no messages addressed to correct TCBs are lost. The set of all local TCB modules, interconnected by the control channel, constitutes the distributed TCB. Note that the interposition, shielding and validation principles must also be satisfied by the distributed TCB.

Given the above set of construction principles and properties, a TCB can be turned into an oracle providing time-related services to applications or middleware components. To accomplish this, a set of minimal services has to be defined, as well as a payload-to-TCB interface.

3.2 TCB Services

In order to keep the TCB simple, the services defined are only those essential to satisfy a wide range of applications with timeliness requirements: ability to measure distributed durations with bounded accuracy; complete and accurate detection of timing failures; ability to execute well-defined functions in bounded time. Table 1 presents an informal summary of these services.

Timely Execution
TCB 1 Timely Execution: *Given any function f with an execution time bounded by T and a delay D, for any execution of f triggered at real time t the TCB will not execute f within D from t and is able to execute f within T from t.*
Duration Measurement
TCB 2 *Given any two events occurring in any two nodes at instants t_s and t_e, the TCB is able to measure the duration between those two events with a known bounded error. The error depends on the measurement method.*
Timing Failure Detection
TCB 3 Timed Strong Completeness: *Any timing failure is detected by the distributed TCB within a known interval from its occurrence.*
TCB 4 Timed Strong Accuracy: *Any timely action finishing no later than some know interval before its deadline is never wrongly detected as a timing failure.*

Table 1: Basic services of the TCB.

Service **TCB 1** 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. **TCB 2** allows the measurement of arbitrary durations with a known bounded error. If no external time sources are available, the measurement error will be proportional to the distance between the events, by a factor that depends on the drift rate of local TCB clocks ($T_{D^2_{max}}$). Finally, **TCB 3** and **TCB 4** describe the properties that a *Perfect Timing Failure Detector (pTFD)* should exhibit. We use an adaptation of the terminology of Chandra [10] for the timed versions of the completeness and accuracy properties. Although the timing failure detector service is constructed upon the other ones, it is essential for any useful TCB.

3.3 Programming Interface

Beside defining essential services to be provided by the TCB, it is very important to provide a programming interface to allow potentially asynchronous applications to dialogue with a synchronous component. From a practical point of view, the interface should be simple to allow an easy use of TCB services.

A relevant aspect to understand what can be done, is that applications or middleware components can only be as timely as allowed by the synchronism of the payload system. The TCB, although being a synchronous component, does not make applications timelier, it only provides the means to detect how timely they are. However, since it can detect timing failures, it may execute timely contingency plans, such as timely fail-safe shutdown, which is very relevant for the implementation of *fail-safe applications*. Another important aspect is that application components on top of the TCB are autonomous entities that take advantage of TCB services by construction. They typically use it as a pacemaker, letting it assess (explicitly or implicitly) the correctness of past steps before proceeding to the next step. This is relevant in the context of *time-elastic applications* since adaptation measures can be taken at these intermediate points, with the help of the TCB, to maintain required QoS coverage levels.

When defining an interface between an asynchronous and a synchronous environment, one of the most important problems is that the latency of service invocation, as well as the latency of service replies, may not be bounded. So it is not possible to relate (in a time line) events occurring in one side with events occurring in the other. The interface summarized in Table 2 makes a bridge between a synchronous environment and a potentially asynchronous one. Some examples of how to use this interface can be found in [25].

Duration Measurement

```
timestamp ← getTimestamp ()
id ← startMeasurement (start_ts)
end_ts,duration ← stopMeasurement (id)
```

Timely Execution

```
end_ts ← exec (start_ts, wait, exec_dur, f)
```

Timing Failure Detection

```
id ← startLocal (start_ts, spec, handler)
end_ts,duration,faulty ← endLocal(id)
id ← send (send_ts, spec, handler)
id,deliv_ts ← receive ()
id,dur_1,faulty_1 ··· dur_n,faulty_n ← waitInfo()
```

Table 2: Summary of the API.

The most basic function is `getTimestamp`, which allows an application to get a timestamp. With this single function an application is able to obtain an upper bound on the time it has needed to execute a computation step. It would suffice to request a timestamp before the execution and another after it. If this execution is a timed action, then the knowledge of this upper bound is also sufficient do detect a timing failure, should it occur. The `startMeasurement` and `stopMeasurement` functions are provided to do this in a more explicit way. A duration is measured by calling `startMeasurement` with a timestamp (previously obtained) that marks the start event. An `id` identifies the on-going measurement. The measurement terminates by issuing `stopMeasurement`, which returns the measured duration.

The timely execution of critical functions is provided through the `startExec` function. This is not a general purpose function. On the contrary, it is intended to be used only for sporadic

actions with real-time requirements. When `startExec` is correctly used, the TCB will execute `func` accordingly to the parameters `start_ts`, `wait` and `exec_dur`. The first marks a reference point from where both the `wait` delay (deferral) and the maximum execution interval `exec_dur` should be counted. On return, a timestamp of the termination instant is provided through `end_ts`. It is obvious that not all `startExec` requests can be executed by the TCB, in which case an error status will be returned to the application. There must exist an admission control layer that performs the required admission tests before accepting any request. A more extensive discussion of this admission layer can be found in another paper [9].

The timing failure detection (TFD) service is presented to applications as a set of five functions. Two of them concern the detection of timing failures in local timed actions and the other three do the same for distributed timed actions. Note that failures in local actions are only important for the process performing them, while in the distributed case it is important to all those processes affected by the action.

To a certain extent, the `startLocal` and `endLocal` functions are similar to those of the duration measurement service. There are two new parameters: `spec` and `handler`. The former specifies the maximum execution duration and the later indicates an handler that is executed by the TCB, should a timing failure occur. As before, there is an `id` associated to each action under observation. With this interface an application can be constructed to timely react to timing failures: in fact, it is the TCB who executes the handler as soon as a timing failure is detected. Note that even if the failure could be timely signaled to the application, there would be no guarantees about the timeliness of the reaction if it was done in the payload part of the system. When the timed execution finishes, the application has to call `endLocal` in order to disable detection for this action and to receive the measured duration (`duration`) and the timeliness status (`faulty`).

A distributed execution requires at least one message to be sent between two processes. Thus, in addition to local delays, the TFD service has to observe the delay of message delivery. This is done by intercepting message transmissions, in a very simple and intuitive manner, through the provision of `send` and `receive` functions. Note that for brevity reasons the function prototypes presented in Table 2 omit normal parameters such as addresses, message buffers, etc. The meaning of the `send` function parameters is identical to the ones of the `startLocal` function. We assume it is possible to multicast a message to a set of destination processes using this `send` function. A distributed duration is bounded by the `send_ts` and by a receive event generated within the TCB of a destination node. This means that each receiver will measure its own duration. All the observed durations for some message can be known by means of the `waitInfo` function. A process issuing this function will remain blocked until the TCB sends the information concerning some message. Although it would be possible to explicitly wait for the information concerning a specific message identified by `id` (as presented in [25]), this interface is more versatile and is therefore adopted.

Finally, note that the distributed duration measurement service is implicitly provided by the distributed TFD service.

4 Dependable QoS Adaptation

Given the system model described in the previous section, and the set of services and interfaces we consider essential to address timeliness issues in unpredictable environments, we analyze in this section the implications of using the TCB model to construct QoS architectures.

Quality of Service can have different meanings that depend essentially on the application. This is why the definition of a completely generic model for specifying QoS needs is perhaps an impossible task: it is usually necessary to use mapping mechanisms to translate user level QoS requirements into system level ones. In the present work we assume that it is possible to define mapping mechanisms from application to system level, so that QoS requirements of different applications

can be specified in terms of timing variables, which are the variables of interest in the TCB model.

The TCB model provides a generic framework to deal with synchronism problems and to provide certain safety and liveness properties in the time domain to applications. In this sense, there is a potential for this model to be used as a base model for the development of some application classes. Adaptable applications with QoS requirements is one of those classes.

Typically, QoS adaptable applications are realized on top of a QoS framework. This framework is defined by a compound of QoS mechanisms which altogether characterize its ability to deal with application QoS requirements. There are three basic categories of QoS mechanisms: QoS provision, QoS control and QoS management mechanisms. If we want to use the TCB as a basic model to serve the design of some end-to-end QoS architecture, we must investigate the benefits that it might bring for the implementation of these QoS mechanisms. In particular, and since the TCB model provides a set of services to applications, the questions we have to find answers for are the following:

- Which QoS mechanisms can be improved by using TCB services?
- How can TCB services be used to improve QoS mechanisms?

The rest of the section will essentially focus on the discussion of these questions.

4.1 QoS Mechanisms under the TCB Framework

Before all, recall that the TCB does not restrict the properties of the payload part of the system, which allows for any QoS architecture to be implemented in a TCB based system. In fact, we stress that a TCB is just a small component that, mostly because of its small size and simplicity, can be constructed with more synchronous properties (or can be more dependably synchronous) than the rest of the system. Therefore, simply imagine that a TCB can be plugged into any existing system and be used as an oracle that provides a few basic services. The question is whether these services can be used **to improve** the existing QoS mechanisms. Let us first take a brief look on these mechanisms, closely following the systematization presented in [3])

The first category of QoS mechanisms, QoS provision, allows applications to use low level services, such as communication services, and to specify desired levels of QoS. QoS provision may require several things to be done, including: a) QoS mapping – to translate application level notation to system level specifications; b) Admission testing – to verify that there exist enough available resources to admit the requested QoS level; c) End-to-end reservation – to reserve resources and prevent resource outage during execution.

The second category consists of QoS control mechanisms, which serve to regulate and control the way in which resources are used and shared by the several flows (traffic or execution flows) during run-time in order to satisfy the specified QoS needs. They include several mechanisms, namely a) Flow shapping – to regulate usage according to some formal representation (for instance, a statistical one); b) Flow scheduling – to manage the several flows in an integrated way; c) Flow policing – to verify that the contract is being adhered to by applications; d) Flow control – to guarantee resource availability, either using open loop schemes (with advanced resource reservation) or closed loop schemes (with flow adaptation based on feed-back information); e) Flow synchronization – to account for precise interactions (e.g. multimedia interactions) of different flows.

Finally, the QoS management category includes the following mechanisms: a) QoS monitoring – to observe QoS levels achieved by lower layers; b) QoS maintenance – to perform fine adaptation of resource usage to maintain QoS levels; c) QoS degradation – to deliver QoS indication to applications when QoS cannot be sustained; d) QoS availability – to allow applications to specify bounds for QoS parameters; e) QoS scalability – which comprises QoS filtering, to allow the manipulation

of (traffic) flows, and QoS adaptation, to scale flows at the end system in order to respond to fluctuations in end-to-end QoS.

Now we must recall that TCB services can be used, in a general sense, to help applications observe their timeliness (using the duration measurement service), execute short real-time operations (with the timely execution service) or timely detect timing failures and react upon their detection within given time bounds. We must therefore look for QoS mechanisms that rely directly or indirectly on time, which are the ones that might be improved using these services.

Of the QoS provision mechanisms, QoS mapping and end-to-end reservation only perform logical operations and do not need timeliness informations to achieve their goals. Clearly, they are not eligible mechanisms for possible improvements. On the other hand, admission testing mechanisms need to compare available resources with requested ones, and so may possibly use the TCB to have the knowledge of available resources, measured in terms of timeliness. Relatively to QoS control mechanisms, the one that most clearly can use a TCB, in particular its duration measurement service, is the closed loop flow control mechanism. The basic idea is to use the information about distributed durations (relative to end-to-end transmission of data) to derive conclusions about the correct control actions to take. Finally, the QoS management mechanisms that deal with observing the behavior of the environment (or lower level services) can also use information about distributed durations to measure the QoS level (in terms of timeliness) that is being provided in a given moment.

The fundamental conclusion is that a TCB can essentially be used to improve the mechanisms that need information about the environment. This is clearly the case of mechanisms such as the flow control or the QoS monitoring, but can also be the case of other mechanisms, such as the QoS maintenance, which by using the output of a possibly improved monitoring can also employ finer grain techniques to achieve better results.

4.2 Improving QoS Mechanisms

We have just seen that a TCB can eventually be useful to construct or improve any QoS mechanism that needs information about the environment. Since this information is crucial, among other things, to take correct adaptation decisions, we argue that by using a framework which explicitly addresses the unpredictable nature of the environment in terms of timeliness, it is possible to adapt applications in a dependable manner, based on the timely observation of the environment.

To demonstrate our point of view, and having in mind that timeliness is the fundamental property in this context, we take a constructive approach that consists in analyzing why systems fail in the presence of uncertain timeliness, and deriving sufficient conditions to solve the problems encountered, based on the behavior of applications and on the properties of the TCB.

4.2.1 Effects of Timing Failures

We assume that an application is a computation in general and that any application is defined by a set of safety and timeliness properties $\mathcal{P}_A$. We remind the reader that we consider a model where components only do late timing failures. In the absence of timing failures, the system executes correctly. When timing failures occur, it is generally assumed that the effect is **unexpected delay**. However, we observe two additional effects, which produce different pathologies in the system, by the way they affect high-level system properties: **decreased coverage** and **contamination**. We define and discuss below the effects that are relevant for the current work. The effect of contamination is discussed in detail in [27].

Unexpected Delay

The immediate effect of timing failures is unexpected **delay**, defined as the violation of a timeliness property. That can sometimes be accepted, if applications are prepared to work correctly under increased delay expectations (e.g. mission-critical or soft real-time systems).

Uncoverage

When we make assumptions about the prevention of timing failures, we have in mind a certain coverage, which is the correspondence between system timeliness assumptions and what the environment can guarantee. We define **assumed coverage** $P_{\mathcal{P}}$ of a property $\mathcal{P}$ as the assumed probability of the property holding over an interval of reference. This coverage is necessarily very high for hard real-time systems, and maybe somewhat relaxed for any other real-time system, like a mission-critical or soft real-time system. However, in an environment with uncertain timeliness, the actual coverage varies during system life. Each time the environment conditions degrade to states worse than assumed, the coverage incrementally decreases, that is, the probability of timing failure increases. On the other hand, if the coverage is better than assumed, we are not taking full advantage from what the environment offers. This is a generic problem for any class of system relying on the assumption/coverage binomial[22]: if coverage of failure mode assumptions does not stay stable, a fault-tolerant design based on those assumptions will be impaired, because dependability properties, such as reliability, may be violated, or one does not take full advantage of the system. A sufficient condition for that not to happen, expressed by the *Coverage Stability* property, consists in ensuring that coverage stays close to the assumed value, over an interval of mission.

In [27] we have already presented formal proofs to show that the TCB can help applications to secure coverage stability despite the uncertainty of the environment. However, since in this paper we concentrate on more practical aspects, we discuss the concrete applicability of the coverage stability property.

4.2.2 Enforcing Coverage Stability

The definition of coverage stability for an application (instead of for a single property) simply consists in the expression of coverage stability for all timeliness properties defined within the application. But not all applications can benefit from this property.

A class that can indeed benefit is what we define as the **Time-Elastic Class ($\mathcal{TE}$)** [27]. In practical terms, $\mathcal{TE}$ applications are those whose bounds can be increased or decreased dynamically, such as QoS-driven applications. As already mentioned, provided that correct mappings are used to express application specific QoS requirements as timeliness requirements, it is possible to include these applications in the $\mathcal{TE}$ class and possibly enforce their coverage stability.

Coverage stability is a useful property in the presence of uncertain timeliness. It means that an application has the guarantee that its timeliness properties will hold during execution with a constant probability. The trade-off is that they must be time-elastic, which means that they must allow run-time adaptation of their timing parameters.

To explain how can a $\mathcal{TE}$ application achieve coverage stability under certain conditions, we must reason in terms of the services provided by the TCB and of what can be done with them. In this case, the relevant service is the duration measurement service (**TCB** 2) that may be used provide the necessary information to construct histograms of the distribution of durations and thus gathering evidence about coverage.

Note that a timeliness property implies that a given action has always to be performed within certain time bounds. Note also that the TCB is capable of observing those executions and measure their duration. Therefore, provided that there is a minimum number n_0 of *observed durations*, T,

representing the *specified duration bound*, $\mathcal{T}$ (derived from the timeliness property), it is possible to build a discrete probability distribution function *pdf* that represents the actual distribution of the timing variable with an error p_{dev0}[14]. This error depends on the measurement error introduced by the TCB and on the error introduced by the method used to build the *pdf*. With the *pdf* one can determine the probability $P = pdf(T)$, such that being p the actual probability of T, $|p - P| \leq p_{dev0}$

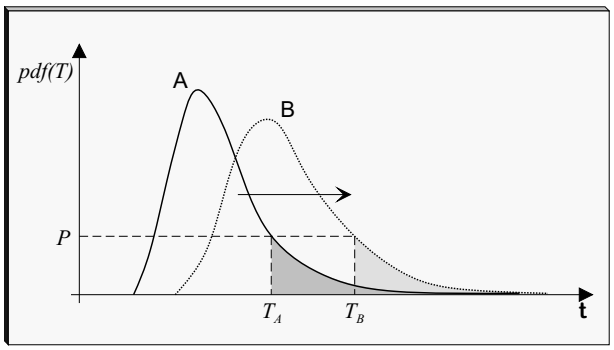

Figure 2: Example variation of distribution $pdf(\mathcal{T})$ with the changing environment

In systems of uncertain timeliness, the *pdf* of a duration varies with time. Therefore, if two distinct intervals of observation are considered, what can be observed is a shift of the baseline *pdf*, as depicted in Figure 2, by curves A and B. Note that at this point we are not assuming any particular distribution, and so the curves depicted in the figure merely represent any two possible *pdf*s. Although it is enough to know that p_{dev} remains bounded, one wishes to keep the error p_{dev} small in order to predict the probability of any T accurately, even with periods of timeliness instability. To achieve this goal it is necessary to periodically rebuild the *pdf* that represents the actual distribution. One can take the previous *pdf*, $pdf_{i-1}(\mathcal{T})$, or at least a part of that history (the most recent subset of values used to compute $pdf_{i-1}(\mathcal{T})$), and the immediately subsequent n observed durations, and compute a new $pdf_i(\mathcal{T})$ that reflects more accurately the current system state and forces p_{dev} to remain bounded and small.

It may not be practical to recompute a new *pdf* after every observed duration ($n = 1$), since this may require too many computational resources without considerable benefits. However, if the value of n is set too high and if the environment presents large variations, the accuracy of $pdf(\mathcal{T})$ can degrade and the error p_{dev} will have to be higher. The adequate value of n, as well as the history size, depends in fact on the behavior of the environment, and cannot be assigned an optimal value by default. From a practical point of view it is possible to make additional assumptions about the behavior of the environment (see section 6), or else it is necessary to implement additional mechanisms to detect (or even learn) particular behaviors and adjust the critical values accordingly to them.

Having described the general principle that allows a *pdf* to be constructed with the help of a TCB service, we still have to discuss how can this *pdf* be used to enforce the coverage stability property of a Time-Elastic application.

The basic idea is simple and relies on the assumption that QoS requirements may be specified in terms of $\langle bound, coverage \rangle$ pairs. This means that application are constructed assuming that each time bound holds with a certain coverage. When the environment degrades, the only way in which it is possible to maintain the coverage is to assume a different bound. This is where the availability of a *pdf* becomes useful, namely for Time-Elastic applications which are able to change the bounds dynamically. It allows to directly determine the new bound that has to be used in order to maintain the coverage. In the example of figure 2, when the environment changes and the new *pdf* B is obtained, the application maintains the degree of coverage by adapting the bound from T_A

to T_B. Note that these remarks are also true when the environment gets faster: the bound should get back to its lower value as soon as possible.

A pertinent question that could be made at this point is about the entity responsible for building the *pdf*: it could be the TCB itself, or the application, or a middleware layer specifically designed for that purpose. We introduce the QoS coverage service as the logic entity in the system responsible for handling the coverage related issues. This service is presented in the next section.

5 The QoS Coverage Service

In a general sense, the QoS coverage service can be described as providing to applications the ability of dependably decide how to adapt time bounds in order to maintain a constant coverage level. Figure 3 illustrates the overall aspect of a system with QoS oriented services and a TCB extended with a QoS coverage service.

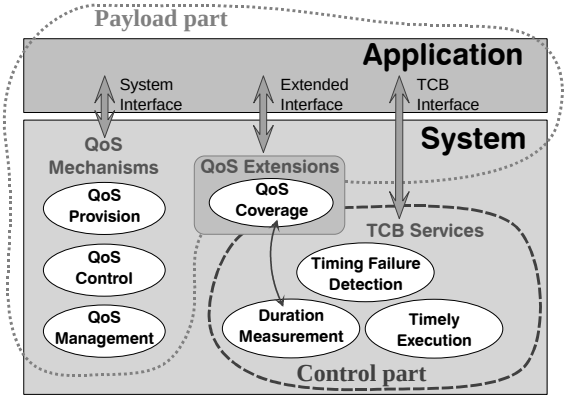

Figure 3: QoS extensions for the TCB.

Applications are layered on top of the system, which provides a set of services through dedicated interfaces. We distinguish three interfaces: the basic system interface, which provides access to all system services, including those related with QoS provision, control and management; the TCB interface that we described in section 3.3; and the QoS coverage service interface, which is presented below. The payload part of the system includes everything except the TCB, which constitutes the control part of the system. As we will see the QoS coverage service can reside in any of these two parts.

5.1 Service Interface

We have designed an interface for the QoS coverage service that being as simple as possible yet allows the fundamental objective to be achieved. Obviously, since this objective consists in maintain the coverage of a given timeliness property, the service has to know, at least, the *bound* that must be observed and its respective desired *coverage*. In the interface functions presented in Table 3 these parameters are represented, respectively, by `bound` and `cov`. Unchanged API portions are printed in gray.

In the framework that we have defined so far, it is possible to identify two kinds of bounds. In fact, the nature of the bounds associated to actions executed locally is different from the bounds that are imposed to actions executed among distributed nodes. For instance, this difference is quite explicit in the way durations are measured by the TCB. Because of that we have explicitly defined

QoS Coverage

```
id ← monDistr (bnd, cov, dev, c_id, hdlr)
id ← monLocal (bnd, cov, dev, f_id, hdlr)
new_bound ← waitChange (id)
```

Timing Failure Detection (modified functions)

```
id ← send (c_id, send_ts, spec, handler)
id,deliv_ts ← receive (c_id)
id,dur_1,faulty_1 ··· dur_n,faulty_n ← waitInfo (c_id)
```

Duration Measurement (modified functions)

```
id ← startMeasurement (start_ts, f_id)
```

Table 3: Extended and modified API.

two different functions to request a QoS coverage service: `monDistr` is used to monitor the coverage of distributed durations and `monLocal` is used for local durations.

A distributed duration always includes the delay for transmitting a message. Therefore, all observable distributed durations will result from messages sent through some communication channel using the modified `send` function presented in above table. The identifier of the channel, `c_id`, is necessary for the coverage service to associate a given ⟨*bound,coverage*⟩ pair to messages transmitted through that channel. This `c_id` parameter was omitted on purpose in Table 2, but is now presented for completeness reasons. Other parameters are still omitted.

Similarly to distributed durations, which are associated to communication channels, local durations are associated to the execution of specific functions or parts of the code. Therefore, when requesting the QoS coverage service to monitor the coverage of a local action it is necessary to provide some identification of that action. This is done through the `f_id` identifier. Naturally, this identifier has also to be used when requesting the measurement of durations, so that these measurements can be associated to specific actions (and their respective time bounds).

For the QoS coverage service to work properly it is necessary that it knows when to inform the application that a bound needs to be adapted. It would be possible to let the service provide information whenever a new *pdf* was built, independently of the variation degree. But in the case of applications that define QoS levels and are able to adapt among these levels, this would cause unnecessary performance degradation because for small variations nothing would be done (yet an indication would be delivered to the application). To prevent this situation, both interface functions contain the `dev` parameter. It is used to indicate the deviation relative to the current bound that triggers the delivery of a QoS change indication to the application. With this parameter it is possible to configure the hysteresis of the triggering of the indications.

The last parameter, `hdlr`, can be used to provide a handler for a function that should be executed whenever a QoS change is detected. The utility of providing an handler is highly determined by the timeliness properties that are enjoyed by the QoS coverage service, but this is discussed below. On return, the `monDistr` and `monLocal` functions provide an identifier, `id`, which is associated to the observed duration.

Note that independently of whether an handler is provided or not, the service always sends an indication to the application. In this interface we assume that the application is responsible for consuming and processing all indications sent by the QoS coverage service, using for that the `waitChange` function call. This function returns the most recent information concerning the duration identified by `id`, that is, it returns the bound that should be used in order to keep the desired coverage.

5.2 Service Operation

The service collects information relative to durations, builds the *pdf*, determines if it is necessary to inform the application of any considerable change and, finally, executes the appropriate actions.

Gathering information about distributed durations is done using the `waitInfo` function. Durations of local actions are explicitly observed using the modified duration measurement API function. The service keeps track of a certain amount of observed durations for each channel (`c_id`) and function (`f_id`). How the *pdf* is actually built depends on the particular implementation and on some assumptions about the environment. Therefore, this is discussed in section 6. Having some representation of the *pdf*, the current (and last reported) bound and the maximum allowed deviation, the service can easily determine if the deviation has been exceeded whenever a new *pdf* is built. If so, the new bound is recorded and reported to the application. If an handler has been specified it will be executed in the context of the QoS coverage service (whichever context this is). Whether the handler is timely executed or not depends on the timeliness properties of the service.

5.3 Timeliness Issues

The QoS coverage service has been so far defined as a service laying between the application and the TCB. But the consequences of implementing it in the *control* part of the system (inside the TCB) or in the *payload* part are quite different and relevant.

If the QoS coverage service is implemented as an additional TCB service, then it will obviously benefit from the fact that the TCB is a synchronous component. Every operation will be executed within known time bounds, in particular all those that conduct to the detection of QoS changes. This means that QoS change handlers will be timely executed allowing timely reactions to QoS changes. We say that a service with these characteristics allows for **real-time, dependable adaptation**. The trade-off for having this real-time behavior is that more complexity is being added to the TCB, increasing the probability of timeliness violations within the TCB itself, and making the TCB a "less synchronous" component than it was before. To decide whether it is worth implementing the QoS coverage service inside the TCB depends on the application and on the relative benefits that it could obtain by timely adapting to QoS changes.

If implemented in the payload part of the system, the QoS coverage service will behave according to the properties of the payload. This means that nothing can be assumed with respect to its timeliness. However, since the information that is used to build the *pdf* is still obtained using TCB services, it is possible to have certain guarantees about the (logical) correctness of the results obtained using this information. For instance, it is possible to ensure that given an interval of observation only the information relative to this interval is used to construct the correspondent *pdf*. It is also possible to ensure that all components of a distributed application have the same view of the environment. This is because the TCB delivers exactly the same information about distributed durations to all participants to which this information is relevant. The result is that applications can dependably (but possibly not timely) make decisions based on that information. The service allows for **dependable adaptation** in response to changing conditions of the environment.

We have seen that it is possible to specify QoS adaptation handlers when issuing requests to the QoS coverage service. However, the kind and complexity of the operations that can be done by these handler functions depends on the location of the QoS coverage service. In fact, if the service is inside the TCB, it will only allow simple operations to be executed in order to preserve the timeliness of the TCB. The TCB admission control layer (see section 3.3) will verify the feasibility of the request. On the other hand, if the request is accepted the handler will execute in the context of the TCB, benefiting from its synchrony properties. The practical implications of having two different execution contexts, the application and the TCB contexts, depend on the concrete implementation. It is possible to employ specific mechanisms to share parts of these contexts and

still preserve the temporal integrity of the TCB.

5.4 Extending the Interface

As already stated, we followed the principle of simplicity when proposing the interface for the QoS coverage service. Now we describe an important extension that can be done in order to make the service more flexible. So that the extension can be better understood, we recall the reader that the QoS coverage service was devised with the main objective of keeping the coverage close to the assumed value, with obvious repercussions in the proposed interface. We also recall that we have considered a particular class of applications, the $\mathcal{T}\epsilon$ class, containing applications with the ability to dynamically adapt their time bounds. However, we could as well have considered another class of applications, the coverage-elastic class ($\mathcal{C}\epsilon$), which includes applications that are able to withstand with variations of the assumed coverage values, keeping constant time bounds. It is clear that these applications cannot enjoy the coverage stability property. Instead, with the help of an extended QoS coverage service they can possibly enjoy a *Coverage Awareness* property, that is, the ability to know at a given moment the coverage value associated to some bound. Note that they still have to be constructed assuming that each time bound holds with a certain coverage, which means that QoS is still specified in terms of $\langle$*bound,coverage*$\rangle$ pairs. With coverage awareness these applications can employ specific adaptation procedures to handle coverage variations. For instance, they can take actions as simple as reporting the fact to the user or as complex as launching new application replicas in response to coverage degradation periods.

To accommodate this coverage-elastic class of applications, in addition to the time-elastic class, the QoS coverage service interface has to be slightly extended. The idea is to have a service that operates in one of two modes: with constant bounds or with constant coverage values. Therefore, the interface must allow for this operation mode to be specified and must be able to interpret the deviation parameter `dev` accordingly. On the other hand, depending on the selected mode, the `waitChange` function must be able to return either a new bound or a new coverage value.

6 Implementation Issues

The implementation of the QoS coverage service encompasses several issues, which can be discussed individually according to their specific functionalities. In this section we discuss some of these issues, in particular those related with the construction of the *pdf*, which we believe to be more relevant for the reader.

From a practical point of view, to implement a QoS coverage service it is necessary to decide which concrete algorithms and values will be used. For instance, since a *pdf* is built using a certain number of observed durations it is necessary to decide which number will this be.

We propose a method to build *pdf*s and detect QoS changes that relies on the following assumptions:

Probabilistic behavior – We are observing the duration of actions executed in the payload part of the system. Generically, the payload part of the system can be of any synchronism, which means that it might by synchronous, but also completely asynchronous. Consequently, there is nothing that formally limits the time it takes for actions to be executed. However, we know that when the environment is stable the execution time of specific actions usually follows some probabilistic distribution. Therefore, arbitrary behaviors can be disregarded. Nevertheless, since the execution conditions may vary, a certain probabilistic behavior can be suddenly affected and may be transformed into a different probabilistic behavior. This is why we assume that durations can follow any *probabilistic distribution*, and that it is not necessary

to know which distribution this is.

Recognition abilities – The ideal situation would be to know the exact probability distribution associated to a certain duration. We know that this distribution varies and that it depends, among other factors, on the application behavior and on the background execution context. Therefore, it would eventually be possible to apply additional run-time mechanisms in order to *recognize* at a given moment the probabilistic distribution more closely suited to the observed durations. However, we assume that we do not have enough computational power to do that during the execution.

Regular execution – We have mentioned that coverage should remain stable over an interval of mission. The idea is to provide a service that is well dimensioned for those intervals. We assume that applications have a regular execution, which allows the clear identification of intervals of mission. We further assume that an interval of mission contains a sufficient number of observable actions, required to construct a *pdf*. Coverage assumptions cannot be guaranteed to hold for sporadic actions.

Since we assume that the probabilistic distribution of durations in unknown, we will show that it is possible to determine $\langle bound, coverage \rangle$ pairs for a duration D, using only the expected value $E(D)$ and the variance $V(D)$ relative to that duration. Then we will describe how to compute $E(D)$ and $V(D)$.

We use a known result of probability theory, the *One-sided Inequality* (for instance, see [2]), which states that for any random variable D with a finite expected value $E(D)$ and a finite variance $V(D)$ we can bound the probability that D is higher than some value t:

$$P(D > t) \leq \frac{V(D)}{V(D) + (t - E(D))^2}, \text{ for all } t > E(D).$$

This expression can be used to calculate an upper bound for the probability of a time bound t being violated. This also corresponds to a lower bound for the coverage of the assumed bound t. For an assumed minimum coverage C_{min}, one can find the time bound t that has to be used in order to guarantee C_{min}:

$$t = \frac{2E(D) + \sqrt{4E(D)^2 - 4(E(D)^2 + V(D) - \frac{V(D)}{1 - C_{min}})}}{2}$$

Obviously, if the objective is to keep constant bounds and simply be aware of the maximum possible coverage for a given bound (see Section 5.4), the expression to use has to be:

$$C_{min} = 1 - \frac{V(D)}{V(D) + (t - E(D))^2}$$

Estimating $E(D)$ and $V(D)$

The values of $E(D)$ and $V(D)$ are *estimated* using a finite number of observed durations. The idea is simply to have a set of measured durations and then obtain the **average** and the **variance** corresponding to that set. The size of the set should be determined by the typical interval of mission of the application which is using the service. Note that intuitively it may seem a better approach to use as many values as possible when estimating $E(D)$ and $V(D)$, even if they are outside the interval of mission. However, since we assume that the environment (and consequently the distribution) may not remain stable, by using values observed outside the interval of mission we may just be degrading the accuracy of the estimated distribution.

Method accuracy

The accuracy of the proposed method is influenced by three kinds of errors. In the first place, the values used to estimate $E(D)$ and $V(D)$ (provided by the duration measurement service) have an associated error. Although this error is not explicitly provided in the interface (the measured durations are upper bounds), it would be possible to have a duration measurement service returning $\langle measurement,error \rangle$ pairs instead of simply $\langle measurement\_upper\_bound \rangle$.

In the second place, there is the error associated with the estimation of $E(D)$ and $V(D)$. This error obviously depends on the number of values used to obtain the estimation, which, as we have seen, should not be arbitrarily high by including values observed outside a certain interval of mission.

Finally, when using the proposed expressions derived from the one-sided inequality, we are introducing an error that depends on the "real" distribution corresponding to the duration. Since we do not assume any particular distribution, the formulas provide pessimistic but *secure* bounds, that in any case can compromise the system safety. Note that by removing the second assumption presented above (recognition abilities), it would become possible to consider specific distributions for the probabilistic behavior of durations (e.g. exponential or normal) and avoid this last source of errors.

7 Conclusions

We have presented a new approach for QoS adaptation, developed in the context of the Timely Computing Base (TCB) model. We addressed the problem of providing an adequate framework and programming infrastructure to applications with QoS requirements (specifically expressed in terms of timeliness properties), running in unpredictable environments.

We introduced an innovative analysis of the effect of timing failures on application correctness. Besides the obvious effect of delay, we identified a long-term effect, of decreased coverage of assumptions, and an instantaneous effect, of contamination of other properties. Even when delays are allowed, any of these effects can lead to undesirable behavior of a system. We addressed from a practical perspective the effect of decreased coverage, and showed that there is a class of applications, the time-elastic class, which, with the help of the TCB, can dependably adapt and enjoy the property of coverage stability. By analysing the different existing QoS mechanisms we concluded that the mechanisms requiring information from the environment, and specially the QoS monitoring mechanism, can be improved by the availability of a TCB providing accurate duration measurement and timing failure detection services.

We presented the QoS coverage service and discussed its synchrony properties. In particular, we compared the properties of a TCB based and a payload based QoS coverage service. We also discussed possible modifications to the interface in order to address other classes of applications as, for instance, the coverage-elastic class. Finally, a few important issues relative to the concrete operation of the QoS coverage service have also been discussed.

As future work we intend to extend our RT-Linux TCB [9] with the ideas presented in this paper. The results of this implementation will be reported in another paper.

References

[1] Tarek F. Abdelzaher and Kang G. Shin. End-host architecture for QoS-adaptive communication. In *Proceedings of the 4th IEEE Real-Time Technology and Applications Symposium*, Denver, Colorado, USA, June 1998.

[2] A. O. Allen. *Probability, Statistics, and Queueing theory with Computer Science Applications.* Academic Press, 2nd edition, 1990.

[3] C. Aurrecoechea, A. T. Campbell, and L. Hauw. A survey of QoS architectures. *Multimedia Systems Journal - Special Issue on QoS Architecture*, 6(3):138–151, May 1998.

[4] Steven Blake, David Black, Mark Carlson, Elwyn Davies, Zheng Wang, and Walter Weiss. An architecture for differentiated services, December 1998.

[5] Robert Braden, David Clark, and Scott Shenker. Integrated services in the internet architecture: an overview, June 1994.

[6] Scott Brandt, Gary Nutt, Toby Berk, and James Mankovich. A dynamic quality of service middleware agent for mediating application resource usage. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 307–317, Madrid, Spain, December 1998. IEEE Computer Society Press.

[7] I. Busse, B. Deffner, and H. Schulzrinne. Dynamic QoS Control of Multimedia Applications based on RTP. *Computer Communications*, 19(1), January 1996.

[8] A. Campbell and G. Coulson. A QoS adaptive transport system: Design, implementation and experience. In *Proceedings of the Fourth ACM Multimedia Conference*, pages 117–128, New York, USA, November 1996.

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

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

[11] S. Chatterjee, J. Sydir, B. Sabata, and T. Lawrence. Modeling applications for adaptive qos-based resource management. In *Proceedings of the 2nd IEEE High Assurance Engineering Workshop*, Bethesda, Maryland, USA, August 1997.

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

[13] M. Cukier, J. Ren, C. Sabnis, D. Henke, J. Pistole, W. Sanders, D. Bakken, M. Berman, D. Karr, and R. Schantz. AQuA: An adaptive architecture that provides dependable distributed objects. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, West Lafayette, Indiana, USA, October 1998.

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

[15] Paul Ferguson and Geoff Huston. *Quality of Service: Delivering QoS on the internet and in corporate networks.* John Wiley & Sons Inc., 1998.

[16] Christof Fetzer and Flaviu Cristian. Using fail-awareness to design adaptive real-time applications. In *Proceedings of the IEEE National Aerospace and Electronics Conference*, Dayton, Ohio, USA, July 1997.

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

[18] Ian Foster, Volker Sander, and Alain Roy. A quality of service architecture that combines resource reservation and application adaptation. In *Proceedings of the Eighth International Workshop on Quality of Service*, pages 181–188, Westin William Penn, Pittsburgh, USA, June 2000.

[19] Baochun Li and Klara Nahrstedt. A control-based middleware framework for quality of service adaptations. *IEEE Journal of Selected Areas in Communications, Special Issue on Service Enabling Platforms*, 17(9):1632–1650, September 1999.

[20] Baochun Li, Dongyan Xu, Klara Nahrstedt, and Jane W. S. Liu. End-to-end qos support for adaptive applications over the internet. In *SPIE Proceedings on Internet Routing and Quality of Service*, volume 3529, pages 166–176, Boston, Massachusetts, USA, November 1998.

[21] Hanan Lutfiyya, Gary Molenkamp, Michael Katchabaw, and Michael Bauer. Issues in managing soft QoS requirements in distributed systems using a policy-based framework. In *Proceedings of the International Workshop, POLICY 2001*, LNCS 1995, pages 185–201, Bristol, UK, January 2001.

[22] David 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.

[23] B. Sabata, S. Chatterjee, and J. Sydir. Dynamic adaptation of video for transmission under resource constraints. In *Proceedings of the 17th IEEE International Conference on Image Processing (ICIP'98)*, Chicago, Illinois, USA, October 1998.

[24] F. Siqueira and V. Cahill. Quartz: A QoS architecture for open systems. In *Proceedings of IEEE International Conference on Distributed Computing Systems (ICDCS 2000)*, pages 197–204, Taipei, Taiwan, April 2000.

[25] 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. IEEE Computer Society Press.

[26] Paulo Veríssimo and Carlos 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.

[27] Paulo Veríssimo and António Casimiro. The Timely Computing Base. DI/FCUL TR 99–2, Department of Computer Science, University of Lisboa, April 1999. Short version appeared in the Digest of Fast Abstracts, The 29th IEEE Intl. Symposium on Fault-Tolerant Computing, Madison, USA, June 1999.

[28] Carsten Vogt, Lars C. Wolf, Ralf Guido Herrtwich, and Hartmut Wittig. Heirat – quality-of-service management for distributed multimedia systems. *Special Issue on QoS Systems of ACM Multimedia Systems Journal*, 6(3):152–166, May 1998.

[29] Xin Wang and Henning Schulzrinne. Comparison of adaptive internet multimedia applications. *IEICE Transactions*, E82-B(6):806–818, June 1999.

[30] Lars C. Wolf, Carsten Griwodz, and Ralf Steinmetz. Multimedia communication. *Invited Paper in Special Issue on Global Information Infrastructure*, 85(12):1915–1933, December 1997.

[31] Dongyan Xu, Duangdao Wichadakul, and Klara Nahrstedt. Multimedia service configuration and reservation in heterogeneous environments. In *Proceedings of International Conference on Distributed Computing Systems*, Taipei, Taiwan, April 2000.