The Implementation of a QoS-based Adaptive Distributed System Model for Fault Tolerance

Raimundo Macêdo[1], **Sérgio Gorender**[1], **Paulo Cunha**[2]

[1]Laboratório de Sistemas Distribuídos (LaSiD), Departamento de Ciência da Computação (DCC), Universidade Federal da Bahia - UFBA
Salvador, Bahia

[2]Centro de Informática, Universidade Federal de Pernambuco - UFPE
Recife, Pernambuco

`{macedo|gorender}@ufba.br` `prfc@ci.ufpe.br`

Abstract. *The capability of dynamically adapting to distinct run-time conditions is an important issue when designing distributed systems where negotiated quality of service (QoS) cannot always be delivered between processes. Providing fault-tolerance for such dynamic environments is a challenging task. Considering such a context, this paper proposes an implementation of an adaptive model for fault-tolerant distributed computing, composed of distributed failure and state detectors (for monitoring available QoS and process failures, respectively). Our implementation relies on a QoS mechanism, that we also defined and implemented, called the QoS provider (QoSP). Thanks to the modularity dimension of the QoSP, porting the model implementation to a given environment requires only to implement the QoSP functions. The developed system is able to react to the changes in the execution environment, dynamically adapting it to the QoS conditions available at runtime, and it is particularly powerful in the sense that it allows for processes with distinct QoS views to continue their computations and cooperate in a safe, fault tolerant manner.*
Keywords*: Adaptability, Asynchronous/synchronous distributed system, Consensus, Distributed computing model, Fault tolerance, Quality of service*

1. Introduction

Distributed systems are composed of processes, usually spread over a set of networked sites, which communicate via message passing to offer services to upper layer applications. As there are no share states accessible by processes (for instance, a shared memory or global clock), a major difficulty a system designer has to cope with in these systems is the capture of consistent global states (from which decisions can be taken in order to guarantee a safe progress of the upper layer applications). To study and investigate what (and how) can be done in these systems, two distributed computing models have received a lot of attention, namely, the *synchronous* model and the *asynchronous* model.

The synchronous distributed computing model provides processes with bounds on processing time and message transfer delay, and the asynchronous model is characterized by the absence of time bounds (that is why this model is sometimes called *time-free* model). Synchronous systems are attractive because they allow system designers to solve a lot of problems. The price that has to be paid is the a priori knowledge on time bounds.

If they are violated, the upper layer protocols can be unable to still guarantee their safety property. As they do not rely on explicit time bounds, asynchronous systems do not have this drawback. Unfortunately, they have another one, namely, some basic problems are impossible to solve in asynchronous systems. The most famous is the consensus problem, that has no deterministic solution when even a single process can crash [8]. The *consensus* problem is, however, a very important building block to implement fault tolerant distributed systems (processes must come to an agreement on an decision value despite of process failures). For instance, it can be used to implement *virtual synchrony* on group communication [12] and to solve the *atomic broadcast* problem [3]. This impossibility in the asynchronous model has motivated researchers to find distributed computing models, weaker than the synchronous model but stronger than the asynchronous model, in which consensus can be solved.

In practice, systems are neither fully synchronous, nor fully asynchronous. Most of the time they behave synchronously, but can have "unstable" periods during which they behave in an anarchic way. Moreover, there are now QoS architectures that allow processes to dynamically negotiate the quality of service of their communication channels [16, 18, 1]. These observations motivated us the design of a hybrid and adaptive distributed computing model (combining characteristics of both, synchronous and asynchrounous models), that does its best to provide the processes with safe information on the current state of the other processes [13].

Our model is time-free in the sense that processes are not provided with time bounds guaranteed by the lower system layer. Each process p_i is provided with three sets denoted $down_i$, $live_i$ and $uncertain_i$. These sets, that are always a partition of the whole set of processes, define the view p_i has of the state of the other processes. More precisely, if $p_k \in down_i$, then p_i knows that p_k has crashed; when $p_k \in live_i$, then p_i can consider p_k as being alive; finally, when $p_k \in uncertain_i$, p_i has no information on the current state of p_k. These sets can evolve in time, and can have different values for different processes. For example, it is possible that (due to the fact that some quality of service can no longer be ensured) the view p_i has on p_k be degraded in the sense that the model moves p_k from $live_i$ to $uncertain_i$. It is also possible that (due to the fact that a stable period lasts "long enough") the view p_i has on p_k be upgraded in the sense that the model moves p_k from $uncertain_i$ to $live_i$. So, the model is able to benefit from time windows to transform timeliness properties (currently satisfied by the lower layer) into time-free properties (expressed on the previous sets) which can be used by upper layer protocols. Interestingly, this model includes the synchronous model and the asynchronous model as particular cases. The synchronous model corresponds to the case where, for all the processes p_i, the sets $uncertain_i$ are always empty. The asynchronous model corresponds to the case where, for all the processes p_i, the sets $uncertain_i$ always include all processes.

It is important to notice that our approach is orthogonal to the failure detector (FD) approach [3]. It aims at benefiting from the fact that systems are built on QoS architectures, thereby allowing a process not to always consider the other processes as being in an "uncertain" state. As the proposed model includes the asynchronous model, it is still necessary to enrich it with appropriate mechanism to solve problems that are impossible to solve in pure time-free systems. To illustrate this point and evaluate the proposed approach, we developed an adaptive consensus protocol, which is based on the fact that each process p_i is provided with (1) three sets $down_i$, $live_i$ and $uncertain_i$ with the previously defined semantics, and (2) an appropriate failure detector module (namely, $\diamond\mathcal{S}$). While $\diamond\mathcal{S}$-based consensus protocols in asynchronous systems require $f < n/2$, the proposed $\diamond\mathcal{S}$-based protocol allows bypassing this bound when few processes are in the $uncertain_i$ sets during the protocol execution [13].

Contributions of this paper The main contribution of this paper is to show how a fault tolerant application can benefit from execution environments with QoS facilities to adjust its behaviour to dynamically modifying run-time conditions, without affecting its correct execution and improving its ability to detect process crashes when possible. To our best knowledge, no previous work explored QoS facilities in such context. In particular, we show how to implement the adaptive distributed computing model, above-described, and introduced by us in [13], on top of a system with typical QoS facilities. To accomplish this objective, we define a new QoS support mechanism to interface the fault tolerant applications, named the QoS Provider (QoSP), and introduce two adaptive mechanisms that assess existing QoS conditions and update process states, accordingly (the *adaptive state and failure detectors*). Additionally, the paper shows experimental data from a Java/Linux prototype.

Related work Fault tolerance for distributed systems has been addressed in several ways, according to the system and fault models assumed for the target environment. Usually, once defined a given system and fault model, the algorithms to handle fault tolerance are fixed for such model prior to execution. Although such a static configuration already provides a degree of adaptivity, in some situations it is desired that the fault tolerant mechanism be itself adaptive.

The *timed asynchronous* model [5] considers asynchronous processes equipped with physical clocks. The *timely computing base* [20] provides services built on top of an underlying dedicated synchronous network.

Closer to our approach there are AquA [17] and Ensemble [19]. AquA provides adaptive fault-tolerance to CORBA applications by replicating objects and providing a high-level method that an application can use to specify its desired level of reliability. AquA also provides an adaptive mechanism capable of coping with application dependability requirements at runtime. It does so by using a majority voting algorithm. Ensemble offers a fault-tolerant mechanism that allows adaptation at runtime. This is used to dynamically adapt the components of a group communication service to switch between two total order algorithms (sequencer-based vs token-based) according to the desired system overhead and end-to-end latency for application messages.

Several works aimed at circumventing the impossibility of consensus in asynchronous systems [8]. Minimal synchronism needed to solve consensus is addressed in [6]. Partial synchrony making consensus solvable is investigated in [7]. Finally, the failure detector approach and its application to the consensus problem have been introduced and investigated in [2, 3]. Other authors have also developed consensus protocols using the unreliable failure detectors of Chandra and Toueg, some of them with adaptivity properties. Mostefaoui and Raynal [15] presented a quorum-based adaptive consensus protocol, where the quorum depends on the failure detector class and it is statically determined for a whole execution of the consensus. So, their solution does not provide adaptation at runtime. Hurfin, Macêdo et al [11] proposed a consensus protocol with adaptability properties that allows for processes to dynamically change the number of concurrent rounds (from 1 to the maximum number of processes - n), and can assume distinct communication patterns (centralized or decentralized), providing, therefore, adaptation to the communication bandwidth and system resources available at runtime. The work presented in [10] described an adaptive model based on QoS architectures that works by dynamically searching for a spanning tree of timely channels, and once such spanning tree is found, the whole system is considered synchonous. In contrast, the present paper describes a different system model and, besides, it allows for a given process to exploit a synchronous behaviour even if there is only one timely channel connected to it (no spanning tree is required).

Differently from previous works, in our paper we consider an environment where the system behaviour can vary dynamically, leading to the possibility of using distinct QoS during one execution of a given distributed algorithm.

Structure of the paper This paper is made up of four sections. Section 2. overviews the model introduced in [13]. Section 3. describes an implementation of the model based on negotiated QoS guarantees, discusses its correctness, and presents performance data collected from experiments. Finally, Section 4. draws conclusions and indicates future works.

2. An Overview of the Adaptive Model for Fault-Tolerant Distributed Computing

This section overviews our distributed computing model presented in [13]. We consider a system consisting of a finite set Π of $n \geq 2$ processes, namely, $\Pi = \{p_1, p_2, \ldots, p_n\}$. A process executes steps (a step is the reception of a message, the sending of a message, each with the corresponding local state change, or a simple local state change). A process can fail by *crashing*, i.e., by prematurely halting. After it has crashed a process does not recover. It behaves correctly (i.e., according to its specification) until it (possibly) crashes. By definition, a process is *correct* in a run if it does not crash in that run. Otherwise, a process is *faulty* in the corresponding run. In the following, f denotes the maximum number of processes that can crash ($1 \leq f < n$). Until it possibly crashes, the speed of a process is positive but arbitrary.

Processes communicate and synchronize by sending and receiving messages through channels. Every pair of processes (p_i, p_j) is connected by two directed channels, denoted $p_i \rightarrow p_j$ and $p_j \rightarrow p_i$. Channels are assumed to be reliable: they do not create, alter or lose messages. In particular, if p_i sends a message to p_j, then eventually p_j receives that message unless it fails. There is no assumption about the relative speed of processes or message transfer delays (let us observe that channels are not required to be FIFO). The primitive *broadcast* MSG(v) is a shortcut for: $\forall p_j \in \Pi$ **do** *send* MSG(v) *to* p_j **end** (MSG is the message tag, v its value).

How a Process Sees the Other Processes: Three Sets per Process A crucial issue encountered in distributed systems is the way each process perceives the state of the other processes. To that end, the proposed model provides each process p_i with three sets denoted $down_i$, $live_i$ and $uncertain_i$. The only thing a process p_i can do with respect to these sets is to read the sets it is provided with; it cannot write them and has no access to the sets of the other processes.

These sets, that can evolve dynamically, are made up of process identities. Intuitively, the fact that a given process p_j belongs to $down_i$, $live_i$ or $uncertain_i$ provides p_i with some hint on the current status of p_j. More operationally, if $p_j \in down_i$, p_i can safely consider p_j as being crashed. If $p_j \notin down_i$, the state of p_j is not known by p_i with certainty: more precisely, if $p_j \in live_i$, p_i is given a hint that it can currently consider p_j as not crashed; when $p_j \in uncertain_i$, p_i has no information on the current state (crashed or live) of p_j.

At the abstraction level defining the computation model, these sets are defined by abstract properties (the way they are implemented is irrelevant at this level, it will be discussed in Section 3.). The specification of the sets $down_i$, $live_i$ and $uncertain_i$, $1 \leq i \leq n$, is the following (where $down_i(t)$ is the value of $down_i$ at time t, and similarly for $live_i(t)$ and $uncertain_i(t)$):

R0 Initial global consistency. Initially, the sets $live_i$ (resp., $down_i$ and $uncertain_i$) of all the processes pi are identical. Namely, $\forall i, j$: $live_i(t) = live_j(t)$, $down_i(t) = down_j(t)$, and $uncertain_i(t) = uncertain_j(t)$, for $t = 0$.

R1 Internal consistency. The sets of each p_i define a partition:
- $\forall i$: $\forall t$: $down_i(t) \cup live_i(t) \cup uncertain_i(t) = \Pi$.
- $\forall i$: $\forall t$: any two sets ($down_i(t)$, $live_i(t)$, $uncertain_i(t)$) have an empty intersection.

R2 Consistency of the $down_i$ sets.
- A $down_i$ set is never decreasing: $\forall i$: $\forall t$: $down_i(t) \subseteq down_i(t+1)$.
- A $down_i$ set is always safe with respect to crashes: $\forall i$: $\forall t$: $down_i(t) \subseteq F(t)$.

R3 Local transitions for a process p_i. While an upper layer protocol is running, the only changes a process p_i can observe are the moves of a process p_x from $live_i$ to $down_i$ or $uncertain_i$.

R4 Consistent global transitions. The sets $down_i$ and $uncertain_j$ of any pair of processes p_i and p_j evolve consistently. More precisely:
- $\forall i, j, k, t_0$: $(\ (p_k \in live_i(t_0)) \wedge (p_k \in down_i(t_0+1))\ ) \Rightarrow (\ \forall t_1 > t_0 : \ p_k \notin uncertain_j(t_1)\ )$.
- $\forall i, j, k, t_0$: $(\ (p_k \in live_i(t_0)) \wedge (p_k \in uncertain_i(t_0+1))\ ) \Rightarrow (\ \forall t_1 > t_0 : \ p_k \notin down_j(t_1)\ )$.

R5 Conditional crash detection. If a process p_j crashes and does not appear in the $uncertain_i$ set of any other process p_i for an indefinitely long period of time, it eventually appears in the $down_i$ set of each p_i. More precisely:
$\forall p_i$, if p_j crashes at time t_0, and there is a time $t_1 \geq t_0$ such that $\forall t_2 \geq t_1$ we have $p_j \notin uncertain_i(t_2)$, then there is a time $t_3 \geq t_2$ such that $\forall t_4 \geq t_3$ we have $p_j \in down_i(t_4)$.

As we can see from this specification, at any time t and for any pair of processes p_i and p_j, it is possible to have $live_i(t) \neq live_j(t)$ (and similarly for the other sets). Operationally, this means that distinct processes can have different views of the current state of each other process. Let us also observe that $down_i$ is the only safe information on the current state of the other processes that a process p_i has.

Finally, notice that R3 prevents applications from modifying a channel's QoS during the execution of consensus (in fact, we show that without this restriction consensus cannot be achieved in this model [13]). However, it is important to notice that nothing prevents to upgrade the model between consecutive instances of the consensus, by moving a process $p_x \in uncertain_i(t1)$ into $live_i(t2)$ or $down_i(t2)$. Such an upgrade of a $live_i$ or $down_i$ sets between two runs of an upper layer protocol do correspond to "synchronization" points during which the processes are allowed to renegotiate the quality of service of their channels.

Enriching the Model to Solve Consensus It is well known that the consensus problem cannot be solved in pure time-free asynchronous distributed systems [8]. So, we consider that the system is augmented with a failure detector of the class denoted $\diamond\mathcal{S}$ [3] (which has been shown to be the weakest class of failure detectors able to solve consensus despite asynchrony [2]).

In [13] we have presented a $\diamond\mathcal{S}$-based consensus protocol suitable for our adaptive distributed system model, that has a noteworthy feature on its generic dimension: the same protocol can easily be instantiated in fully synchronous systems or fully asynchronous systems. Of course, these instantiations have different requirements on the value of f. A significant characteristic of the protocol is to suit to distributed systems that are neither fully synchronous, nor fully asynchronous. The price that has to be paid consists then in equipping the system with a failure detector of the class $\diamond\mathcal{S}$. The benefit it brings lies in the fact the constraint on f can be weaker than $f < n/2$ ([1]).

3. Implementation of the Adaptive Distributed Computing Model

For applications like distributed real-time control and multimedia systems, it is essential that *quality-of-service* (QoS) be guaranteed system-wide, i.e., from the operating system to the network. Though such QoS infrastructures are not yet widely available (e.g., in the Internet), there is a growing number of systems and architectures being developed and used to fulfill the QoS requirements of these modern distributed applications. Our system model builds on the facilities typically encountered in such QoS architectures, which include mechanisms to specify, enforce and manage end-to-end QoS requirements for a variety of classes of applications.

In particular, we assume that the underlying system is capable of providing *timely* communication channels. That is, as long as a message sender remains operational and the QoS of the related channel is sustained during transmission, a sent message is always received within a bounded time limit, say Δ. Such a level of service, which is largely supported in existing architectures, is achieved and denoted in different ways for distinct QoS architectures (e.g., *deterministic* [18] and *Express Forward* [1]). Similarly, we also assume the existence of best-effort channels where messages are transmitted without guaranteed bounded time delays. We call these channels untimely. For both kinds of channels (*timely* and *untimely*), we assume that messages are neither lost nor corrupted, and can be delivered in any order. Another feature we assume for the underlying QoS system is the capability of informing the current QoS, timely or untimely, available for the created channels - bearing in mind that as problems can occur during the communication (such

[1]The only consensus protocols we are aware of, that work in distributed systems that are neither fully synchronous, nor fully asynchronous, are the protocols designed for fully asynchronous systems. These protocols require (1) $\diamond\mathcal{S}$ (or a failure detector that has the same power as far as failure detection is concerned, e.g., a leader oracle [2, 4]), and (2) the upper bound $f < n/2$ on the number of process crashes. Our protocol has the same requirement for item (1), but a weaker for item (2).

as congestion and faults), and also as an application can renegotiate the QoS for its channels, the QoS of a channel can be dynamically modified, changing between timely and untimely.

Thus, the QoS-based underlying distributed system we consider is a set of *n* processes $\Pi = p_1, \ldots, p_n$, located in one or more sites, communicating through a set Γ of $n(n-1)/2$ channels, where $c_{i/j}$ means a communication channel between p_i and p_j. (That is, the system is represented by a complete graph $DS(\Pi, \Gamma)$, where Π are the nodes and Γ the edges of the graph.)

We also assume that processes in Π are equipped with enough computational power so that the time necessary to process control messages are negligible small compared with network delays. Therefore, control messages originated by the implemented model are assumed to be promptly computed[2]. Moreover, the processes are assumed to fail only by crashing and the network is not partitionable.

3.1. The QoS Provider

In order to make our system model portable to distinct QoS architectures, we define a number of functions to be encapsulated in a mechanism we call the QoS Provider, for creating and assessing QoS communication channels on behalf of application processes. Thanks to this modular encapsulation, porting our system to a given QoS infrastructure means implementing the QoSP functions in such a new target environment. The QoSP is made up of a module in each site of the system. The basic data structure maintained by each module is a table holding information about existing channels. These modules exchange messages to carry out modifications on the QoS of channels (due to failures or application requests).

The QoS provider has characteristics similar to services present in QoS architectures such as Omega [16]. This section describes its main functionalities, which are needed for implementing our system model. (The complete description of the QoS provider is beyond the scope of this paper, and can be seen elsewhere [9].) Processes interact with the QoS Provider through the following functions.

- $CreateChannel(p_x, p_y) : \Pi^2 \rightarrow \Gamma$.
- $DefineQoS(p_x, p_y, qos) : \Pi^2 \times \{timely, untimely\} \rightarrow \{timely, untimely\}$.
- $QoS(p_x, p_y) : \Pi^2 \rightarrow \{timely, untimely\}$.
- $Delay(p_x, p_y) : \Pi^2 \rightarrow N^+$.

The functions $CreateChannel()$, $DefineQoS()$, $QoSl()$, and $Delay()$ are used for creating a channel, changing its QoS, obtaining its current QoS, and obtaining the expected delay -in miliseconds- for message transfer for the channel $c_{x/y}$, respectively. Besides the above functions, each QoSP module continuously monitors all timely channels linked to the related site, to check whether failures or lack of resources have resulted in a modification of the channel QoS (from timely to untimely). A particular case that the QoSP also assesses is the existence of a timely channel where no message flow happens within a period of time, which indicates that the timely channel is possibly linking two crashed processes (in this circumstance, the QoS of the channel is modified to untimely

[2]This assumption can be relaxed by using real-time operating systems, which can provide bounded process execution times

in order to release resources). Modifications on the QoS of channels are immediately reported to the state detector related to a given module of the QoSP, through messages *changeQoS(p_x,p_y,newQoS)*, indicating that the QoS of the channel $c_{x/y}$ has been modified to *newQoS* (see Section 3.2.1.).

When a process crashes, the QoS Provider can still give information about the QoS of the channels linked to that crashed process. However, if the site hosting a process crashes or the related QoS Provider crashes, all the channels allocated to processes in this site are destroyed. If a given QoS Provider module cannot deliver information about a given channel (possibly, because it has crashed), this channel is then assumed to be untimely (which may represent a change in its previous QoS condition).

3.2. The System Model Implementation

In our system model, distributed processes perceive each other's state by reading the contents of the sets *down*, *live*, and *uncertain*. These sets can evolve dynamically following system state changes while respecting the rules R0 to R5. Therefore, implementing our system model implies providing the necessary mechanisms to maintain the sets according to their semantics. Two mechanisms have been developed to this end:

- a *state detector* that is responsible for maintaining the sets *live* and *uncertain*, in accordance with the information delivered by the QoSP, and
- a *failure detector* that utilizes the information provided by both, the QoSP and the state detector, to detect crashes and update the *down* sets accordingly.

Associated with each process p_i there is a module of the state detector, a module of the failure detector, a representation of the $DS(\Pi, \Gamma)$ graph, and the three sets: $live_i$, $uncertain_i$, and $down_i$. The $DS(\Pi, \Gamma)$ graph is constructed by using the QoSP functions *CreateChannel()* and *DefineQoS()*, for creating channels according to the QoS required and resources available in the system. The modules of the state detector exchange the information of the created channels so that they keep identical $DS(\Pi, \Gamma)$ graphs during the system initialization phase.

During the initialization phase, the set $down_i$ is set to empty and the contents of $live_i$ and $uncertain_i$ are initialized so that the identity of a process p_j is placed into $live_i$ if and only if there is a timely channel linking p_j to another process (i.e., $\exists\ p_x \in \Pi$ such that $QoS(p_j, p_x)$ = *timely*). Otherwise, the identity of p_j is placed in $uncertain_i$. When the initialization phase ends, processes in Π observe identical contents for their respective *live*, *down*, and *uncertain* sets, and a given process is either in *live* or in *uncertain* (ensuring, therefore, the restrictions R0 and R1 of our model).

During the application execution, the contents of the three sets are dynamically modified according to the existence of failures and/or QoS modifications. Next it is described the implemented mechanisms in charge of updating the contents of *live* and *uncertain* (the state detector), and the contents of *down* (the failure detector).

3.2.1. An Implementation of the State Detector

There is a module of the state detector for each process in Π. The module of the state detector associated with p_i executes two concurrent tasks to update the *DS* graph maintained by p_i.

The first task is activated by messages *changeQoS(p_i,p_x,newQoS)* from the local module of QoSP, indicating that the QoS of the channel $c_{i/x}$ has been modified. Upon receiving the *changeQoS* message, the state detector of p_i first verifies whether p_x is not in the $down_i$ set (this is necessary to guarantee R4, see Section 3.3.). If that test turns out to be true, it passes on the information of the new QoS of the channel $c_{i/x}$ to the remote modules of the state detector, and the local *DS* graph is updated accordingly (i.e., $DS_i(p_i, p_x)$ is set to *NewQoS*).

The second task is activated when the failure detector communicates the crash of a process p_x (see details in the next section). The goal of this task is to check whether there is process in the *live* set, say p_y, that had a timely channel to the crashed process. If the channel $c_{x/y}$ is the only timely channel to p_y, it can no longer be detectable and therefore must be moved from *live* to *uncertain*. This is realized by setting all channels linked to the crashed process as untimely in the *DS* graph.

In both tasks, after updating the *DS* graph, the procedure *UpdateState()*, described in Figure 1, is called for each p_x linked to a modified channel, to update the sets $live_i$ and $uncertain_i$, accordingly. Process p_x is moved from $live_i$ to $uncertain_i$ if no timely channel linking p_x is left (lines 1-4) ; p_x is moved from $uncertain_i$ to $live_i$ if a new timely channel linking p_x has been created (lines 6-8).

```
Procedure UpdateState($p_x$, $live_i$, $uncertain_i$)

(1)  if ($p_x \in live_i$) $\wedge$ ($\forall\, p_y \in \Pi : ((p_y \neq p_x) \rightarrow (DS_i[p_x, p_y] = untimely)))$
(2)          then % This is for Rule R3 %
(3)               $live_i \leftarrow live_i - \{p_x\}$;
(4)               $uncertain_i \leftarrow uncertain_i \cup \{p_x\}$
(5)          else
(6)               if ($p_x \in uncertain_i$) $\wedge$ ($\exists\, p_y \in \Pi$: $((p_y \neq p_x) \rightarrow (DS_i[p_x, p_y] = timely)))$
(7)                 then $uncertain_i \leftarrow uncertain_i - \{p_x\}$;
(8)                      $live_i \leftarrow live_i \cup \{p_x\}$
(9)               end_if
(10) end_if
```

Figure 1. Algorithm to Update the Sets $live_i$ and $uncertain_i$

3.2.2. An Implementation of the Failure Detector

Besides maintaining the set $down_i$, the failure detector also maintains the set $suspected_i$ for keeping the identities of processes suspected of having crashed. A process p_i interacts with the failure detector by accessing these sets.

The failure detector works in a *pull* model, where each module (working on behalf of a process p_x) periodically sends "are you alive?" messages to the other modules related to the other processes in Π. The timeout value used for awaiting "I am alive" messages from a monitored process p_y is calculated using the QoSP function $Delay(p_x, p_y)$. The timeout includes the so-called round-trip time (*rtt*)[3], and a safety margin (α), to account for necessary time to process these messages at p_x and p_y.

For timely channels, the calculated timeout is accurate in the sense that network and system resources and related scheduling mechanisms guarantee the *rtt* within a

[3]That is the time to transfer the "are-you-alive?" message from p_x to p_y plus the time to transfer the "I am alive" message from p_y to p_x.

bounded limit. Therefore, the expiration of the timeout is an accurate indication that p_y crashed, and in that case p_y is moved from *live* to *down*. To account for a possible modification of the QoS of the channel $c_{x/y}$, before producing a notification, the failure detector checks it out whether the channel remained timely using the QoSP function *QoS(p_x,p_y)*[4]. On the other hand, if the channel linking p_x and p_y is untimely, the expiration of the timeout is only a hint of a possible crash and, in that case, besides belonging to *uncertain*, p_y is also included in the set *suspected*.

The algorithm for the failure detector for a process p_i, described in Figure 2, is composed by 5 parallel tasks. The parameter *monitoringInterval* indicates the time interval between two consecutive "are-you-alive?" messages sent by p_i. The array $timeout_i[1..n]$ holds the calculated timeout for p_i to receive the next "I-am-alive" message from each process in Π. The function $CT_i()$ returns the current local time and α is a safety margin that also includes the time necessary to process each pair of "are-you-alive?" and "I-am-alive" messages. Task 1 periodically sends a "I-am-alive" message to all processes (actually, the related failure detector modules) after setting a timeout value to receive the corresponding "I-am-alive" message, which in turn is sent by Task 5. Task 2 assesses the expiration of timeouts, and it sends notification messages when the timeouts expire for processes in $live_i$, moving them into the $down_i$ set. Otherwise, if the timeout expires for processes in the $uncertain_i$ set, their identities are also included into the $suspected_i$ set. Task 3 removes a process from the $suspected_i$ set when a message from that process is received. Task 4 handles crash notification messages and updates the sets $down_i$ and $live_i$, accordingly.

3.3. Correctness Proof Sketches of The Implementation

This section presents correctness arguments showing that the above-described mechanisms properly implement the proposed adaptive system model. That is, they assure the correct semantics for the construction and maintainance of the sets $uncertain_i$, $live_i$ and $down_i$, as defined by the rules R0-R5, which are demonstrated by the following lemmas.

Lemma 1 *Rules R0 and R1 are respected*

Proof As discussed in section 3.2., during the system initialization the sets are initialized respecting R0 and R1. To see that R1 holds during the system execution, notice that any inclusion of a given element in a specific set is followed by the removal of this particular element from another set (lines 3-4 and 7-8 of the *UpdateState()* procedure of the state detector (Figure 1) and lines 7-8 and 18-20 of the failure detector (Figure 2). Therefore, R1 cannot be violated for correct processes. $\Box_{Lemma\ 1}$

Lemma 2 *Rule R2 is respected*

Proof To see that a $down_i$ set is never decreasing (first part of R2), observe that the $down_i$ set is only modified to include new elements (lines 7 and 18 of Figure 2). The second part of R2, which states that a $down_i$ set is safe regarding crashes, is respected since a process identifier is only moved into the $down_i$ when a timeout for a timely channel to that process expires (lines 7 and 18 of Figure 2). To assure that the timely channel has not lost its timely condition during the transmission of the "I-am-alive" message, the QoS function

[4]One should observe here that the QoS Provider holds the information and resources related to a given channel even after the crashes of the processes linked by that channel.

```
     Task $T1$: every monitoringInterval do
(1)      for_each $p_j,\ p_j \neq p_i$ do
(2)          $timeout_i[p_j] \leftarrow CT_i() + Delay(p_i, p_j) + \alpha$;
(3)          send are-you-alive($p_i$) to $p_j$
(4)      end_do
     Task $T2$: when $\exists p_j : (p_j \notin down_i) \wedge (CT_i() > timeout_i[p_j]))$ do
(5)      if $((p_j \in live_i)$
(6)        then if $((DS_i[p_i, p_j] = timely) \wedge (QoS(p_i, p_j) = timely))$
(7)            then $down_i \leftarrow down_i \cup \{p_j\}$;
(8)                $live_i \leftarrow live_i - p_j$;
(9)                send notification $(p_i, p_j)$ to every $p_x$ such that $p_x \neq p_i, p_j$
(10)           else do nothing (wait for a remote notification)
(11)         end_if
(12)       else if $((p_j \in uncertain_i) \wedge (p_j \neq suspected_i))$
(13)           then $suspected_i \leftarrow suspected_i \cup \{p_j\}$ end_if
(14)     end_if
     Task $T3$: when "I-am-alive" is received from $p_j$) do
(15)     if $CT_i() > timeout_i[p_j]$ then
(16)       if ($p_j \in suspected_i$) then $suspected_i \leftarrow suspected_i - p_j$ end_if
(17)     end_if
     Task $T4$: when notification($p_x, p_j$) is received do
(18)     if $p_j \notin down_i$ then $down_i \leftarrow down_i \cup \{p_j\}$;
(19)                 if $p_j \in live_i$ then $live_i \leftarrow live_i - p_j$
(20)                          else $uncertain_i \leftarrow uncertain_i - p_j$;
(21)                               if $p_x \in suspected_y$ then
(22)                                   $suspected_i \leftarrow suspected_i - p_j$
(23)                               end_if
(24)                 end_if
(25)     end_if
     Task $T5$: when "Are-you-alive?" is received from $p_j$ do send "I-am-alive"($p_i$) to $p_j$
```

Figure 2. Algorithm for the Failure Detector Module (p_i)

of the QoSP is used (line 6 of Figure 2). Thus, if all resources and related QoS scheduling mechanisms are still available for the channel (i.e., it remains timely), the expiration of the timeout can only happen when the monitored process fails in sending the "I-am-alive" message (i.e., it crashed). $\Box_{Lemma\ 2}$

Lemma 3 *Rule R3 is respected*

Proof This rule is enforced by the application as it is assumed that a process does not ask for the modification of the QoS of a channel from *untimely* to *timely* during the execution of consensus. $\Box_{Lemma\ 3}$

Lemma 4 *Rule R4 is respected*

Proof Let us first consider the first part of R4 and assume by contradiction that $\forall i, j, k$: $p_k \in live_i(t_0) \wedge p_k \in down_i(t_0 + 1) \Rightarrow \exists t_1 > t_0 : p_k \in uncertain_j(t_1)$. If $p_k \in uncertain_j(t_1)$, p_j must see all channels to p_k as untimely at time t_1 (lines 1-4 of the state detector in Figure 1). However, as p_k crashed at time $t_0 + 1$, there was at least one timely channel to p_k (the one linking to the process that detected p_k's crash, say channel $c_{x/k}$). So, in order to have all channels untimely, the changing of QoS for the channel $c_{x/k}$ must be reported to p_j by t_1. Since the state detector related to p_x will not send any QoS modifications for channels connected to crashed processes (see section 3.2.1.), p_j will not change the status of $c_{x/k}$ and, therefore, at least one channel will remain timely at t_1.

As for the second part of R4, let us assume by contradiction that $\forall i, j, k$: $p_k \in live_i(t_0) \wedge p_k \in uncertain_i(t_0 + 1) \Rightarrow \exists t_1 > t_0 : p_k \in down_j(t_1)$. To detect the crash of p_k at time t_1, there must be at least one timely channel linking to p_k (see lines 6-7 in Figure 2). Since all channels are untimely at time $t_0 + 1$, and by assumption channels cannot become timely during consensus execution (R3), then there will be no timely channel linking to p_k at t_1 and, therefore, p_k cannot belong to $down_i(t_1)$. $\Box_{Lemma\ 4}$

Lemma 5 *Rule R5 is respected*

Proof Assume a process p_j crashes at time t_0, and that p_j does appear in the *uncertain* set of any process for an indefinitely long period of time. That is, $\forall t_2 \geq t_1$ we have $p_j \notin uncertain_i(t_2)$, for $t_1 \geq t_0$. And assume by contradiction that there is a time $t_3 \geq t_2$ such that $\forall t_4 \geq t_3$ we have $p_j \notin down_i(t_4)$. First notice that from t_0 process p_j will not send "I-am-alive" messages (Task T1 of Figure 2). Thus, the timeout set to receive such a message from p_j will eventually expire (Task T2 of Figure 2). As p_j does not appear in the $uncertain_k$ set of any process p_k at t_2, it must belong to the sets *live* of all processes (which implies that there is at least one process, say p_x, with timely channel to p_j at t_2). Therefore, the predicate of lines 5-6 of Figure 2 will become true and p_j will be moved from $live_i$ to $down_i$ (lines 7-8 of Figure 2). Aditionally, a notification message is sent and received (as channels are assumed to be reliable) by all correct processes (i.e., processes that do not crash by t_4), which update their *down* sets accordingly (task T4 of Figure 2). $\Box_{Lemma\ 5}$

3.4. A LINUX/JAVA Prototype and its Performance

The *failure detector*, the *state detector*, the consensus algorithm, and the QoS provider have been implemented (as JAVA classes) and tested over a set of networked LINUX workstations. We utilized the RED HAT LINUX 9 (kernel 2.4.20), which includes the *iproute2* package that allows the configuration of the kernel routing tables to control communication flows and to execute traffic control disciplines (such as the ones necessary to implement *DiffServ* functions [1]). We configured the LINUX kernel with *CBQ*(Class Based Queue) forwarding characteristics to create *DiffServ* classes of service (*Express Forwarding* for *timely* channels and *Best Effort* for *untimely* channels), and we used the *u32* and *tcindex* filters to identify packets and to associate them to classes of service.

We carried out experiments to assess the prototype performability. The experimental environment used consisted of a network of three LINUX Pentium III computers (800 MHz, 128 MB RAM) connected through a 100 megabits network. One of the computers worked as a router connecting the other two computers.

We run the quorum-based consensus protocol described in [13] with four processes over the non-router computers (so that a decision quorum would never been formed in a sole machine), and measured the time to reach consensus. In this particular experiment, the processes were connected by two timely channels and four untimely channels, in such a way that three processes were in the *live* set and one process in the *uncertain* set. During the experiment, two processes in the *live* set were forced to fail. The other two processes detected the failures and moved the identities of the crashed processes from the *live* set to the *down* set, thus adjusting the decision quorum (for the two remaining correct processes), and finally achieving consensus. It is important to notice that if all channels were untimely, consensus would not have been achieved, as in this circumstances, a majority of correct processes is required [3] (this point illustrates the benefit of our hybrid and adaptive model). We run this experiment 100 times and collected the time to reach consensus from the first coordinator (that always belonged to the *uncertain* set), and calculated the mean time and standard deviation for the 100 runs (see first column of Table 1).

We also run two other experiments. One with all channels untimely and the other with all channels timely for a set of three processes, both experiments without failures. The second and third columns of Table 1 show the mean time and standard deviation for 100 runs of each experiment, respectively.

	$\lvert uncertain \rvert = 1$ and $\lvert live \rvert = 3$	$\lvert uncertain \rvert = 3$	$\lvert live \rvert = 3$
$Mean\ Time$	154 ms	50.93 ms	49.96 ms
$Standard\ Deviation$	78.64 ms	21.18 ms	26.85 ms

Table 1. Mean time to reach consensus

4. Conclusion

This paper showed how to exploit QoS facilities to implement an adaptive model for fault-tolerant distributed computing, that encompasses both the synchronous model (where there are time bounds on processing speed and message delay) and the asynchronous model (where there is no time bound).

This new model can be particularly relevant for applications that require run-time adaptiveness characteristics, such as distributed multimedia systems, where previously negotiated QoS cannot always be delivered between processes. In order to specify the underlying functionality needed to implement the adptive model, a mechanism (called the QoS provider) has been developed and implemented. Thanks to this modularity dimension of the approach, porting the model implementation to a given environment requires only to implement the QoS Provider functions that have been defined. The proposed system has been implemented in JAVA and tested over a set networked LINUX workstations, equipped with QoS capabilities.

The work presented in this paper is part of a QoS middleware infrastructure intended for adaptive fault tolerant applications, being developed in the Distributed System Laboratory (LaSiD) at UFBA, in cooperation with the distributed systems and networks research group at UFPE. In [13] we introduced an adaptive distributed computing model and related consensus algorithm that can benefit from the QoS support and fault tolerant mechanisms presented in this paper. Other efforts are needed to complement the functionalities of the infrastructure being developed. For instance, the development of replication management and group communication services and a tool capable of mapping fault tolerant and QoS specifications into the infrastructure services, are efforts planned for future work.

References

[1] Blake S., Black D., Carlson M., Davies E., Wang Z. and Weiss W., An Architecture for Differentiated Services, *RFC 2475*, June, 1998.

[2] Chandra T.D., Hadzilacos V. and Toueg S., The Weakest Failure Detector for Solving Consensus. *Journal of the ACM*, 43(4):685-722, July, 1996.

[3] Chandra T.D. and Toueg S., Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225-267, March, 1996.

[4] Chu F., Reducing Ω to $\diamond\mathcal{W}$. *Information Processing Letters*, 67(6):289-293, September, 1998.

[5] Cristian F. and Fetzer C., The Timed Asynchronous Distributed System Model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):642-657, June, 1999.

[6] Dolev D., Dwork C. and Stockmeyer L., On the Minimal Synchronism Needed for Distributed Consensus. *Journal of the ACM*, 34(1):77–97, January, 1987.

[7] Dwork C., Lynch N. and Stockmeyer L., Consensus in the Presence of Partial Synchrony. *Journal of the ACM*, 35(2):288-323, April, 1988.

[8] Fischer M.J., Lynch N. and Paterson M.S., Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374-382, April, 1985.

[9] Gorender S. and Macêdo R., Fault-tolerance in Networks with QoS, *Technical Report Number RT001/03, Distributed System Laboratory (LaSiD), UFBA (in Portuguese)*, 2003.

[10] Gorender S. and Macêdo R., Um Modelo para Tolerância a Falhas em Sistemas Distribuídos com QoS. *Anais do XX Simpósio Brasileiro de Redes de Computadores (SBRC'02),* Maio, 2002, pp.277-292

[11] Hurfin M., Macêdo R., Tronel F., and Raynal M., A Consensus Protocol based on a Weak Failure Detector and a Sliding Round Window, *Proceedings of the 20th IEEE Int. Symposium on Reliable Distributed Systems (SRDS'01),* New Orleans, pp. 120-129, October 2001.

[12] Macêdo R., Silva F., The mobile groups approach for the coordination of mobile agents, *Journal of Parallel and Distributed Computing (JPDC),* Elsevier 65(3), pp. 275-288, March 2005.

[13] Macêdo, R., Gorender, S., and Raynal, M. A set-based Adaptive Distributd Computing Model and its Application to Distributed Consensus *Technical Report Number RT002/04*, Distributed System Laboratory (LaSiD), UFBA, 2004. A version of this paper will appear in IEEE/IFIP Int. Conference on Computer Systemas and Networks Yokohama, Japan, June/2005 (full paper).

[14] Hadzilacos V. and Toueg S., Fault-tolerant Broadcasts and Related Problems In *Distributed Systems*, ACM Press (S. Mullender Ed.), New-York, pp. 97-145, 1993.

[15] Mostefaoui A. and Raynal M., Solving Consensus Using Chandra-Toueg's Unreliable Failure Detectors: a General Quorum-Based Approach. *Proc. 13th Symposium on Distributed Computing (DISC'99),* Springer Verlag LNCS #1693, pp. 49-63, September 1999.

[16] Nahrstedt K. and Smith J. M., The QoS Broker, *IEEE Multimedia,* 2(1):53-67, 1995.

[17] Ren Y., Cukier M. and Sanders W.H., An Adaptive Algorithm for Tolerating Values Faults and Crash Failures. *IEEE Transactions on Parallel and Distributed Systems,* 12(2):173-192, February 2001.

[18] Siqueira F. and Cahill, V., Quartz: A QoS Architecture for Open Systems, *Proceedings of the 18th Brazilian Symposium on Computer Networks*, pages 553-568, May 2000.

[19] van Renesse R., Birman K., Hayden M., Vaysburd A. and Karr D., Building Adaptive Systems Using Ensemble. *Software Practice and Experience*, 28(9):963-979, July 1998.

[20] Veríssimo P. and Casimiro A., The Timely Computing Base Model and Architecture. *IEEE Transactions on Computers, Special Issue on Asynchronous Real-Time Systems*, 51(8):916-930, August 2002.