To appear in the proceedings of the international Conference on Availability, Reliability and Security (ARES 2009), Fukuoka, Japan, March/2009

Perfect Failure Detection in the Partitioned Synchronous Distributed System Model

Raimundo José de Araújo MACÊDO and Sergio GORENDER
Distributed System Laboratory (LaSiD)
Computer Science Department
Federal University of Bahia
Campus de Ondina, 40170-110, Salvador, Brazil

{macedo|gorender}@ufba.br

***Abstract*—In this paper we show that it is possible to implement a perfect failure detector P (one that detects all faulty processes if and only if those processes failed) in a non-synchronous distributed system. To realize that, we introduce the *partitioned synchronous system* (Spa) that is weaker than the conventional synchronous system. From some properties we introduce (such as *strong partitioned synchrony*) that must be valid in Spa and a trivially implementable Timeliness oracle, we show how to implement P in Spa. Moreover, we show that even if *strong partitioned synchrony* is not valid, we are still able to take advantage of the existing synchronous partitions for improving the robustness of applications, by introducing a partially perfect failure detector named xP. We also discuss how applications can benefit from these failure detectors and present some related experimental data. The necessary properties and algorithms for implementing P and xP are presented in the paper, as well as the related correctness proofs.**

I. INTRODUCTION

The ability to solve certain fault-tolerant (FT) problems in distributed systems is closely related to the existence of an adequate system model upon which such problems can be proved correct. In this context, in the last decades, researchers have proposed a variety of system models, where the asynchronous (or time-free) and synchronous (or time-based) have attracted most attention due to the fact that these models can be considered extreme cases in terms of FT problem solvability. For instance, under process crashes and reliable channels, the reliable multicast problem is solvable in both models [1], [2], whereas distributed consensus can be solved in the synchronous model, but not in the asynchronous model [3].

Motivated by the impossibility result in asynchronous systems, researchers have proposed a number of partially synchronous distributed system models, where the consensus problem is solvable [4], [5], [6], [7], [8]. In particular, Chandra and Toueg [8] proposed a FT solvability model for asynchronous distributed systems based on axiomatic properties of a number of failure detector classes, identifying the classes for which there is a solution for consensus with the minimum run-time system guarantees (i.e., the $\diamond W$ class or its equivalent, the $\diamond S$ class).

The strongest class (the one requiring the maximum run-time system guarantees) presented by Chandra and Toueg is the perfect failure detector class, denoted P and possessing the axiomatic properties of *strong completeness* (eventually faulty processes are detected by correct processes) and *strong accuracy* (correct processes are never detected). In the same paper, Chandra and Toueg showed that it would be possible to implement P in a synchronous distributed system by means of *timeouts*.

One of the main difficulties in solving FT problems in the asynchronous model is the fact that in this model it is impossible to distinguish between the crash failure of a process p and a late message from p. As a consequence, researchers have long believed that the FT problem solvability power of asynchronous systems equipped with a perfect failure detector would be equivalent to the FT problem solvability power of the synchronous system. Such believe was properly contested by Charron-Bost, Guerraoui and Schiper [9], who showed that the synchronous model (denoted by them as Ss) is strictly stronger than the asynchronous model equipped with perfect failure detectors (denoted by them as Sp). In their work they demonstrated that some problems that are solvable in Ss, are not in Sp, even when they do not involve timeliness specifications (problems with timeliness specifications are, by definition, impossible to be solvable in Sp). Besides, they proved that a solution for the uniform consensus problem - a fundamental FT problem - is made more efficient in Ss when compared with Sp, by using a round-based time complexity measure.

Founded on the idea that *timeouts* or temporal hypotheses can be used in Ss for implementing perfect failure detectors, Charron-Bost et al proceeded in their article [9] to conclude that the comparison between the two models, Ss and Sp, would be made by identifying the properties of Ss that would be lost in the axiomatic transformation introduced by Chandra and Toueg. As a consequence, such discussions seem to imply that the implementation environment of both models, Ss and Sp, would necessarily be the distributed synchronous system where there are known upper bounds on process computation steps and communication channel message delivery delays. Therefore, choosing the strongest model (Ss) in such circumstances is the obvious choice, as the implementation cost or environment guarantees of both models would necessarily be the same (the synchronous system). On the other hand, given the FT problem solvability power of Sp, finding alternative implementations for Sp that require less environment guarantees (in certain measures) is an attractive challenge that must be faced by researchers.

In the present paper we address this challenge by showing that P can indeed be implemented in a system that is weaker and requires less temporal guarantees from the run-time environment when compared with the classical Ss, which we named Spa (*partitioned synchronous*). Such a possibility makes Spa a proper choice in certain scenarios where it is not possible to implement Ss.

The implementation of P in Spa (therefore, Sp) does not require the existence of a synchronous *wormhole* [10] or, equivalently, a synchronous *spanning tree* [11], where it would be possible to implement synchronous computations among all system processes, such as internal clock synchronization. In the Spa system that we propose, system components (subsets of processes and channels) need to be synchronous but these components do not need to be interconnected by timely channels (we call this property *strong partitioned synchrony*), which makes it impossible to execute synchronous computations that include all system processes (that is why we call Spa a non-synchronous system). Even so, we show that it is possible to implement P in Spa equipped with a trivially implemented Timeliness oracle. Moreover, we show that even if *strong partitioned synchrony* is not valid in Spa, we are still able to take advantage of the existing synchronous components for improving the robustness of applications, by introducing a partially perfect failure detector named xP. However, in order to implement P in Spa, there must exist at least 1 correct process in each complete synchronous subgraph (or synchronous partition - see section III for a precise definition of synchronous partition). Such a requirement guarantees that the properties of P are valid despite $n-k$ process crashes, where k is the number of synchronous partitions and n the number of processes. Observe that the *strong partitioned synchrony* as well as the 1-correct process per synchronous partition requirements are not unrealistic. Consider, for example, k real-time computer clusters that are mutually interconnected by the Internet. Because of the real-time operating system and network guarantees, all processes and channels inside a cluster are synchronous and it is reasonable to assume that at least one computer of a cluster will not fail during an interval of interest (e.g., during an application execution). Therefore, implementing P in Spa with *strong partitioned synchrony* turns out to be an attractive alternative in such settings where the assumptions of a synchronous system do not hold.

Finally, we observe that, contrarily of what is generally believed, being perfect (for a detector) does not necessarily mean being timely - in fact, P is actually only defined in terms of axiomatic properties (completeness and accuracy) that do not refer to any timeliness properties. Such misconception makes it counterintuitive the observation that P can indeed be implemented in a non-synchronous system such as Spa. We believe that this paper also contributes to a better clarification of such a common misconception.

The remainder of this paper is structured as follows. Related work is discussed in section II. The model Spa is described in section III. The implementation algorithms for P and xP in Spa and related correctness proofs are presented in sections IV and V, respectively. A discussion on the benefits of this new approach is presented in section VI. Finally, conclusions are presented in section VII.

II. Related Work

The solvability of FT problems, such as consensus [1], depends decisively on the existence of guaranteed upper bounds for message transmission (δ) and process scheduling delays (ϕ) (i.e., timely channels and timely process executions). Such upper bounds can only always be guaranteed in synchronous systems. On the other hand, FT services can also be guaranteed in some asynchronous or partially synchronous environments where the "synchronous behavior" is verified during sufficient long periods of time, even if such a behavior is not continuously assured. Thus, such partially synchronous systems can alternate between "synchronous behavior", with time bound guarantees, and "asynchronous behavior", with no time bound guarantees. So, in some sense, these systems are hybrid in the time dimension. For example, the timed asynchronous system depends on a sufficient long stability period to deliver the related FT services, and the system behavior can alternate between stable and unstable periods (i.e., synchronous and asynchronous) [6]. The GST - Global Stabilization Time - assumption is necessary to guarantee that $\diamond\mathcal{S}$-based consensus protocols eventually deliver the proper service [8] (i.e., to work correctly, the system must eventually exhibit a "synchronous behavior").

There are other models that consider not only the hybridism in the time dimension, but also that the system is composed by a synchronous and an asynchronous part at the same time. So, we can regard such systems as being hybrid in the space dimension. This is the case of the TCB model, which relies on a synchronous wormhole to implement FT services [10].

As in [12], we assume that our underlying system model is capable of providing distinct QoS communication guarantees. Differently from the work in [12], in the present paper we assume that the underlying system is also capable of providing distinct process scheduling guarantees, with both timely (with guaranteed deadlines) and untimely (best-effort) process executions - such behavior can be implemented over hybrid real-time systems and networks [13]. Because timely and untimely channels and processes may exist in parallel, our underlying system model can be hybrid in space (in this sense, similar to TCB), but, differently from TCB, the nature of *Spa*'s hybridism does not require that all processes are interconnected by timely channels (which would characterize a synchronous wormhole).

Regarding the implementation of perfect failure detectors, it is clear that such an implementation is not possible in pure asynchronous systems, otherwise a solution for the consensus problem would be possible, contradicting the FLP result [3]. On the other hand, it has been shown that perfect failure detectors are not implementable in partially synchronous systems either, where some stability condition, such as GST, eventually holds [14].

Nonetheless, some researchers have proposed alternative models for implementing perfect failure detectors. In the work of Veríssimo and Casimiro, they show how a synchronous wormhole - in a hybrid architecture - can be used for implementing a perfect failure detection service [10], [15]. So, they can also implement P. Fetzer has proposed a perfect failure detection service for the timed-asynchronous system model augmented with a hardware watchdog [16], which is used to force the crash failure of computers. Such forced crash failures avoid that false suspicions are passed on, thus simulating perfect failure detection (or fail-stop behavior). However, implementing P in Spa does not require a synchronous wormhole, nor processes are forced to crash.

To the best of our knowledge no previous work has addressed a system model with the properties we presented in this paper and related detectors (P and xP)[1].

III. The Partitioned Synchronous Model - Spa

System Model and Assumptions

A system is made up of a set $\Pi = \{p_1, p_2, ..., p_n\}$ of processes sited on possibly distinct networked computers and a set $\chi = \{c_1, c_2, ..., c_m\}$ of communication channels. Networked computers form arbitrary network topologies and processes communicate by using a transport level communication protocol that implements process-to-process communication. Such process-to-process communication defines communication channels that may include several intermediate network level communication links. Therefore, a communication channel c_i connecting processes p_i and p_j defines a "is able to communicate" relation between p_i and p_j, rather than a network level link connecting the machines that host p_i and p_j. We assume that the system formed by processes and communication channels forms a simple complete undirected communication graph $DS(\Pi, \chi)$ with $(n \times (n-1))/2$ edges - that is, each channel c_i of χ links only distinct processes of Π and each process of Π is linked to every other process of Π. Network partitioning is not considered.

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), and has access to a local hardware clock with drift rate ρ. Processes and channels can be timely or untimely[2]. A given process p_i is timely if there exists a known upper bound ϕ for the execution of a computation step by p_i. Similarly, a channel c_i is timely if there exists a known upper bound δ for the transmission delay of a message in c_i, and c_i connects two timely processes. Otherwise, processes and channels are said to be untimely and the related time bounds are finite but arbitrary. It is also assumed that δ can be used as an estimation for the upper bound transmission delay in an untimely channel.

[1]A preliminary version of this paper has been published in Portuguese in a workshop of the Brazilian community of dependability [17].

[2]Timely/untimely is equivalent to synchronous/asynchronous as presented in [4]. However, the partially synchronous models considered in [4] do not consider hybrid configurations, as we do in this paper, where some processes/channels are synchronous and others asynchronous.

A communication channel $c_i = (p_i, p_j)$ implements message transmission in both directions, from p_i to p_j and from p_j to p_i. δ and ϕ are parameters of the underlying computing system, provided by appropriate real-time operating system and network mechanisms. We assume that once defined at creation time the quality-of-service (QoS) of processes and channels do not change over time. Later, in this paper, we discuss on the implications and necessary mechanisms to cope with the relaxation of this static assumption. We also assume that processes in Π know the QoS of all processes and channels before the application execution.

It is assumed the existence of a Timeliness oracle defined by the function QoS that maps processes and channels to values T and U (timely and untimely). Therefore, the Timeliness oracle provides executing processes with the current QoS of processes and channels in terms of timeliness. The Timeliness oracle is assumed to be accurate: the execution of QoS(x) at a real time t returns T/U if and only if the QoS of the element x (be it a process or a channel) at time t is timely/untimely. Since the QoS of processes and channels are assumed to be static and known, a trivial implementation of the Timeliness oracle can be realized during an initialization phase: for instance, by maintaining two boolean arrays, one for processes and another for channels, whose true and false values represent timely and untimely, respectively.

Channels are assumed to be reliable: they neither lose nor alter sent messages. Faulty processes are assumed to fail by halting prematurely, without producing further actions, and correct processes do not fail during a time interval of interest.

Synchronous/Asynchronous Subgraphs

Given $\Pi' \subseteq \Pi$, $\Pi' \neq \emptyset$ and $\chi' \subseteq \chi$, a connected (communication) subgraph $C(\Pi', \chi') \subseteq DS(\Pi, \chi)$ is synchronous if $\forall p_i \in \Pi'$ and $\forall c_j \in \chi'$, p_i and c_j are timely. If these conditions do not hold, $C(\Pi', \chi')$ is a non-synchronous subgraph. We use the notation Cs to refer to a synchronous subgraph and Ca for a non-synchronous subgraph.

Synchronous Partitions

Given a $Cs(\Pi', \chi')$, we define a synchronous partition as the maximum synchronous subgraph $Ps(\Pi'', \chi'')$, such that $Cs \subseteq Ps$. In other words, DS has no $Cs'(\Pi''', \chi''') \supset Cs$ with $|\Pi'''| > |\Pi''|$.

We assume that there is at least one correct process in each synchronous partition[3].

[3]As previously observed, such an assumption is realistic if one considers computer clusters (the synchronous partitions) with reasonable sizes - say, with more than three computers per cluster. This is actually the case for conventional synchronous systems where computer units are usually spreed over a local area network

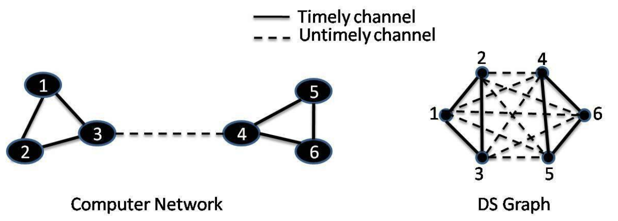

Figure 1. An example of a computer network and its representation in the DS graph.

Figure 1 illustrates the DS graph (right side) that represents two synchronous local area networks interconnected by an asynchronous channel (left side). Such a composition may well represent two real-time computer clusters interconnected via the Internet. In the example, each computer node (numbered 1 to 6) of the network hosts only one process which is timely. It can be observed in DS that, for instance, the subgraphs $Cs_1(\{1,2,3\},\{(1,2),(1,3),(2,3)\})$ and $Cs_2(\{4,5\},\{(4,5)\})$ are synchronous and $Ca_1(\{3,4,5\},\{(3,4),(4,5),(3,5)\})$ is non-synchronous. Also, note that there are two synchronous partitions in DS: $Ps_1(\{1,2,3\},\{(1,2),(1,3),(2,3)\})$ and $Ps_2(\{4,5,6\},\{(4,6),(5,6),(6,5)\})$.

It is interesting to notice that every Ps forms a clique. That is, Ps is a subgraph of DS where for every two vertexes in Ps there exists an edge connecting the two (in this case an edge that represents a timely channel).

In the distributed system Spa, we assume the following property necessary for implementing P.

strong partitioned synchrony: $\forall p_i \in \Pi, \exists Ps \subset DS, p_i \in Ps$.

In the example of Figure 1, *strong partitioned synchrony* holds because Ps_1 and Ps_2 together include all processes in DS. The *strong partitioned synchrony* hypothesis will be relaxed later in the paper when the failure detector xP is introduced, where processes may belong to non-synchronous subgraphs. It is important to observe that for a given distributed system, *strong partitioned synchrony* does not imply the existence of a synchronous *wormhole* [10] or a synchronous *spanning tree* [11], that is, a subgraph Cs where $\Pi = \Pi'$ (note that in the example of Figure 1 there is no such a Cs subgraph that includes all vertexes of DS).

Finally, observe that because $Ps \subset DS$ in the specification of Spa with *strong partitioned synchrony*, such specification excludes a synchronous system with a unique synchronous partition with all processes of Π (Ss). Indeed, it is easy to see that Spa with *strong partitioned synchrony* and a Timeliness oracle is weaker

```
Task $T1$: every monitoringInterval do
(1)  for_each $p_j$, $p_j \neq p_i$ do
(2)      $timeout_i[p_j] \leftarrow CT_i() + 2\delta + \alpha$;
(3)      send "are-you-alive?" message to $p_j$
(4)  end
Task $T2$: when $\exists p_j : (p_j \notin faulty_i) \wedge (CT_i() > timeout_i[p_j])$ do
(5)  if $(QoS(c_{i/j}) = T)$ then
(6)      $faulty_i \leftarrow faulty_i \cup \{p_j\}$;
(7)      send notification $(p_i, p_j)$ to
         every $p_x$ such that $p_x \neq p_i \wedge p_x \neq p_j$
(8)  else do nothing (wait for a remote notification)
(9)  end_if
Task $T3$: when "I-am-alive" is received from $p_j$ do
(10) $timeout_i[p_j] \leftarrow \infty$; /* cancels timeout */
Task $T4$: when notification$(p_x, p_j)$ is received do
(11) if $p_j \notin faulty_i$ then
         $faulty_i \leftarrow faulty_i \cup \{p_j\}$;
(12) end_if
Task $T5$: when "are-you-alive?" is received from $p_j$ do
(13) send "I-am-alive" to $p_j$
```

Figure 2. Algorithm of the perfect failure detector module of process p_i

than Ss in terms of FT solvability power, as such a configuration excludes the possibility of solving problems with timeliness specifications (because of the untimely channels linking processes in distinct synchronous partitions).

Assuming the model Spa with *strong partitioned synchrony* and a Timeliness oracle as presented in the previous sections, in the following we show that a failure detector of class P can be implemented in such a model.

IV. Failure Detector P in Spa

The failure detector executes in modules, one for each process of the system. The failure detector modules monitors each process sending *"Are-you-alive?"* messages, and waiting for *"I-am-alive"* messages in response. If a timeout expires without the reception of the corresponding *"I-am-alive"* message, the monitored process is detected and its identification is inserted into a set named $faulty$.

The Figure 2 presents the algorithm of a module of the failure detector, composed of 5 tasks. The first task (*T1*) sends *"Are-you-alive?"* messages to each process of the system (line 3), every monitoring interval. The task calculates a timeout for the reception of the corresponding *"I-am-alive"* messages, from the monitored processes. The timeouts are calculated by the formula $CT_i()+2\delta+\alpha$, where CT_i is the current time for process p_i, δ is the delay bound for timely channels, as defined in section 3, and α is a safety margin[4].

[4]α should account for the execution time of line 2, the execution time of task *T5*, and the clock drift ρ

The second task (*T2*) monitors the processes of the system. When a timeout expires without the reception of the corresponding *"I-am-alive"* message, and the communication channel linking p_i to p_j (denoted $c_{i/j}$) used to monitor the process is timely (line 5), the monitored process (p_j) is notified as failed, and its identification is inserted into the $faulty$ set (line 6). A $notification(p_i, p_j)$ message is sent to every process in Π (line 7). If the channel is untimely, the algorithm does nothing (line 8).

The third Task (*T3*) executes when an *"I-am-alive"* message arrives, indicating that the monitored process is still alive. The timeout defined for the reception of the message is canceled (line 10). Task four (*T4*) is executed when a $notification(p_x, p_j)$ message arrives, informing that process p_x detected the crash of process p_j. The identification of p_j is inserted into the $faulty$ set (line 11). Finally, task five (*T5*) executes when an *"Are-you-alive?"* message is delivered. It replies by sending an *"I-am-alive"* message (line 13).

When executing in Spa with *strong partitioned synchrony* property and the Timeliness oracle, the algorithm presented in Figure 2 implements a failure detector of class P (i.e., it possesses the strong completeness and accuracy properties).

Theorem 1: The failure detector algorithm satisfies the Strong Completeness property - Every faulty process will eventually be detected by every correct process.

Proof

The proof is constructed by contradiction. Assume that process p_x fails in time t and that a correct process, say p_y, does not detect this failure. Let the time $t' > t$ be such that, after time t', p_y has received the last *"I-am-alive"* message sent by p_x and the local time for p_y is greater then the timeout for the reception of the next *"I-am-alive"* message from p_x.

p_y starts executing task *T2* of the algorithm of Figure 2 at t', as condition $CT_y() > timeout_y[p_x]$ is satisfied. Because the *strong partitioned synchrony* property is assumed, p_x is a member of a Ps, and consequently, it has a timely channel with a correct process (observe that by assumption there exists at least one correct process in each Ps - thus, no synchronous partition is ever destroyed). We have two possible executions, depending on the QoS of the channel $c_{x/y}$.

1) channel $c_{x/y}$ is timely (p_x and p_y are members of the same Ps): The condition of the if command of line 5 of the algorithm is satisfied and the identification of process p_x is inserted into the $faulty_y$ set. p_y sends a $notification(p_x, p_y)$ message to all processes (line 7). Consequently, after time t‘,

p_y detects p_x permanently.

2) channel $c_{x/y}$ is untimely (p_x and p_y are members of distinct synchronous partitions): The condition of the **if** of line 5 of the algorithm is not satisfied, because channel $c_{x/y}$ is untimely. Executing Task *T2*, p_y does nothing (line 8). As p_x is member of a Ps and every Ps has a correct process, there is a correct process, say p_z, such that p_z and p_x are members of the same Ps and $c_{x/z}$ is timely. As demonstrated in item 1 of the proof, p_z detects p_x and sends $notification(p_x, p_z)$ messages to all processes. As the channels are reliable, p_y eventually receives the $notification(p_x, p_z)$ message, and executes Task *T4*, inserting p_x identity into the $faulty_y$ set. Consequently, p_y detects p_x.

The initial assumption is contradicted.

$\square_{Theorem\ 1}$

Theorem 2: The failure detector algorithm satisfies the *Strong Accuracy* property - No correct process is detected.

Proof

This proof is constructed by contradiction. We first assume that a process p_x is correct, and that another process p_y detects p_x.

As the *strong partitioned synchrony* property is satisfied, every process is a member of a synchronous partition. We have two possibilities of execution for p_y to detect p_x:

1) p_y directly detects p_x by executing lines 6 and 7 of the algorithm): However, if channel $c_{x/y}$ is timely (i.e., p_y and p_x are members of the same partition), the messages sent by p_x to p_y are delivered before the corresponding timeouts. In this case, the condition $CT_y > timeout_y[p_x]$ is never satisfied, and p_y does not execute Task *T2* of the algorithm to detect p_x. If channel $c_{x/y}$ is untimely, p_y does nothing (line 8).
2) p_y receives a $notification(p_z, p_x)$ message from process p_z (p_y executes Task *T4* of the algorithm): As we showed in item 1 of the proof, no process executes lines 6 and 7 of the algorithm and then detects p_x, consequently, no process sends a $notification(p_z, p_x)$ message, executing line 7 of the algorithm, and notifying p_x's failure. As a result, p_y never receives a $notification(p_z, p_x)$ message, and does not execute Task *T4* of the algorithm to detect p_x's failure.

Therefore, p_y never detects p_x's failure, contradicting the initial assumption.

$\square_{Theorem\ 2}$

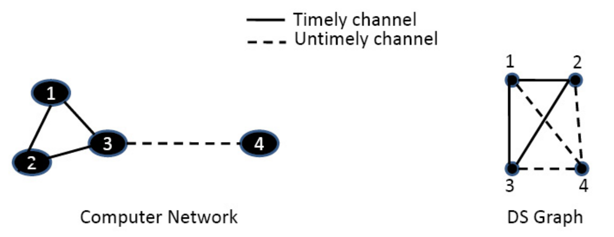

Figure 3. An example of a DS graph with weak partitioned synchrony.

V. Partially Perfect Failure Detector in Spa

In *Spa*, even when *strong partitioned synchrony* cannot be satisfied, it is possible to take advantage of the existing synchrony for implementing perfect failure detection, provided that some subgraphs are synchronous. Such a weaker partitioned synchrony property is presented below.

weak partitioned synchrony: The set of processes that belongs to synchronous partitions is a proper subset of Π. More formally, $\exists p_i \in \Pi, \forall Ps_x \subseteq DS, p_i \notin Ps_x$.

Figure 3 illustrates an example of a DS graph that represents a synchronous local area network connected by an untimely channel with a non-real time computer (for example, this could be a real-time cluster connected to an ordinary PC via an *Ethernet* card). In the example, each node of the network (numbered 1 to 4) hosts only one process. Processes 1 to 3 are timely and process 4 is untimely. It can be observed in DS that, for instance, the subgraph $Cs_1(\{1,2,3\}, \{(1,2),(1,3),(2,3)\})$ forms a synchronous partition, whereas $Ca_1(\{2,3,4\}, \{(2,3),(3,4),(4,2)\})$ forms a non-synchronous subgraph.

In the example, *weak partitioned synchrony* holds because process p_4 does not belong to any synchronous partition of DS.

As we show below, if the *weak partitioned synchrony* holds in Spa, it is possible to implement a failure detector of a new class, that we call *partially perfect failure detector*, denoted xP. Failure detectors of this new class satisfies the new properties *Partially Strong Completeness* (every process that belongs to a synchronous partition will be detected by every correct process, in a finite time) and *Partially Strong Accuracy* (no process that belongs to a synchronous partition can be erroneously detected by any other process). We implement this failure detector with the same algorithm used for P, which was presented in Figure 2.

The implementation of the xP failure detector also uses the Timeliness oracle and executes as described in the previous section. The correctness proofs for the *Partially Strong Completeness* and the *Partially Strong Accuracy* properties assumes Spa with *weak partitioned synchrony* and a Timeliness oracle and are similar to the proofs for the properties of P, and will be omitted here due to lack of space.

VI. Discussion: how applications can benefit from P and xP

The first obvious advantage of using P in Spa (that is, implementing Sp) is the ability to solve FT problems in a system model weaker than the synchronous system (Ss) where a theoretical solution has been presented for the asynchronous systems equipped with P (Sp): for instance, the *global data computation* problem [18] and atomic multicast problem [19].

On the other hand, there are many actual system settings and applications that can benefit from this new approach, where the underlying system is non-synchronous. Consider for instance, grid clusters interconnected via the Internet, and that tasks are distributed among distinct clusters (for instance, for parallel computations). Maintaining a mutually consistent view of the functioning processes distributed in distinct clusters is an important requirement, for instance, for re-executing failed tasks when the task coordinator is replicated for improving availability. Such a functionality can be achieved with P implemented in Spa, where each grid cluster forms a synchronous partition. However, as connections among the clusters are realized via TCP/IP, the whole grid system is non-synchronous.

We have not assumed a specific implementation for Spa and its required mechanisms. This can be carried out, for instance, by using hybrid real-time systems, where there are guaranteed deadlines for critical tasks and best-effort response times for aperiodic or non-critical tasks (for instance, by using aperiodic servers [20], [21], [13]). The same kind of solution can be used in the network level that must be a hybrid real-time network. Another possible approach is to use QoS architecture solutions [22], and we have developed a prototype of the failure detector presented in this paper in such an environment. In order to test the failure detector, we executed it with the adaptive consensus solution presented in [12] in the execution scenario presented in the example of Figure 3, made up of four processes, three of them belonging to a synchronous partition (processes p_1, p_2, and p_3). In such a scenario the *weak partitioned synchrony* is valid and, therefore, xP failure detector implementable. This example illustrates the advantage of using Spa with the adaptive consensus: the ability to tolerate a greater number of failures compared with the conventional asynchronous system.

During the experiment, processes p_2 and p_3 were forced to fail. The other two processes detected the failures thus adjusting the decision quorum (for the two remaining correct processes), and finally achieving consensus. It is important to notice that if there was no synchronous partition, consensus would not have been achieved, as in this circumstances, a majority of correct processes is required [8]. We also executed this same scenario without failures (the second column of the Table). Notice that the mean time taken for the experiment with failures (first column of Table I) is larger than the mean time for the experiment without failures. This larger time is due to the extra detection time related to the two crashed processes of the first experiment. However, the point to be illustrated here is the ability to tolerate more faults provided by xP in Spa, not the performance figures of the consensus implementation. Thus, this point will not be further discussed in this paper.

	with failures	without failures
$Mean\ Time$	147,9 ms	79,38 ms
$Standard\ Deviation$	53,08 ms	31,53 ms
$Median$	134,5 ms	80 ms

Table I
Mean time to reach consensus in Spa with xP.

Relaxing the static QoS requirement

Consider now that QoS of channels and processes may change over time. In such conditions, the implementation mechanisms must take into account the QoS changes so that the *accuracy* of the oracle as well as the *strong partitioned synchrony* remain valid despite QoS changes. Observe that though we have assumed QoS of processes and channels once defined do not change, the specification of the Timeliness oracle given in section III accounts for the fact that QoS may change: the execution of QoS(x) at a real time t returns T/U if and only if the QoS of the element x (be it a process or a channel) at time t is timely/untimely. These features can be achieved in a dynamic QoS environment if the QoS management system judiciously assesses the QoS of channels and processes using a proper monitoring system. Besides, there are also implications on the failure detector implementation: for instance, before detecting a process failure, the failure detector must inquiry the QoS management system to assure that the QoS of the related component has not downgraded (from timely to untimely). Such required characteristics have actually been implemented in a software called the *QoS Provider* [12] as a layer between the processes and underlying system. The *QoS Provider* creates, manages and monitors the communication channels for processes. Its description is, however, beyond the scope of this paper whose aim was to discuss the conditions for perfect failure detection in a non-synchronous system. Finally, we observe that though the failure detector presented here has a structure similar to the one of the failure detector discussed in [12], it actually behaves in a distinct way because of the *strong partitioned synchrony* and *weak partitioned synchrony* assumed in this paper.

VII. Conclusions

In the present paper we addressed the challenge of implementing perfect failure detection in non-synchronous systems. In the system we introduced, named Spa, distributed processes are not required to form a synchronous *wormhole* where it would be possible to execute distributed synchronous computations (such as deterministic internal clock synchronization). In the system Spa, system components (subsets of processes and channels) need to be synchronous but these components do not need to be interconnected by synchronous channels (we called this property *strong partitioned synchrony*). Even so, we showed that it is possible to implement P in Spa. Moreover, we showed that even if *strong partitioned synchrony* cannot be sustained, we are still able to take advantage of the existing synchronous partitions for improving the robustness of applications, by introducing a partially perfect failure detector named xP. All the algorithms for implementing P and related correctness proofs were presented in the paper. We have developed a prototype of P and xP over a QoS architecture infrastructure, and some performance figures were presented reporting the use of xP in an adaptive consensus. The failure detectors presented in this paper can be used for improving the robustness of distributed applications in non-synchronous environments (for instance, grid systems where synchronous clusters are interconnected via asynchronous channels), where it is not possible, or cost-effective, to implement a synchronous system or a hybrid architecture with a synchronous wormhole. To the best of our knowledge no previous work has addressed a system model with the properties we presented in this paper and related detectors (P and xP).

References

[1] N. A. Lynch, *Distributed Algorithms.* Morgan Kaufmann Publishers, Inc., 1996.

[2] V. Hadzilacos and S. Toueg, *Distributed Systems (Edited by Sape Mullender). Chapter 5. Fault-Tolerant Broadcasts and Related Problems.* Addison-Wesley Pub Co, 1993.

[3] M. J. Fisher, N. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM*, vol. 32, no. 2, pp. 374–382, april 1985.

[4] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *Journal of the ACM*, vol. 35, no. 2, pp. 288–323, april 1988.

[5] D. Dolev, C. Dwork, and L. Stockmeyer, "On the minimal synchronism needed for distributed consensus," *Journal of the ACM*, vol. 34, no. 1, pp. 77–97, january 1987.

[6] F. Cristian and C. Fetzer, "The timed asynchronous distributed system model," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 6, pp. 642–657, june 1999.

[7] L. Lamport, "The part time parliament," *ACM Trans. on Computer Systems*, vol. 16, no. 2, pp. 133–169, may 1998.

[8] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM*, vol. 43, no. 2, pp. 225–267, march 1996.

[9] B. Charron-Bost, R. Guerraoui, and A. Schiper, "Synchronous system and perfect failure detector: solvability and efficiency issues," in *Proceedings of the International Conference on Dependable System and Networks*, june 2000, pp. 523–532.

[10] P. Veríssimo and A. Casimiro, "The timely computing base model and architecture," *IEEE Transactions on Computers*, vol. 51, no. 8, pp. 916–930, august 2002.

[11] S. Gorender and R. J. A. Macêdo, "A dynamically qos adaptable consensus and failure detector," in *Proceedings of The IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2002 - Fast Abstract Track*, june 2002, pp. B80–B81. An extended version has been published in the Proceedings of Brazilian Symposium on Computer Networks and Distributed Systems (SBRC2002), Pages 277–292.

[12] S. Gorender, R. J. A. Macêdo, and M. Raynal, "An adaptive programming model for fault-tolerant distributed computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 4, no. 1, pp. 18–31, january 2007.

[13] J. W. S. W. Liu, *Real-Time Systems.* Upper Saddle River, NJ, USA: Prentice Hall PTR, 2000.

[14] M. Larrea, "Understanding perfect failure detectors," in *PODC '02: Proceedings of the twenty-first Annual Symposium on Principles of Distributed Computing.* New York, NY, USA: ACM, 2002, pp. 257–257.

[15] A. Casimiro and P. Veríssimo, "Timing failure detection with a timely computing base," in *Third European Research Seminar on Advances in Distributed Systems*, april 1999.

[16] C. Fetzer, "Perfect failure detection in timed asynchronous systems," *IEEE Transactions on Computers*, vol. 52, no. 2, pp. 99–112, 2003.

[17] R. J. A. Macêdo and S. Gorender, "Detectores perfeitos em sistemas distribuídos não síncronos," in *IX Workshop de Teste e Tolerância a Falhas (WTF 2008), Rio de Janeiro, Brazil*, May 2008.

[18] J. Helary, M. Hurfin, A. Mostefaoui, M. Raynal, and F. Tronel, "Computing global functions in asynchronous distributed systems with perfect failure detectors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 11, no. 9, pp. 897–909, 2000.

[19] N. Schiper and F. Pedone, "Solving atomic multicast when groups crash," in *12th International Conference on Principles Of Distributed Systems (OPODIS'2008)*, December 2008.

[20] J. P. Lehoczky, L. Sha, and J. Strosnider, "Enhanced aperiodic responsiveness in hard real-time environment," in *Proceedings of the 8th IEEE Real-Time Systems Symposium.* San Jose, California: IEEE, december 1987, pp. 110–123.

[21] T. M. Ghazalie and T. P. Baker, "Aperiodic servers in a deadline scheduling environment," in *Real-time Systems*, 1995, pp. 31–67.

[22] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss, "An architecture for differentiated services," *RFC 2475*, december 1998.