How Bad Are Wrong Suspicions?
Towards Adaptive Distributed Protocols

Lívia M. R. Sampaio, Francisco V. Brasileiro, Walfredo Cirne, Jorge C. A. Figueiredo
Universidade Federal de Campina Grande
Departamento de Sistemas e Computação
Av. Aprígio Veloso, 882 - 58.109-970, Campina Grande, PB, Brazil
Phone: +55 83 310 1433 Fax: +55 83 310 1365 {livia,fubica,walfredo,abrantes}@dsc.ufcg.edu.br

Abstract

In this paper, we analyze the performance of consensus protocols based on the rotating coordinator paradigm. We consider a simulated production environment, on which processing and communication resources available for the different processes running the protocols are not necessarily the same. Firstly, we show that, in some scenarios, the performance of the consensus protocol is enhanced when there is an increase in the number and duration of the wrong suspicions periods of the failure detection service used. Since it is well known that wrong suspicions may also decrease the performance of the consensus protocol, a new dilemma is posed to the designers of such protocols. We then propose a new approach to address performance issues in the design of crash-detection based distributed protocols for asynchronous systems. We argue that they must be designed to adapt themselves to the variations on the availability of resources. The concept of slowness oracles is proposed to achieve this goal. Finally, we present a slowness oracle that can be used to transform a non-adaptive consensus protocol into an adaptive one. Simulations show that the adaptive protocol outperforms its conventional non-adaptive counterpart in a number of scenarios, having an equivalent performance in the other scenarios.

1. Introduction

An unreliable failure detector [1] is an important abstraction to design fault-tolerant distributed protocols that execute over an asynchronous infrastructure. They provide an elegant and modular way to encapsulate the extra synchrony required to circumvent the well known FLP impossibility result [6]. Further, some of the protocols built with the aid of this abstraction possess an important safety property, in the sense that even if the underlying failure detection service does not provide the service promised, this may only impact the termination of the protocol, and will never lead it to take wrong decisions [1].

Following the first theoretical results [1, 2], a number of implementations of unreliable failure detectors has been reported in the literature [5, 10, 11, 7, 12]. Also, to allow the comparison of the performance of different implementations of failure detectors, some quality of service (QoS) metrics have been proposed [3].

Currently, performance analysis of distributed fault-tolerant protocols that use a failure detection service is a research topic that is gaining considerable attention [14, 4, 8]. Most of the results presented so far, analyze how the QoS of the failure detection service, or the extra communication load generated by its implementation, impact the performance of consensus protocols. Further, they make the simplifying assumption that the failure detection service and the consensus protocols are the only computations in place, i.e. there are no other sources of contention for both processing and communication resources.

Like other papers previously published [14, 4, 8], we also analyze the performance of the classical consensus protocol based on the rotating coordinator paradigm, introduced in [1]. However, in this paper, we analyze its performance under a more common environment, where the processing power and network bandwidth available for the different processes running the protocol (and the underlying failure detection service required) are unknown a priori, and not necessarily the same for all processes. If the availability of resources at any given time were known a priori (or easily inferred), it would probably be a better approach to consider a synchronous system model, hence, we believe the assumption we make is more realistic.

We have started off our research by simulating, within this non-dedicated environment, the behavior of the consem-
sus protocol, using failure detection services with different levels of QoS. As will be presented in Section 3, these simulations revealed that better QoS for the underlying failure detection service does not necessarily imply better performance for the higher level consensus protocol. In particular, the consensus protocol can sometimes achieve better performance when the failure detection service makes more wrong suspicions. This is because the performance of this particular consensus protocol, is very much dependent on the responsiveness of the process that coordinates a decision round. When processing or communication resources available to this process are scarce, it will run slower, impacting the performance of the protocol. Therefore, in this situation, a performance gain might be achieved if a sufficient number of processes wrongly suspect a slow process that coordinates a round, and proceed to the next round, which could possibly be coordinated by a faster process. However, as previous studies show, wrong suspicions may also impact negatively the performance of the higher level protocol [4].

For instance, if we consider that not only the coordinator of the first round is running slow, but also a large number of other processes, wrongly suspecting slow processes will not help, and instead, will further reduce the performance of the protocol. These observations suggest that, in addition to the usual trade-off of avoiding wrong suspicions without increasing too much the detection latency of the service, appropriate tuning of the failure detection service requires the ability of allowing, under particular scenarios, slow processes to be wrongly suspected.

To address the first dilemma, a compromise may be achieved if the failure detection service is able to adapt itself to the changing conditions of the environment, increasing and decreasing timeouts appropriately [12]. However, adaptive failure detection services aim to restrict the number of wrong suspicions the service makes, and are not tailored to identify the application-dependent scenarios on which wrong suspicions should take place. We argue that this issue must be tackled by the higher level protocol, thus, in this case, adaptation must take place at the protocol level. The protocol has more knowledge, not only to precisely identify the cases where detecting slow processes would help to increase its performance, but also to take the necessary actions to improve its performance. Based on this argument, we advocate that implementations of failure detection services should strive to avoid wrong suspicions and leave the task of detecting slow processes to the higher level protocols.

The remaining of the paper is structured in the following way. In Section 2 we present our assumptions. We also give a brief review of the non-adaptive consensus protocol studied and the QoS metrics used to compare different failure detection services. In Section 3 we discuss the behavior of the consensus protocol for different resource availability scenarios and considering the use of failure detection services with different levels of QoS. As pointed out before, the simulation results attained prompted us to propose an alternative approach to separate efficiency concerns in the design of crash-detection based fault-tolerant distributed protocols for asynchronous systems [13]. This is done in Section 4, where the concept of slowness oracles is introduced. This section also presents a generic design for an adaptive consensus protocol, based on the definition of a slowness oracle for the conventional non-adaptive consensus protocol previously studied. In Section 5, a very simple implementation of this oracle is discussed, and performance analysis of the resulting adaptive protocol are presented. Finally, Section 6 brings our concluding remarks.

2. System Model and Definitions

The system model is patterned after the one described in [1, 6]. It consists of a finite set $\Pi$ of $n, n \geq 1$, processes, namely, $\Pi = \{p_1, \ldots, p_n\}$. A process can fail by crashing, i.e. by prematurely halting, and a crashed process does not recover. A process behaves correctly (i.e., according to its specification) until it (possibly) crashes. By definition, a correct process is a process that does not crash. A faulty process is a process that is not correct. At most $f$ processes may be faulty.

Processes communicate with each other by message passing through reliable communication channels: there are no message creation, alteration, duplication or loss. Processes are completely connected via unidirectional communication channels, i.e. $p_i$ sends messages to $p_j$ through the communication channel $c_{i,j}$, while $p_j$ sends messages to $p_i$ via $c_{j,i}$. Thus, a process $p_i$ may: 1) send a message to another process $p_j$ through $c_{i,j}$; 2) receive a message sent by another process $p_j$ through $c_{j,i}$; 3) perform some local computation; or 4) crash. There are assumptions neither on the relative speed of processes nor on message transfer delays. However, we consider that processes have access to a local clock device that can be used to measure the passage of time. Further, every process has access to a failure detection service.

The Consensus Problem In the consensus problem, every process $p_i$ proposes a value $v_i$ and all correct processes have to decide on some value $v$, in relation to the set of proposed values. More precisely, the consensus problem is defined by the following three properties [1, 6]:

---

1 So far, designers of failure detection services have concentrated their efforts in addressing the problem of appropriately tuning timeouts. For one side, the failure detection service must avoid, as much as possible, wrong suspicions. To achieve that, timeouts need to be set large enough to cope with periods of heavy system load. This, in turn increases detection latency, which also impacts performance when failures occur [4, 8, 13], thus constraining timeouts not to be set too large.
processes execute the following phases. In the first phase, the coordinator of each round. Within each round of the protocol, the identity of the process that plays the role of the coordinator is assumed that all processes have an a priori knowledge of brevity, we will name this protocol CT-consensus. CT-consensus uses a failure detection service of the $\diamond S$ class, and is based on the rotating coordinator paradigm. It requires that a majority of processes are correct, i.e., $n \geq 2f + 1$. A $\diamond S$ failure detection service is defined by the following completeness and accuracy properties [1]:

- **Strong completeness**: eventually every process that crashes is permanently suspected by every correct process; and
- **Eventual weak accuracy**: there is a time after which some correct process is never suspected by any correct process.

Several protocols to solve the consensus problem have been proposed. We focus our work on the consensus protocol presented by Chandra and Toueg [1] (for the sake of brevity, we will name this protocol CT-consensus). CT-consensus uses a failure detection service of the $\diamond S$ class, and is based on the rotating coordinator paradigm. It requires that a majority of processes are correct, i.e., $n \geq 2f + 1$. A $\diamond S$ failure detection service is defined by the following completeness and accuracy properties [1]:

- **Strong completeness**: eventually every process that crashes is permanently suspected by every correct process; and
- **Eventual weak accuracy**: there is a time after which some correct process is never suspected by any correct process.

The protocol is executed in asynchronous rounds. It is assumed that all processes have an a priori knowledge of the identity of the process that plays the role of the coordinator of each round. Within each round of the protocol, processes execute the following phases. In the first phase, every process sends its current estimation of the decision value to the round coordinator. In the second phase, the round coordinator gathers $[(n + 1)/2]$ estimations, chooses one of them, and sends it to all processes as its new proposition value. This choice must respect a locking mechanism that guarantees the agreement property of the consensus. It is based on the value of a time-stamp associated with every estimation message received. This time-stamp indicates the largest round on which a process has acknowledged a proposition. Upon receiving a majority of estimations, a coordinator must choose an estimation from those carrying the highest time-stamps. It also uses this value to update its current estimation value.

In phase three processes wait for the proposition from the round coordinator. To avoid the possibility of blocking due to a faulty coordinator, a process constantly queries its failure detection service to assess the round coordinator status. If the round coordinator is suspected, the process sends a nack message to the round coordinator (notice that a suspicion does not mean that the round coordinator has indeed failed, thus nacks are sent to prevent a wrongly suspected coordinator from blocking). On the other hand, if it receives the proposition value from the round coordinator, it adopts the proposition (by updating its current estimation with this value), updates its logical clock (used to generate time-stamps) with the number of the current round, and sends an ack message to the round coordinator. In the last phase the round coordinator collects $[(n + 1)/2]$ replies (acks and nacks), and if all replies are acks it decides for the proposition value it has proposed. The processes are informed of the decision via the execution of a reliable broadcast protocol [1]. A process finishes the execution of the protocol when it reliably delivers the decision value. Until a decision is not reached, a process that has finished its execution of round $k$, proceeds to execute round $k + 1$.

**Failure Detection QoS Metrics** We assume that the failure detection service is implemented by independent failure detection modules. Each process $p_i$ has access to $n - 1$ local failure detection modules, each monitoring one of the other $n - 1$ processes executing the consensus protocol. QoS metrics for the local failure detection module $FD_{i,j}$ of process $p_i$ that monitors process $p_j$ have been proposed in [3]. We consider the following three primary QoS metrics [3]:

- **Detection time** ($T_{D(i,j)}$): it is a random variable that represents the time that elapses since the crash of $p_j$ until it is permanently suspected by $FD_{i,j}$;
- **Mistake duration** ($T_{M(i,j)}$): it is a random variable that represents the length of a period during which $FD_{i,j}$ stays wrongly suspecting $p_j$; and
- **Mistake recurrence time** ($T_{MR(i,j)}$): it is a random variable that represents the time elapsed between the start time of two consecutive periods of wrong suspicions of $FD_{i,j}$.

**Failure Detection Service Definition** We define a particular failure detection service $FDS$ in terms of its completeness and accuracy properties, and the QoS provided by each of its constituent failure detection modules. Thus, a $FDS$ is a tuple:

$$FDS = (\text{completeness}, \text{accuracy}, QoS_{i,j})$$

where

$$\forall i,j, 1 \leq i,j \leq n, i \neq j$$

the QoS of $FD_{i,j}$, $QoS_{i,j}$, is given by

$$QoS_{i,j} = (T_{D(i,j)}, T_{M(i,j)}, T_{MR(i,j)})$$

For the sake of simplicity, we will restrict our performance evaluation of the consensus protocols for a setting where $n = 3$ and $f = 1$. Also, since we consider only $\diamond S$ failure detection services, we omit the completeness and accuracy properties of the service definition. Thus, the failure detection services used are defined in the following way:

$$FDS = (QoS_{1,2}, QoS_{1,3}, QoS_{2,1}, QoS_{2,3}, QoS_{3,1}, QoS_{3,2})$$
3. A Case for Wrong Suspicions

In this section we analyze the performance of the CT-consensus protocol under different resource availability scenarios, and several configurations for the underlying failure detection service.

We have used a hierarchical colored Petri Net (HCPN) model [9] to simulate the consensus protocol. The performance analysis of the consensus protocol was made with the support of the Design/CPN Tool and the Design/CPN Performance Tool².

The HCPN model of the consensus protocol consists of a collection of modules (pages), organized in a hierarchy. The top level of the model is the system module (System page) that is decomposed in three main parts: the consensus protocol (Process page), the failure detection service used by the consensus protocol (FailureDetectionService page), and the network that supports the protocol execution (CommunicationChannel page). The DS failure detection service is modeled in the FailureDetectionService page. The QoS of each service module \( F_D_{i,j} \) is represented as a tuple of elements, as previously described. Module state transitions depend on the values of the QoS provided by the module at each particular simulation period. In other words, each \( F_D_{i,j} \) module alternates between a period of time on which \( p_i \) suspect \( p_j \) and a period of time on which \( p_i \) does not suspect \( p_j \). The duration of each period \( (T_M) \) and the interval between two consecutive suspicion periods \( (T_MR) \) follow a normal distribution \( N(n, s) \) with mean \( n \) and variance \( s \). The network model represents the communication channels that interconnect the processes. Channels are represented by resources that must be obtained by a process to send messages through the network. Communication delays for a channel \( c_{i,j} \) follow a discrete uniform distribution \( U(l, u) \), with lower bound \( l \) and upper bound \( u \).

To simulate the variation on the availability of network and processing resources, we express contention on both types of resources as an increase on the communication delays. By introducing extra delays in the communication channels that convey messages to and from a process \( p_i \) (\( c_{x,i} \) and \( c_{i,x} \), respectively), it is possible to simulate different resource availability scenarios for any process \( p_i \).

In our work we have configured the behavior of the communication channels \( c_{i,j} \) using the same type of distribution used in [4]. We have measured the end-to-end transmission delays of 5 machines on our local area network and 2 machines on our campus network; the fastest machine pair had, in average, a transmission delay of 115 \( \mu s \), while the slowest machine pair was, in average, 5 times slower than the other pairs. We have considered two possibilities for a communication channel \( c_{i,j} \). In the first scenario (that of a “normal” channel), communication delays in \( c_{i,j} \) follow a discrete uniform distribution \( U(100, 130) \). In the other scenario (that of a “slow” channel) communication delays in \( c_{i,j} \) are 5 times slower, and follow a discrete uniform distribution \( U(500, 650) \).

We have also consider two types of \( F_D \) modules. These modules are configured using the values measured in [4]. Those values were measured using a push-style failure detection service implementation, on which heartbeat messages are sent periodically. The simulation model used in [4] is very similar to ours, therefore, we have used two configurations for the failure detection modules, for which there was a match between the values measured and simulated in [4]. We have named these types of modules “good” (G) and “bad” (B), respectively. The mean values for the maximum duration \( (T_{M}) \) random variable have been set so that the “good” module has the best value measured for a three-process system \( (7000 \mu s) \), while the “bad” module has the worst value measured \( (12000 \mu s) \). For the mean value of the mistake recurrence \( (T_{MR}) \) random variable we have used two values measured such that the mean for the “good” module is bigger than the mean for the “bad” module. In all cases the variance was set to approximately \( 30\% \) of the mean. Further, we have considered only the most frequent runs of the protocol where no failures occur, thus the value of detection time \( (T_D) \) is not relevant, and is not considered.

The two types of failure detection modules used are defined as:
\[
G = (T_M = N(7000, 2000), T_{MR} = N(190000, 60000)), B = (T_M = N(12000, 3600), T_{MR} = N(50000, 15000)).
\]

Based on these two module types, we have 64 different configurations for the failure detection service of a three-process system. Among these configurations, the best service possible is represented by:
\[
F_D_{S_{best}} = (QoS_{1,2} = G, QoS_{1,3} = G, QoS_{2,1} = G, QoS_{2,3} = G, QoS_{3,1} = G, QoS_{3,2} = G).
\]

Any other of the 63 remaining combinations, provide a service that is worse than that provided by \( F_D_{S_{best}} \). In [4], it is pointed out that, in case of contention on system resources, it is likely that some sort of correlation will occur between the modules that implement the failure detection service. Thus, in addition to \( F_D_{S_{best}} \), we have chosen three other failure detection services, with the following correlation patterns: i) all failure detection modules of the same process have the same QoS; ii) all failure detection modules that monitor a given process have the same QoS; and iii) the failure detection module \( F_D_{1,3} \) has the same QoS of \( F_D_{1,3} \). From all possible combinations of failure detection services with QoS worse than that of \( F_D_{S_{best}} \), we have chosen the following:

\[
F_D_{S_{process}} = (QoS_{1,2} = G, QoS_{1,3} = G, QoS_{2,1} = B, QoS_{2,3} = B, QoS_{3,1} = G, QoS_{3,2} = G).
\]

²These tools are available at http://www.daimi.au.dk/designCPN.
the system is not homogeneously loaded. In particular, CT-consensus vice on the performance of the
consensus execution is given by the smallest time any pro-

a slightly worse performance than that with
detection services that make more wrong suspicions have
atively low. It can be seen that the executions with failure
are “normal”, vices, and considering the first scenario, where all channels
CT-consensus using the four different failure detection ser-
protocol in study. We have simulated runs with
fact, this is an already known result [4]. The difference is
for the best configuration and each one of the FDS con-
fications presented above, we have simulated two differ-
ment scenarios: i) all communication channels are normal; and ii) \( p_1 \) is the coordinator of the first round of any exe-
duction of the consensus protocol, all channels \( c_{1,x} \) and \( c_{x,1} \)
are slow, and all other channels are normal.

The completion time of the consensus protocol is the
metric we have used to analyze the performance of the pro-
tocol in study. We have simulated runs with 1000 consec-
executions of the consensus protocol. All processes
start the first consensus at the same time, and consensus \( k \),
\( 1 < k \leq 1000 \), is started by a process \( p_k \), as soon as it
decides consensus \( k - 1 \). The completion time of the \( k^{th} \)
consensus execution is given by the smallest time any pro-
cess decides consensus \( k \).

Figure 1 shows the results obtained for the execution of
CT-consensus using the four different failure detection ser-
ices, and considering the first scenario, where all channels
are “normal”, i.e. the system load is homogeneous and rel-
atively low. It can be seen that the executions with failure
detection services that make more wrong suspicions have
a slightly worse performance than that with \( FDS_{best} \). In
fact, this is an already known result [4]. The difference is
not bigger, because the “not so good” failure detection ser-
ices are still quite good.

\[ FDS_{monitor} = \{ QoS_{1,2} = G, QoS_{1,3} = G, QoS_{2,1} = \]
\( B, QoS_{2,2} = G, QoS_{3,1} = B, QoS_{3,2} = G \} \); and
\[ FDS_{monitor\text{-irr}} = \{ QoS_{1,2} = B, QoS_{1,3} = G, QoS_{2,1} = \]
\( B, QoS_{2,2} = G, QoS_{3,1} = G, QoS_{3,2} = G \} \).

For the best config and each one of the FDS con-
fications presented above, we have simulated two differ-
ment scenarios: i) all communication channels are normal;
and ii) \( p_1 \) is the coordinator of the first round of any exe-
duction of the consensus protocol, all channels \( c_{1,x} \) and \( c_{x,1} \)
are slow, and all other channels are normal.

The completion time of the consensus protocol is the
metric we have used to analyze the performance of the pro-
tocol in study. We have simulated runs with 1000 consec-
executions of the consensus protocol. All processes
start the first consensus at the same time, and consensus \( k \),
\( 1 < k \leq 1000 \), is started by a process \( p_k \), as soon as it
decides consensus \( k - 1 \). The completion time of the \( k^{th} \)
consensus execution is given by the smallest time any pro-
cess decides consensus \( k \).

Figure 1 shows the results obtained for the execution of
CT-consensus using the four different failure detection ser-
ices, and considering the first scenario, where all channels
are “normal”, i.e. the system load is homogeneous and rel-
atively low. It can be seen that the executions with failure
detection services that make more wrong suspicions have
a slightly worse performance than that with \( FDS_{best} \). In
fact, this is an already known result [4]. The difference is
not bigger, because the “not so good” failure detection ser-
ices are still quite good.

4. Separating Concerns

4.1. Slowness Oracles

A crucial point in the design of failure detection services,
is the issue of tuning the timeouts that are used to iden-
tify faulty processes. Too small timeouts increase wrong
suspicions, which, as it was shown in Section 3 and else-
where [4], may harm the performance of the higher level
protocols. On the other hand, too large timeouts increase
the detection latency of the service, imposing heavy over-
heads to those runs on which failures occur [4, 8, 13]. Thus,
difficult trade-off issues have to be analyzed in order to ac-
commodate the conflicting requirements of avoiding wrong
suspicions and, at the same time, minimizing detection la-
tency. A large part of the work on failure detection ser-
ices developed so far has been dedicated to address this
dilemma.

The analysis presented in Section 3 has also highlighted
a new issue: for the CT-consensus protocol, when the per-
formance of the round coordinator is substantially worse
than that of a majority of the other processes, wrong sus-
picions have a positive impact in the performance of the
protocol. The new problem to be faced is how to differenti-
ate “good” wrong suspicions (those that improve the perfor-
ance of the protocol) from “bad” wrong suspicions (those

![Figure 1. CT-consensus completion time for 1000 consensus and all channels being "normal"

Figure 2 shows the influence of the failure detection ser-
vice on the performance of the CT-consensus protocol when
the system is not homogeneously loaded. In particular,
when the channels from/to the coordinator of the first round
of the consensus are “slow” and all other channels are “nor-
mal”. In this case, the performance of CT-consensus is bet-
ter when the failure detection service makes more wrong
suspicions. Notice that, in this scenario, even a small de-
crease in the QoS of the failure detection service is enough
to improve the performance of the consensus. For example,
the completion time for the execution using \( FDS_{monitor} \)
was about 13.5% smaller than that using \( FDS_{best} \).

![Figure 2. CT-consensus completion time for 1000 consensus and coordinator channels being "slow"

Figure 2 shows the influence of the failure detection ser-
vice on the performance of the CT-consensus protocol when
the system is not homogeneously loaded. In particular,
when the channels from/to the coordinator of the first round
of the consensus are “slow” and all other channels are “nor-
mal”. In this case, the performance of CT-consensus is bet-
ter when the failure detection service makes more wrong
suspicions. Notice that, in this scenario, even a small de-
crease in the QoS of the failure detection service is enough
to improve the performance of the consensus. For example,
the completion time for the execution using \( FDS_{monitor} \)
was about 13.5% smaller than that using \( FDS_{best} \).
that reduce the performance of the protocol). One could argue that this issue could be addressed in a straightforward way, by considering not only crash failures, but also performance failures. In this way, one would only have “bad” wrong suspicions, since “good” wrong suspicions would, in fact, be correct suspicions (of processes that have suffered performance failures). However, it is important to notice that, for the protocol studied, it is not enough to detect that a particular process is slow, but rather that it is slower than the others.

We argue that the issue of system wide slowness is very much dependent on the design of the higher level protocol that uses the failure detection service. For the consensus protocol studied, we need to analyze how slower a coordinator is in relation to the other processes, while for other crash-detection based protocols a completely different requirement may arise. Thus, we advocate that, instead of trying to introduce this functionality into the failure detection service (a task that will only make the implementation of efficient failure detection services more difficult), one should rather modify the higher level protocols, in such a way that they can adapt themselves to the changing conditions of the environment.

On the other hand, we would not like to loose all the good properties (e.g. easiness for proving correctness, safety, modularity) of the protocols that have already been designed using the abstraction of unreliable failure detection services. We believe that most crash-detection based protocols that have already been designed, can be transformed in adaptive protocols by the introduction of a suitable slowness oracle that is able to identify the particular situations where the environment conditions are not favorable to the execution of the protocol, and then, trigger an adaptation. In the next subsection we present one such oracle for the non-adaptive consensus protocol earlier analyzed.

4.2. Adaptive CT-consensus Protocols

A fundamental requirement that guarantees the correctness of the CT-consensus protocol is the ability of the involved processes to agree on the identity of the coordinator of any round of any execution of the protocol [1]. This is achieved through an a priori agreement on the identity of the coordinator of the first round of all consensus executions. Subsequent rounds of a particular consensus execution follow a round-robin schedule on the ordered list of processes to define the identity of the coordinator of each round.

The performance evaluation of the CT-consensus protocol presented in Section 3, has stressed the influence that the speed (in terms of both processing and communication latency) of the first coordinator of a consensus execution has in the overall performance of the protocol. The coordinator selection mechanism used by this protocol, albeit extremely simple, does not allow for adaptation to take place, since the identities of round coordinators are selected before the execution of the protocol has even started. Adaptation can only occur if the selection of the coordinator of the first round of an execution of the protocol takes into account the environment conditions during the execution of the protocol.

A perfect selection mechanism should always output the identity of the process that would be the most responsive in the next execution of the protocol. It is clear that such a mechanism is impossible to be built, unless the future behavior of the system is known a priori. Thus, any practical coordinator selection mechanism for an adaptive protocol must be best-effort. Nevertheless, assuming that most of the time the recent past is a good indicator of the near future (which is true for real systems), it is possible to build a selection mechanism that most of the time outputs a suitable coordinator. Thus, a possible slowness oracle for the CT-consensus protocol, would be one that could order processes by their current perceived responsiveness, allowing the fastest one to be always chosen to coordinate the first round of the protocol. An adaptive protocol could then be designed by, somehow, consistently using this information to choose the first coordinator of a consensus execution. More formally, an adaptive consensus protocol, based on CT-consensus, can be designed by providing each process $p_i$ with a slowness oracle $SO$ that provides the following properties:

- **Termination**: when a correct process queries $SO$ for the identity of the coordinator of the first round of the $k^{th}$ execution of the consensus, $k > 0$, eventually $SO$ returns a process identity;
- **Validity**: $SO$ always returns the identity of a process that belongs to the set $I$;
- **Agreement**: no two correct processes receive a different identity when they query $SO$ for the same execution of the consensus; and
- **Adaptability**: $SO$ must use new knowledge gathered during its execution, in order to choose a suitable round coordinator.

In the adaptive version of the CT-consensus protocol, every $p_i$ queries the slowness oracle $SO$, at the beginning of each consensus execution, to get the identity of the coordinator of the first round of the protocol. After that, the execution of the consensus protocol proceeds as in CT-consensus.

As will be seen in Section 5, to avoid extra overhead introduced by the implementation of the slowness oracle, small modifications to the original consensus protocol are required. This involves piggybacking resource availability information in the original messages of the protocol, and the introduction of watchdog timers to measure timeouts.
The slowness oracle encapsulates application dependent system feedback. For the adaptive CT-consensus protocol, the slowness oracle is an atomic coordinator selector which guarantees that every correct process will choose the same coordinator for the first round (and, therefore, for any subsequent round) of every execution of the protocol. The first three properties of the slowness oracle provide atomicity. They guarantee the correctness of the consensus protocol. Note that the adaptability property does not impact the correctness of the consensus protocol. Rather, it only guarantees that system feedback is taken into account, thus allowing adaptation to take place.

4.2.1. Proof of Correctness

Theorem 1  The consensus problem is solved by the adaptive CT-consensus protocol that uses a slowness oracle SO.

Proof The proof is straightforward. The only difference between the adaptive and the non-adaptive versions of the CT-consensus protocol is in the way that the former chooses the coordinator of the first round of a particular execution of the protocol. Therefore, it suffices to prove that all processes choose the same coordinator for the first round of every consensus execution. The validity property of SO guarantees that the coordinator of the first round is a valid process; the termination and agreement properties of SO guarantee that every process will choose the same coordinator for the first round (and therefore, of any subsequent round) of any execution of the protocol.

5. An Implementation of an Adaptive CT-consensus Protocol

5.1  #1-A-CT-consensus

We now present the implementation of a very simple slowness oracle SO to be used in the implementation of an adaptive CT-consensus protocol. We have named the oracle #1-SO, and the resulting protocol #1-A-CT-consensus.

As discussed before, the choice of the coordinator by #1-SO must be atomic and use system feedback. Our implementation of #1-SO achieves atomicity with the support of the consensus protocol with which it is associated. For the first execution of #1-A-CT-consensus there is an a priori agreed coordinator. For the $i$th execution of the protocol, the coordinator is chosen by applying a deterministic function on slowness information that is locally gathered at execution time by each process during previous runs and atomically disseminated by the $(i-1)$th execution of the consensus protocol.

In the #1-A-CT-consensus protocol, each process maintains three $n \times n$ matrices that it uses to store system slowness information. Line $i$ of each matrix stores information on $p_i$’s view of the system. Each cell $[i,j]$ of a matrix, stores a boolean value that indicates whether $p_i$ believes that $p_j$ is slow (when the value is equal to true) or not (when the value is equal to false). Initially, all cells, in all matrices, are set to false.

The first matrix, named $local_i$, is used by a process $p_i$ to store information inferred from the piggybacked information it receives from the other processes, and from local timers it manages. The second one, named $global_i$, represents the current consistent global view that all correct processes have. The last matrix, named $current\_estimate_i$, is used to store $p_i$’s current estimation value for the new global view that is established at the end of the execution of the current consensus. In #1-A-CT-consensus, the estimation values exchanged and the decisions reached by the processes carry both an application value and a slowness information matrix. The $global_i$ matrix is atomically updated via the execution of a consensus over the values of the $current\_estimate_i$ matrices of all processes.

Every time process $p_i$ sends its estimation value to the round coordinator (in the first phase of CT-consensus, see Section 2), it piggybacks in this message, line $i$ of its $local_i$ matrix, and its $current\_estimate_i$ matrix. Then, it starts a watchdog timer. Whenever $p_i$ receives a proposition from the round coordinator (phase three of CT-consensus), it resets the watchdog timer for that consensus round (if it has not yet fired) and sets $local_i[i,c] = false$, with $p_c$ being the round coordinator. $p_i$ also updates its $local_i$ matrix by simply updating the $c$th line of $local_i$ with the value $p_i$ has piggybacked on its proposition message (line $c$ of its $local_i$ matrix). When a watchdog timer fires, $p_i$ sets $local_i[i,c] = true$. The round coordinator $p_c$ updates its $local_i$ matrix, every time it receives an estimation message from a process $p_i$ (phase two of CT-consensus). In this case, the $i$th line of $local_i$ is updated with the value $p_i$ has piggybacked on its estimation message.

At the beginning of every consensus execution, $p_i$ initializes its $current\_estimate_i$ matrix with the value of its $local_i$ matrix. Further, the $current\_estimate_i$ matrix is updated every time the current estimation for the consensus value is updated (see Section 2). More precisely, in phase two of the protocol, the round coordinator $p_c$ gathers $[(n+1)/2]$ estimations (each containing an application value and a system slowness information matrix - the $current\_estimate_i$ matrix) and chooses one of them. This choice uses the same locking mechanism earlier discussed. The selected value is used to update both the current estimation for the consensus decision, as well as the $current\_estimate_i$ matrix. A process $p_i$ that receives a proposition message from $p_c$, updates its $current\_estimate_i$ matrix with the value piggybacked by $p_c$ in the proposition message.

Process $p_i$ has access to the service of the slowness ora-
Theorem 2 The #1-A-CT-consensus protocol solves consensus.
Proof From Theorem 1, any adaptive CT-consensus protocol solves consensus, provided it has access to a slowness oracle that possesses the properties presented in Section 4.2. Thus, it suffices to prove that the slowness oracle #1-SO, used in the #1-A-CT-consensus protocol, provides these properties. The proof follows directly from lemmas 1, 2, 3 and 4.

5.2. Performance Evaluation

We have modified the HCPN model presented in Section 3 to simulate the #1-A-CT-consensus protocol, and have conducted some experiments. Again, we have only considered failure-free runs, since these are the most frequent runs. Notice that since we have completely decoupled failure detection issues from system wide slowness adaptability, we expect failures and wrong suspicions to impact the performance of both adaptive and non-adaptive versions of
the consensus protocol in similar ways. Figure 3 compares the completion time of the CT-consensus and the #1-A-CT-consensus protocols using the same failure detection service (FDs, Cx) in two different scenarios: i) all communication channels are “normal”; and ii) all channels from/to the coordinator of the first round are “slow” and all other channels are “normal”. For the first scenario, the performance of both protocols are very similar. On the other hand, considering the second scenario, the completion time of the CT-consensus for 1000 consensus is 3.2 times bigger than that of the #1-A-CT-consensus. Moreover, while the #1-A-CT-consensus protocol has almost the same completion time for 1000 consensus in both situations (only 7% bigger), the CT-consensus protocol has a completion time that is 240% bigger, when comparing the two scenarios.

The results illustrated in Figure 3 allows the evaluation of the influence of system feedback on the performance of the adaptive protocol. Such results suggest that the use of system feedback in the design of fault-tolerant distributed protocols may have an important impact on their performance.

We have also considered an experiment, on which resource availability varied along the execution of the protocol. For this experiment, communications delays experienced by any channel $c_{i,j}$ were defined by a bi-modal uniform distribution. The channel alternates between periods on which it behaves as a “normal” channel and periods on which it behaves as a “slow” channel. There is a minimal duration for each such interval. Further, the channel has a configured probability of behaving as a “normal” one. In order to analyze with more confidence the influence of system feedback on the performance of the consensus protocols in study, we have eliminated any possible interference of the failure detection service. This is achieved by forcing the failure detection service used to behave as a perfect service during the whole execution of the experiments.

Figure 4 shows the completion time for 1000 executions of both CT-consensus and #1-A-CT-consensus for the following configurations: i) 90% probability of the channel being “normal”; and ii) 80% probability of the channel being “normal”. In both scenarios the minimal duration of the periods was 50ms.

In both cases the performance of #1-A-CT-consensus was better. For the more homogeneous workload (90% of probability of a channel being in the “normal” mode), #1-A-CT-consensus outperforms CT-consensus by 11% when we consider the completion time for 1000 consensus, while for a less homogeneous workload (80% of probability of a channel being in the “normal” mode), the performance of #1-A-CT-consensus is 17% better than that of CT-consensus, also considering the completion time of 1000 consensus.

6. Concluding Remarks

After all, how bad are wrong suspicions? We have found that, for the particular consensus protocol studied, they are actually bad. We have first found an example that suggested they would be good in some cases. However, looking closer, these good scenarios appeared only because of a collateral effect caused by the erroneous output provided by the failure detection service. In these scenarios, wrongly suspecting slow processes allowed adaptation to take place. We have proposed to equip existing non-adaptive protocols with slowness oracles that are able to identify the situations on which adaptation can bring an increase in the performance of the higher level protocol. A slowness oracle for the non-adaptive consensus protocol studied was presented, and performance analysis was conducted. The results obtained showed that the adaptive protocol has a better performance than the non-adaptive protocol in many scenarios.
Further, in all scenarios, the adaptive protocol is at least as good as the non-adaptive one.

Previous studies on the performance of agreement protocols have been done [14, 4, 8]. They analyze several factors that impact the performance of these protocols. [14] discusses the impact of the resource contention associated to different implementations of failure detection services on the performance of the CT-consensus protocol. [4] analyzes the impact of failures as well as the QoS of the failure detection service on the performance of CT-consensus. It does so by simulating and measuring an implementation of the protocol considering three classes of runs: i) no failures, and no wrong suspicions; ii) failures, and no wrong suspicions; and iii) no failures, but wrong suspicions. [8] also assesses the impact of failures on the performance of consensus protocols. It compares the performance of CT-consensus and another consensus protocol assuming a failure detection service that makes no wrong suspicions, and considering both failure-free runs and runs on which failures occur.

Differently from the aforementioned works, we have analyzed the performance of consensus protocols on an environment where the sources of processing and network contention are not limited to those associated to the protocol itself and to the underlying failure detection service it uses, allowing the availability of resources to vary during the execution of the protocol. Further, to the best of our knowledge, no previous work considers the possibility of improving the performance of distributed protocols by allowing them to adapt themselves to the changing availability of resources, as we have presented in this paper. It is important to point out that variations in the availability of resources are very frequent in practice, and our simulations have shown that these variations can have an important impact on the performance of distributed protocols. Hence, providing a protocol with the ability to adapt itself to the changing conditions of the environment is an important issue.

We are continuing our research on three main directions. Firstly, we are validating the simulation results presented, the implementation of adaptive consensus protocols, and the evaluation of these implementations under a production environment. Secondly, we are experimenting the design of more elaborate slowness oracles for the CT-consensus protocol. Finally, further research is being dedicated to the design of other adaptive crash-detection based distributed protocols using the concept of slowness oracles.

Acknowledgments Authors would like to thank the comments from the anonymous referees and the support from CAPES/Brazil and CNPq/Brazil (grants 478.752/01 and 300.646/96).

References