Efficient and robust adaptive consensus services based on oracles

Lívia Sampaio* **and Francisco Brasileiro****

*Coordenação de Pós-Graduação
em Engenharia Elétrica
**Coordenação de Pós-Graduação
em Informática
Universidade Federal de Campina Grande
58109-970 - Campina Grande - PB - Brazil
{livia — fubica}@dsc.ufcg.edu.br

Raul Ceretta Nunes

Departamento de Eletrônica
e Computação
Universidade Federal
de Santa maria
97105-900 - Santa Maria
RS - Brazil

ceretta@inf.ufsm.br

Ingrid Jansch-Pôrto

Instituto de Informática
Universade Federal
do Rio Grande do Sul
91501-970 - Porto Alegre
RS - Brazil

ingrid@inf.ufrgs.br

Abstract

Due to their fundamental role in the design of fault-tolerant distributed systems, consensus protocols have been widely studied. Most of the research in this area has focused on providing ways for circumventing the impossibility of reaching consensus on a purely asynchronous system subject to failures. Of particular interest are the indulgent consensus protocols based upon weak failure detection oracles. Following the first works that were more concerned with the correctness of such protocols, performance issues related to them are now a topic that has gained considerable attention. In particular, a few studies have been conducted to analyze the impact that the quality of service of the underlying failure detection oracle has on the performance of consensus protocols. To achieve better performance, adaptive failure detectors have been proposed. Also, slowness oracles have been proposed to allow consensus protocols to adapt themselves to the changing conditions of the environment, enhancing their performance when there are substantial changes on the load to which the system is exposed. In this paper we further investigate the use of these oracles to design efficient consensus services. In particular, we provide efficient and robust implementations of slowness oracles based on techniques that have been previously used to implement adaptive failure detection oracles. Our experiments on a wide-area distributed system show that by using a slowness oracle that is well matched with a failure detection oracle, one can achieve performance as much as 53.5% better than the alternative that does not use a slowness oracle.

***Keywords**:* consensus protocols; asynchronous distributed systems; adaptive protocols; slowness oracles; unreliable failure detectors; predictors.

1 Introduction

Consensus protocols play a key role in the design of fault-tolerant distributed systems, as they form the basis for several important distributed services, such as atomic broadcast, group membership, atomic commitment, among others [20, 1, 8]. Informally, a consensus protocol allows a set of n processes (among which up to f may crash) to reach agreement on a common value (*eg.* the total order for message delivery, the set of members to exclude or include in a group, whether a transaction should be committed or not).

Despite its apparent simplicity, it has been shown that this problem is impossible to be solved in a purely asynchronous system, even if as little as a single process may crash [6]. In fact, most of the work that has been developed in this area has attempted to provide ways for circumventing this impossibility result. One possible approach consists in considering that the distributed system possesses some extra level of synchrony sufficient to allow deterministic solutions to the problem. The most popular representative of this approach is based on equipping the (otherwise) purely asynchronous system with unreliable failure detection oracles [1]. Such oracles output the set of processes that they currently *suspect* to have crashed (or, alternatively, a set of processes that are currently *trusted* to be correct).

Different classes of failure detectors have been defined. They differ in the set of properties that the members of the class must provide. It has been shown that the $\Diamond S$ class is the weakest class of failure detectors that allows a deterministic solution to the consensus problem in asynchronous systems subject to failures [2]. Thus, it is not surprising that most of the consensus protocols based on

failure detectors that have been proposed so far use a $\diamond S$[1] failure detector [1, 18, 12, 13, 7].

Most consensus protocols based on a $\diamond S$ failure detector are asymmetric, in the sense that different participants can assume different roles during the execution of the protocol [1, 18, 12, 13, 7]. Such protocols are structured as a sequence of asynchronous communication rounds. In a given round, processes may normally assume one of two possible roles: a 'special' role, in which the process behaves as a coordinator[2] of the round, and a 'normal' role, in which the process simply cooperates with the round coordinator. These protocols must guarantee *liveness* and *safety*, *i.e.*, the protocol eventually terminates and processes engaged in the consensus never decide different values. Liveness is achieved by using a failure detector to prevent processes from blocking forever, waiting for an action from a crashed coordinator. In this case, another coordinator is chosen. To guarantee safety, *i.e.*, that a decision taken by a process in the earliest round that yields a decision is the same that will be taken by another process which, for some reason, is only able to decide in a future round, these protocols require a majority of correct processes, *i.e.*, $n > 2f+1$, as well as some kind of *locking* mechanism. This fact provides these protocols with a very important characteristic: they are *indulgent* [7] in relation to the failure detector that they use, *i.e.*, even if the failure detector misbehaves, the safety properties of consensus are still preserved.

On the other hand, the asymmetric structure of these protocols has a performance pitfall, specially when processes and communication channels are subject to considerable variability in load. For instance, this is the case when the protocol is executed over an open system whose communication is supported by wide-area channels. This is a common scenario for many applications, including the emerging grid middleware infra-structure (see for example the scheduler service of the PARADIS system [10]). In this scenario, if the coordinator of the first round is correct but runs slow (because either its processor or its communication channels - or both - are overloaded), the performance of the protocol may suffer [17]. Sampaio *et al.* [17] have proposed the use of slowness oracles to tackle this problem.

A slowness oracle collects system wide information about the responsiveness of processes and outputs the (unique) identity of the process that it judges to be the most responsive at the moment. The consensus protocol that uses a slowness oracle chooses the coordinator of the first round by querying the oracle, instead of relying on an a priori agreement (that could result in choosing a slow coordinator). To guarantee agreement, it takes advantage of the fact that, in many practical systems, consensus is provided as a service that is invoked by its clients many times in succession (for instance, in a replicated server, consensus is invoked every time a new request arrives, while in the replicated scheduler of PARADIS it is invoked whenever a new job/task submission is received). In this way, the process identity to be output in the k^{th} invocation of the slowness oracle ($k > 1$) is agreed upon during the execution of the $(k - 1)^{th}$ consensus[3].

In the very simple implementation of a slowness oracle presented in [17], processes are classified as slow or fast based on the round trip delays of the messages they exchange during the past executions of consensus. There is a fixed threshold that is used to identify slow/fast processes. The most responsive process is the one that is seen as fast by the largest number of processes (with the ordered identities of the processes being used to break ties). Moreover, to better account the impact of adaptation due to the use of the slowness oracle, it is assumed that the failure detector makes no mistake. In fact, in the simulations conducted, the failure detector module was replaced by a mock implementation that always returned an empty set of suspected processes, obviating the need for exchanging failure detection messages. Nevertheless, the simulation results presented in [17] show that even this very naive implementation of a slowness oracle is able to increase the performance of consensus in as much as 17% for a lightly loaded system.

Contributions

In this paper we further investigate the use of slowness oracles in the design of adaptive consensus services. In particular, we study more elaborated implementations of such mechanism. Firstly, we address the issue of better estimating the process that is the most responsive. The implementation discussed above relies on a fixed threshold to identify slow/fast processes. Thus, setting this threshold is a key deployment decision. Too small thresholds will mask fast processes and trigger unnecessary adaptation, while too large thresholds will mask slow ones and prevent a required adaptation. This problem is closely related to that of dynamically tuning timeouts of failure detectors. In this paper we apply techniques used to implement adaptive failure detectors to achieve efficient and robust implementations of slowness oracles, in the sense that they allow adaptation to occur without requiring any difficult deployment decisions to be taken. Secondly, we tackle the issue of efficiently accommodating failure detection and slowness oracles in the design of a consensus service. In particular, we propose an architecture for the design of oracle-based consensus services. Finally, we analyse the performance of adaptive and non-adaptive implementations of consensus services, using both simulations and experiments on a real

1 Or a failure detector that is equivalent to a $\diamond S$ failure detector such as $\diamond W$ or Ω [2].

2 We will use the term *coordinator* to refer to the process that plays this special role; we note, however, that sometimes it has been referred by other terms in the literature (*eg. leader*).

3 Obviously, agreement on the first invocation of the slowness oracle ($k = 1$) must be achieved a priori.

system. Our experiments on a real system show that by using a slowness oracle that is well matched with a failure detection oracle, one can achieve performance as much as 53.5% better than the corresponding alternative that does not use a slowness oracle.

Road map

The rest of the paper is structured in the following way. Related work is discussed in Section2. Section 3 presents the system model and some definitions. Then, in Section 4 we present the design of slowness oracles to support adaptive consensus protocols. In Section 5 we investigate, through simulation, the performance of a consensus service based on the protocols presented in Section 4. Section 6 discusses how to better integrate failure detection and slowness oracles when designing a consensus service. Performance analysis of the non-adaptive and adaptive versions of the service within a real system is conducted in Section 7. Finally, Section 8 concludes the paper.

2 Related Work

Following the first theoretical results that aimed at proving the correctness of consensus protocols based on unreliable failure detectors, performance analysis of such protocols is now a research topic that has gained considerable attention [19, 4, 11, 23].

In [23], Urbán *et al.* compare the performance of two indulgent consensus protocols based on distinct oracles. The results show that the protocol that uses a leader oracle only outperforms the other protocol based on a $\diamond S$ failure detector in scenarios where either the failure detector often makes mistakes or correlated crashes occur. In the other scenarios, both protocols have a similar performance. Since in a wide area setting correlated crashes are very unlikely, and failure detection oracles can be designed to provide very good quality of service (QoS), in practice, both types of oracles lead to similar performance results. In this paper we only consider protocols based on $\diamond S$ failure detectors.

When considering performance of consensus protocols that use a $\diamond S$ failure detector, the majority of the published research analyze how the QoS of the failure detection service, or the extra communication load generated by its implementation, impact the performance of the protocol [19, 4]. $\diamond S$ failure detectors are usually implemented by using timeouts to identify faulty processes. Therefore, the issue of tuning these timeouts is a crucial point in the design of an efficient failure detector. Too small timeouts increase wrong suspicions, which, may harm the performance of the higher level consensus protocol [17]. On the other hand, too large timeouts increase the detection latency of the service, imposing heavy overheads to those runs on which failures occur [4, 23]. Thus, difficult trade-off issues have to be analyzed in order to accommodate the conflicting requirements of avoiding wrong suspicions and, at the same time, minimizing detection latency. A large part of the work on failure detectors developed so far has been dedicated to address this dilemma. As such, different adaptive failure detectors have been proposed [3, 16]. Nunes and Jansch-Pôrto [14, 15, 16] compare different ways for predicting the round trip delays of messages on a wide area network. When combined with appropriate safety margins, these predictors can be used to implement efficient adaptive pull-style failure detection oracles [16]. In this work we study adaptation at the consensus protocol level. In particular, we study the use of predictors to implement efficient slowness oracles and how to conveniently combine them with predictor-based adaptive failure detection oracles.

3 System Model and Definitions

The system model is patterned after the one described in [6, 1]. It consists of a finite ordered set Π of n, $n > 1$, 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, $f < \lceil (n + 1)/2 \rceil$, processes may crash.

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 no assumptions on the relative speed of processes nor on message transfer delays.

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 *uniform consensus problem* is defined by the following three properties [6, 1]:

- **Termination**: every correct process eventually decides some value;
- **Validity**: if a process decides v, then v was proposed by some process; and
- **Uniform agreement**: no two processes (correct or not) decide differently.

To allow a deterministic solution to the consensus problem, we enhance the system with an unreliable failure detector [1]. In particular, we consider consensus protocols that are supported by a failure detection oracle of the class $\diamond S$. A $\diamond S$ failure detection oracle is formally defined by the following completeness and accuracy properties [1]:

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

4 Efficient and Robust Slowness Oracle

Our goal is to build a distributed consensus service that can be concurrently used by several distributed applications. To use the service each process running the application must propose a value to its local consensus module. Thus, over time, a sequence of consensus protocols is executed. It is possible to optimize the service by allowing a single execution of the consensus protocol to carry propositions from several distinct applications [9].

This section is structured in the following way. We start by sketching the structure of a classical non-adaptive consensus protocol and showing how this protocol can be made adaptive by enhancing it with a very simple slowness oracle. Then, we present the design of an efficient and robust slowness oracle. Note that, the consensus protocol described below is used as an example of adaptive distributed protocol based on slowness oracle. We believe that such an oracle can be applied to many other asymmetric distributed protocol.

4.1 A Classical Non-Adaptive Consensus Protocol

Several protocols to solve the consensus problem have been proposed. We focus our work on the classical 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.

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 (for instance, by using a round-robin sequence on the ordered identities of processes, starting from the p_1). 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 $\lceil (n+1)/2 \rceil$ 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. When different from 0, 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-stamp. 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 forever 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 forever). 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 $\lceil (n+1)/2 \rceil$ replies (*acks* and *nacks*), and if all replies are *acks* it initiates the reliable broadcast [1] of the decision value it had proposed. A process finishes the execution of the protocol when it reliably delivers the decision value that has been reliably broadcast by a successful coordinator. Until a decision is not reached, a process that has finished its execution of round k, proceeds to execute round $k + 1$.

4.2 Adaptation via Slowness Oracles

Slowness oracles were defined in [17] as an abstraction to construct adaptive distributed protocols. A slowness oracle is able to identify the particular situations in which the environment conditions are not favorable to the execution of the protocol, and then, trigger an adaptation. For the consensus protocol presented in the previous section, a slowness oracle *SO* must provide the following properties [17]:

- **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 II;
- **Agreement**: no two correct processes receive a different identity when they query *SO* for the same execution of the consensus.

In this case, 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 [17]. Note that, in order to allow consensus protocols to adapt itself to the changing conditions of the execution environment, SO must use new knowledge (system feedback) gathered during its execution to choose a suitable round coordinator.

A very simple implementation of a slowness oracle has been proposed in [17] (let us name the resulting adaptive consensus protocol by *ACT-consensus*). This implementation aimed to prove the concept of slowness oracles. To reduce overheads to a minimum, it is tightly coupled to

the implementation of the *CT-consensus* protocol. The slowness information collected is the round trip delays in the communication steps of the consensus executions. Thus, a round coordinator is able to collect delays with respect to at least a majority of processes, while the other processes are able to collect delays with respect to the round coordinator (see Section 4.1). A fixed predefined threshold is used to classify a process in either fast or slow. Local slowness knowledge is disseminated without extra messages with the aid of the consensus protocol messages (by piggybacking the required information). Each process p_i consolidates the (local and remote) slowness information gathered in a local slowness matrix in which entry [*j, k*] is set to 'fast' if p_j thinks that p_k is running fast, and to 'slow' otherwise. A common global slowness matrix (named *global matrix*) is attained by using the consensus protocol itself to achieve agreement, *i.e.*, the global matrix stored at the beginning of the k^{th} execution of consensus protocol is provided together with the decision value of the $(k - 1)^{th}$ execution of the consensus protocol (all processes start with the same global matrix that they use to chose the coordinator of the first execution of the consensus protocol). An a priori agreed deterministic function is then applied to the common global matrix to chose the appropriate coordinator. The following algorithm can be used to choose a coordinator. First, a process p_i constructs, for all processes p_j, $p_j \in \Pi$, the ordered sets *fast*(*j*) containing the identity of the processes that believe p_j is 'fast' at the moment. The ordered sets *fast*(*j*) are given by:

$$fast(j) = \{p_k \in \Pi : global\_matrix[k, j] = \text{'fast'}\}.$$

Then, p_i builds the following two subsets of Π:

$$Class_1 = \{p_j \in \Pi : \mid fast(j) \mid = n\}; \text{ and}$$
$$Class_2 = \{p_j \in \Pi : \mid fast(j) \mid \geq \lceil (n+1)/2 \rceil\}.$$

The coordinator is the process p_c with the smallest identity from the first non-empty set in the sequence $Class_1$, $Class_2$, and Π. In other words, the algorithm tries to find a process that is believed to be fast by all other processes. When the set $Class_1$ is empty, it tries to find a process that is believed to be fast by a majority of processes (a process in the set $Class_2$). If there is no such process, it chooses one from Π. In all cases, the identities of processes are used to break ties.

The termination, validity and unifrom agreement properties of consensus guarantee that all correct processes will apply the above described deterministic function to the same global matrix, and therefore, will reach the termination, validity and agreement properties of the slowness oracle. Adaptability is guaranteed by using round trip delays collected during run time to update the global matrix [17].

4.3 Robust Design of Slowness Oracles

The implementation of the slowness oracle presented in the previous section has two drawbacks. First, it requires the definition of a threshold to classify fast and slow processes. As will be shown shortly in Section 5, the optimal threshold may vary among different scenarios. Secondly, it uses only the last round trip delay measured to classify a process, which for some scenarios may result in poor adaptation. We show in this section a novel design for a slowness oracle that tackles these issues in an effective way.

As previously discussed, the problem of identifying a threshold to differentiate fast processes from slow ones is closely related to that of setting the timeouts of a failure detection oracle. However, in the case of a slowness oracle, there is no need to use such a threshold. In fact, the very round trip delays measured can be used to identify the most appropriate process to be used as a coordinator. In particular, for the *CT-consensus* protocol, extra performance penalty can be incurred whenever the coordinator blocks waiting for messages. However, as soon as the coordinator receives messages from a majority of processes, it is able to proceed in the execution of the protocol. Thus, the most suitable coordinator is the process with which at least a majority of processes are able to communicate the fastest.

Regarding the second issue, considering only the last known round trip delay to select the coordinator may be inadequate, specially when the round trip delays distribution may oscillate widely within a relatively short time period. Again, this is an issue that has been studied in the context of designing adaptive failure detection oracles [3, 16]. In particular, the approach based on predictors studied by Nunes and Jansch-Pôrto [16] can be used to implement both adaptive failure detection oracles as well as slowness oracles.

Following this approach, it is possible to design a more decoupled version of an adaptive *CT-consensus* service. In this case, the consensus module is completely separated from the slowness oracle module. Further, a predictor element is introduced. The predictor module is associated to a probe generator that is responsible for gathering periodic information about round trip delays. Instead of using round trip information computed within the consensus module, the slowness oracle uses the predictor to periodically compose its local slowness information. Let us name this predictor-based adaptive consensus service *P-ACT consensus*.

Whenever the consensus module wants to start an execution of the consensus protocol it queries the slowness oracle for the identity of the coordinator of the first round. The slowness oracle module keeps a global slowness matrix that stores round trip delays, *i.e.*, entry [*j, k*] is the current predicted value for the round trip delay between processes p_j and p_k. Each entry of this matrix is initially set to 0. The slowness oracle applies an a priori agreed function over the consensual global slowness matrix to choose the coordinator of the first round and returns its identity to the consensus module. Moreover, the consensus module queries its local slowness oracle module to obtain the local slowness matrix that it wants to propose in the current consensus execution. Upon reaching a decision, the consen-

sus module instructs its local slowness oracle module to update its global slowness matrix with the slowness matrix value that has been agreed. Figure 1 illustrates this design (arrows indicate the direction of the communication flow among the consensus modules).

A possible procedure to extract the common identity of the coordinator from the global slowness matrix is the following: let *majority_rtti* be the $\lceil (n+1)/2 \rceil^{th}$ smallest round trip delay expected for the communication between any process and p_i (including p_i); the slowness oracle module calculates *majority_rttk* for every process p_k, $1 \leq k \leq n$ and outputs the identity of the process that has the smallest value for *majority_rtt* (with the ordered identities being used to break ties).

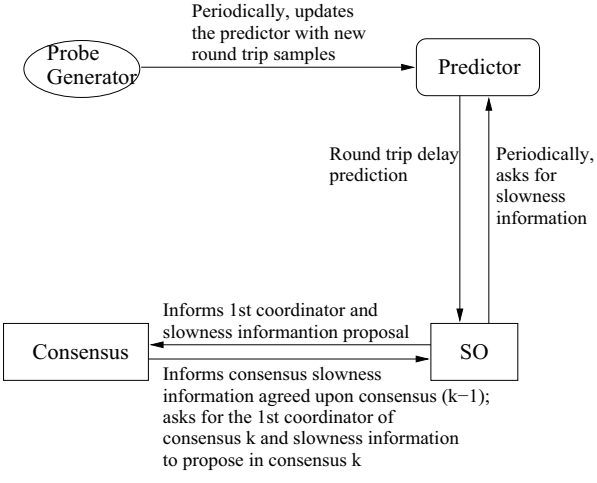


Figure 1: An adaptive consensus protocol supported by a predictor-based slowness oracle

There are several possibilities for implementing the predictor module [16]. In this work we use an implementation of the predictor BROWN. This predictor assumes a linear trend model and works as a double low-pass filter. By extrapolation its prediction follows, $\widehat{rtt}_{t+1} = 2.rtt_t^{st} - rtt_t^{nd} + \frac{\alpha}{1-\alpha}.(rtt_t^{st} - rtt_t^{nd})$, where $\alpha = 0.125$, rtt_t^{st} is the simple exponential smoothing, computed as $rtt_t^{st} = \alpha.rtt_t + (1-\alpha).rtt_{t-1}^{st}$, and rtt_t^{nd} is the double exponential smoothing, computed as $rtt_t^{nd} = \alpha.rtt_t^{st} + (1-\alpha).rtt_{t-1}^{nd}$.

As discussed above, a slowness oracle can be designed as an independent service that provides a specific functionality and can be accessed by a well defined interface. Such a design contributes to the separation of concerns when developing distributed systems. From a theoretical point of view, separation of concerns contributes in the sense of facilitating the proof of correctness of the system. On the other hand, it also allows more portable implementations of distributed protocols.

5 Performance Evaluation

We used a simulation model to evaluate the performance of the consensus protocols previously discussed. This evaluation was conducted with the support of the NEKO framework [22]. The use of the NEKO framework allows sharing the same Java source code for both simulation and executions in a real environment. Since crash failures are rare, we focused our study on failure free runs. Moreover, we considered configuration scenarios with n = 3 and n = 5 processes. The mean of the minimal execution time of the protocols was used as the performance metric. In simulations we measured time with the support of the clock of an external observer. All processes started the execution of a given consensus instance at the same time. Thus, the minimal execution time of each consensus instance is given by the time that the first process decides minus the time that the consensus execution has been started. The mean was calculated over the k consensus executions that comprised a given simulation experiment.

A key characteristic of the communication model provided by the NEKO framework is that it accounts for resource contention (see [21] for more details). We implemented a network model, based on NEKO, for a wide area network (WAN), in which only network contention is considered[4]. In this case, if a particular network channel $c_{i,j}$ is busy when p_i sends a message m to p_j, then m is put into a waiting queue, remaining there until $c_{i,j}$ is available for m to be transmitted. Communication delays are based on real workloads which describe the behavior of machines connected via a WAN. The workloads were obtained with the support of the Network Weather Service Tool (*nws* tool, available at http://nws.cs.ucsb.edu/). The experiments using the *nws* tool were run on five machines distributed among the following domains: USA (planetlab1.ucsd.edu), CANADA (cloudburst.uwaterloo.ca), JAPAN (planetlab1.koganei.wide.ad.jp), BRAZIL (planetlab1.pop.mg.rnp.br) and HAWAII (dsi.i2.hawaii.edu). We have monitored the latency (network transmission delay) between every pair of machines in the WAN, at every 5 minutes, during about 24 hours.

By using the workload traces for these machines, 3 network configurations were constructed, they are: i) **CF1** - JAPAN, USA, CANADA; ii) **CF2** - BRAZIL, CANADA, JAPAN; and iii) **CF3** - CANADA, USA, JAPAN, BRAZIL, HAWAII. Each network configuration describes 8 hours of workload. For each experiment, we simulated 8 hours of consensus executions, considering one consensus per second, which resulted in 28, 800 consensus per experiment.

5.1 Measuring the Impact of Adaptation

In the first experiment we investigated the performance of the slowness oracles. To eliminate any possible

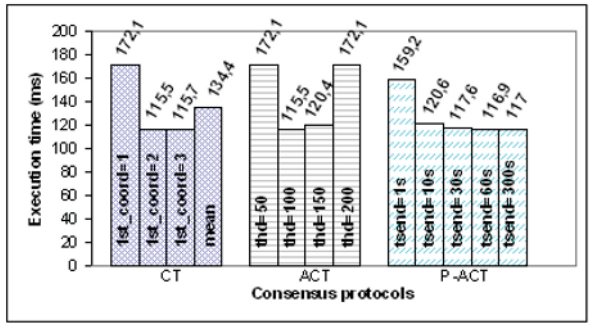
[4] A real distributed system can be subject to heterogeneous communication and processing loads. From the application point of view (*eg.* consensus services) such a load is perceived as the end-to-end delay in message exchanges. Thus, in a simulated environment, it does not make much difference whether it is the communication or the processing that contributes the most for the total end-to-end delay.

collateral effect caused by the failure detection oracle, we considered runs in which the failure detector made no mistake. In fact, similar to [17], we used a mock implementation of the failure detection oracle that always returned an empty set of suspected processes and generated no overhead.

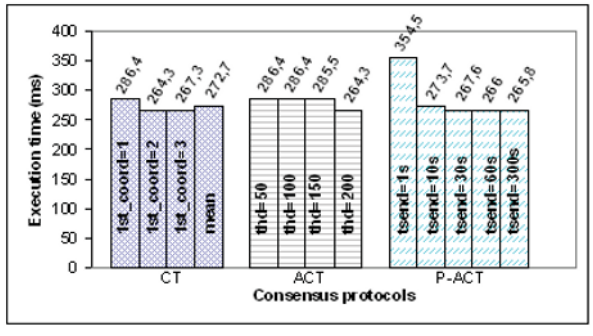
In *CT-consensus*, the coordinator of each consensus round is chosen a priori. Thus, with perfect failure detection and no failures, the performance of this protocol is influenced by the responsiveness of the machine on which the coordinator of the first round (1st coord) is running. We simulated *CT-consensus* under different configuration scenarios for 1st coord. The scenarios depended on the network configuration used. For instance, for **CF1** three scenarios were possible: i) 1st coord running in JAPAN, ii) 1st coord running in USA, and iii) 1st_coord running in CANADA. On the other hand, the performance of *ACT-consensus* is influenced by the threshold (thd) that indicates if a process is fast or slow. We simulated *ACT-consensus* with 4 different values for this threshold, namely: 50*ms*, 100*ms*, 150*ms*, and 200*ms*. Finally, the performance of *P-ACT-consensus* is influenced by the configuration of the predictor module used in the implementation of the slowness oracle. More precisely, it is possible to configure the frequency with which the probe generator calculates new round trip delays and informs the predictor of them. Five periods were used: 1 second, 10 seconds, 30 seconds, 1 minute, and 5 minutes. Note that better prediction is achieved with the lowest granularity, because it will cause the predictor and, consequently, the slowness oracle to be updated more often. On the other hand, the lower the granularity the higher the overhead introduced in the network due to the transmission of probe messages. We simulated one experiment for each of the scenarios described above. The results for all the experiments are presented in Figure 2.

Comparing *CT-consensus* and *ACT-consensus*, there is an optimal threshold for *ACT-consensus* in which its performance is as good as the performance of *CT-consensus* in the best configuration scenario (when 1st coord is running in the fastest machine). When considering the worst scenario for *CT-consensus*, the adaptive consensus service is as much as 59.3% better (see **CF3**). Moreover, comparing the best result for *ACT-consensus* with the mean value of all possible scenarios for *CT-consensus*, the adaptive consensus service outperforms the non-adaptive in as much as 29.7% (see **CF3**). Even when the workload is more homogeneous (see **CF2**), *ACT-consensus* performs as good as possible. However, as previously pointed out, *ACT-consensus* requires the definition of a threshold to classify fast and slow processes. From the results presented, we can observe that the optimal threshold varies among the different workloads (**CF1** = 100*ms*, **CF2** = 200*ms*, **CF3** = 150*ms*). On the other hand, except for the case in which the frequency of the probe generator is the lowest (1*s*), the results of *P-ACT-consensus* are very close to those obtained for *ACT-consensus*, considering the optimal threshold. We argue that configuring the sample rate of the probe generator is much easier than configuring the threshold of *ACT-consensus*. This is because this rate is application dependent, instead of environment dependent. Thus, it is possible to choose good values for it without considering the environment in which it executes. For example, in the case of the experiments performed with the consensus protocols studied, any rate around 1 minute is likely to achieve good performance.

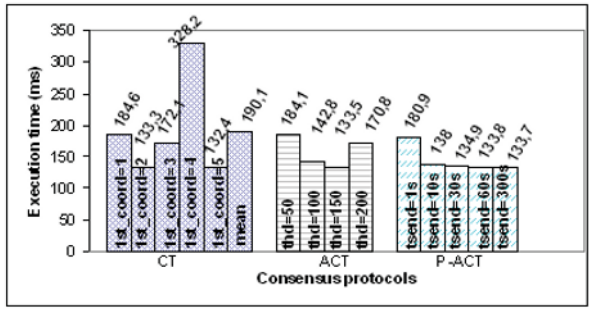
In summary, when conveniently configured, the adaptive protocols are able to adapt themselves to the best execution scenario (the one in which the most suitable coordinator is always chosen). Moreover, *P-ACT-consensus* is more robust against configuration mistakes than *ACT-consensus*.



(a) CF1



(b) CF2



(c) CF3

Figure 2: *CT-consensus*, *ACT-consensus* and *P-ACT-consensus* in different network configuration scenarios

5.2 Measuring the Impact of Unreliable Failure Detection

In this section, we analyze what is the impact of introducing a failure detector module in the consensus services presented. We aim at measuring the overhead introduced by the failure detector implementation and how it impacts the different consensus services.

We used two implementations of failure detectors, one based on a push-style (*FD-Push*) and another based on a pull-style (*FD-Pull*) [5]. Both implementations are non-adaptive but use dynamic timeouts. *FD-Push* uses timeouts that are increased by 1*ms* whenever a wrong suspicion is discovered. On the other hand, the timeouts of *FD-Pull* are also adjusted upon the detection of a wrong suspicion, but following a different approach. When a reply ("I_am_alive!" message) is received from a process p_j that is currently considered suspected, the round trip delay corresponding to the "I_am_alive!" message is calculated and used as the new timeout for p_j.

We ran the same simulation experiments described in Section 5.1 using *CT-consensus*, *ACT-consensus* and *P-ACT-consensus* with *FD-Push* and *FD-Pull*. The failure detectors were configured to send one message per second ("Are_you_alive?" query messages for *FD-Pull* and "I_am_alive!" heartbeat messages for *FD-Push*). The timeouts for *FD-Push* and *FD-Pull* were initially set to 50*ms* and 100*ms*, respectively (such values were defined considering the typical latencies of the workload used in the simulations).

For both *CT-consensus* and *P-ACT-consensus* the performance loss due to the introduction of the failure detector implementation was around 25%, when using *FD-Push*, and 27% when using *FD-Pull*. However, for *ACT-consensus* this impact varied from a 49% decrease to a 10% increase in performance. This is because the overhead introduced by the failure detector has an impact on the configuration of the threshold parameter. Thus, a good value for the threshold in an execution of the protocol without using a failure detector can become inadequate when introducing the failure detector and vice-versa. This is yet another indication that *P-ACT-consensus* is more robust than *ACT-consensus*.

6 An Architecture for Efficient and Robust Adaptive Consensus Services Based on Oracles

The simulation results presented in the previous section indicate that consensus protocols designed for asynchronous systems augmented with unreliable failure detectors can achieve better performance when using slowness oracles. Such oracles allow the protocols to adapt themselves to the changing conditions of the environment in which they execute. Further, the oracle based on a predictor was more robust with respect to changes in the environment than the simpler adaptive protocols that used a fixed threshold to control adaptation. However, depending on the frequency used to probe for round trip samples, the overhead introduced by the predictor may be significant and impact negatively the performance of the adaptive consensus protocols that use it. In this section we propose an optimization for *P-ACT-consensus* that tackles this problem.

We advocate that *P-ACT-consensus* should preferably use a pull-style failure detector. These failure detectors periodically send "Are_you_alive?" query messages and wait for corresponding "I_am_alive!" heartbeat messages to decide whether a process has crashed or not. These messages can also be used to calculate round trip delays among processes. Since the failure detection oracle monitors all processes engaged in the execution of the consensus protocol, the round trip delays collected by the failure detection oracle can be used to update the predictor of the slowness oracle. Following this approach, the failure detector behaves as a probe generator, greatly reducing the overhead traffic generated by the predictor. Furthermore, according to the study by Nunes and Jansch-Pôrto [16], predictors can also be used to implement efficient adaptive (pull-style) failure detection oracles. In this case, by using *P-ACT-consensus* with an adaptive implementation of *FD-Pull* based on predictors, it is possible to obtain even better performance results. This improved implementation of *P-ACT-consensus* can be generalized in an architecture for efficient and robust consensus services based on oracles, as illustrated in Figure 3.

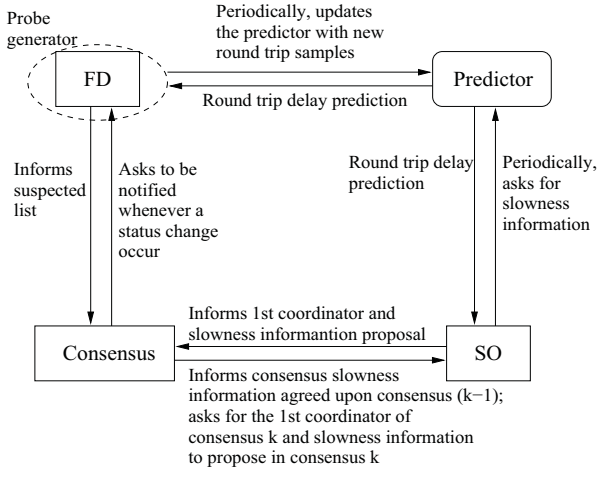


Figure 3: Architecture for a consensus service that uses predictors to support pull-style failure detection and slowness oracles implementations

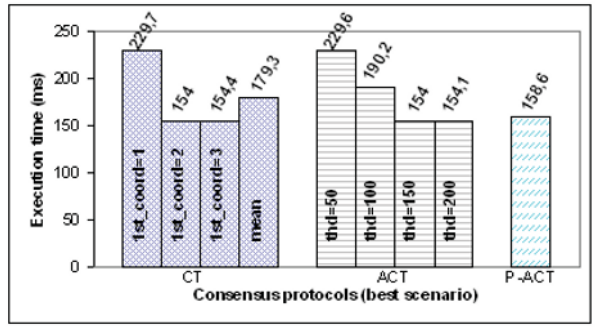
We simulated *P-ACT-consensus* with an adaptive version of *FD-Pull* based on predictors (let us call it by *AFD-Pull*). The design of *AFD-Pull* follows the work by Nunes and Jansch-Pôrto [16] on adaptive failure detectors. In this case, the failure detection oracle uses a predictor to define its timeouts (estimated round trips plus a safety margin). The configuration scenarios are the same described in Section 5. *AFD-Pull* is configured to send one "Are_you_ alive?" mes-

sage per second, causing the predictor to be updated with the same frequency. Further, the predictor uses a safety margin based on a confidence interval of the round trip delay estimator (*cib*[5]) [16]. *CT-consensus* and *ACT-consensus* were also simulated using *AFD- Pull*. However, considering the results for simulations presented in the previous section, the performance of the protocols using *FD-Push* were better. Thus, the performance of *P-ACT-consensus* using *AFD-Pull* is compared with the performance of *CT-consensus* and ACT-consensus using *FD-Push* (the best scenario). The results for all protocols are illustrated in Figure 4.

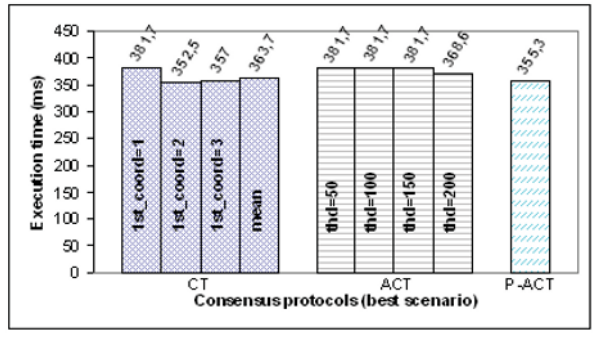
Again, the adaptive consensus services outperforms the non-adaptive service. We note that *P-ACT-consensus* uses only the predictor configuration that yields the greatest overhead (when compared to those presented in the previous section) and still reaches very good performance. For instance, in **CF3**, when *CT-consensus* presents the worst result, *P-ACT-consensus* presents a performance that is 58.7% better. Also, comparing with the mean execution time for all *CT-consensus* configurations, the adaptive protocol with predictor is 28.7% better. Note that *ACT-consensus* can be slightly better than *P-ACT-consensus* in some scenarios (see **CF1** for the threshold set to either 150*ms* or 200*ms*, and **CF3** for the threshold set to 200*ms*), but it is normally much worse than *P-ACT-consensus*, confirming the robustness of the latter.

7 A Case Study

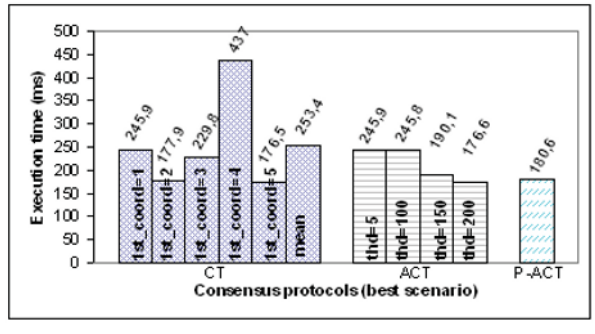
As earlier mentioned, a possible consensus application is a distributed scheduler for grid environments, as the one proposed for the PARADIS system [10]. In this system a distributed scheduler is built with the support of local scheduler modules at each site of the grid. Each local scheduler receives jobs and uses a consensus protocol to decide in which site the jobs will run. We implemented an application (AppScheduler) that mimics such a service. The distributed application is composed by local modules responsible for generating jobs following a particular distribution function. The jobs are stored by the corresponding local scheduler module in a job queue. Consensus is started by the local schedulers, in a sequential manner, whenever the job queue is not empty, using as proposition the content of the queue. Thus, the k^{th} consensus is executed after the $(k\text{-}1)^{th}$ consensus has finished. A decision proposal for the consensus protocol is formed as the union of the set of proposed jobs with the highest time-stamp value. In order to avoid that the jobs generated by slow processes are never included in the consensus decision, every time a new consensus is started the scheduler diffuses its job queue to all other processes. Processes merge job queues received through these messages with their local job queue. Each process *pi* uses the decision of the consensus to identify the jobs that need to be dispatched. Dispatched jobs are removed from the job queue. Moreover, *pi* calculates the queuing time for all locally generated jobs. The mean queuing time is used as performance metric.



(a) CF1



(b) CF2



(c) CF3

Figure 4: *CT-consensus*, *ACT-consensus* and *P-ACT-consensus* in different network configuration scenarios and considering the best matching between the consensus protocol and its oracles (failure detector and/or slowness oracle)

5The safety margin *cib* assumes that the predictor appropriately models the round trip delay (*rtt*), making the prediction error to be considered a white noise. Further, by considering that the estimator could behave as a linear function, at time t, the margin *cib* is computed by $cib_{t+1} = 2,58\ \hat{\sigma}\ \sqrt{1+\frac{1}{r}+\frac{(rtt_t-\overline{rtt})^2}{\sum_{i=1}^{n}(rtt_i-\overline{rtt})^2}}$, in which the constant 2.58 corresponds to the 99% of confidence in the standard Normal distribution function, $\hat{\sigma}$ is the estimator of the standard deviation of the *rtt*, and *r* is the number of samples considered in its computation.

We analyzed the performance of AppScheduler by running the application over the best configurations for *CT-consensus* and *P-ACT-consensus*, considering the results presented in Section 6. Further, we considered an extra scenario for *CT-consensus*, in which the 1st_coord of each instance of consensus execution was allocated in a round-robin manner during the experiment. The experiment consisted in executing a number of consensus for each service and for one configuration of 5 machines (**CF3**). Note that, in a grid environment, AppScheduler would execute in as much as possible machines, thus, **CF3** is the more appropriated configuration for our analysis. To limit the impact of variations in the system load, the consensus instances were divided in batches of 100 consensus each. For each batch, the services with the different consensus protocols were executed one after the other. The results of the experiment are illustrated in Figure 5 in terms of the cumulative average of the mean queuing time over all batches executed. Batches were executed until a convergence on the cumulative average was detected.

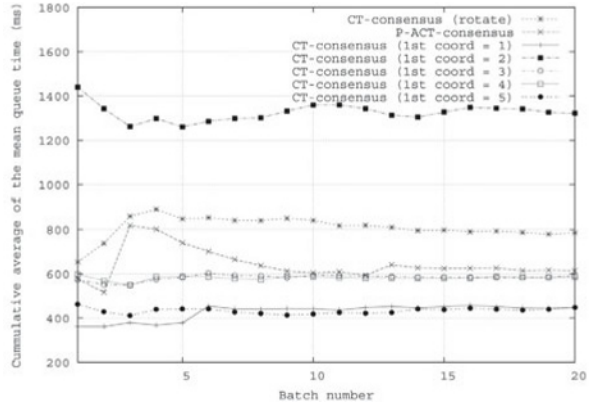


Figure 5: Performance results for AppScheduler using *CT-consensus* with

In the worst scenario for *CT-consensus*, AppScheduler using *P-ACT-consensus* was as much as 53.5% better. Considering the executions using *CT-consensus* with rotating 1st_coord (this scenario is close to the mean execution time for all other consensus configuration) this gain is about 21.7%. The results are compatible with those obtained by means of simulation, when running the protocols separately.

8 Conclusion

Slowness oracles had been proposed as a possible solution for the performance penalty that consensus protocols based on an asymmetric structure can suffer when running over systems subject to variabilities in load. Although only a very simple mechanism was used to define the oracle operation, the results were promising [17].

In this paper, we studied techniques to implement these slowness oracles aiming at an efficient operation; with this objective in view, the adaptation to the network parameters was also explored. More precisely, we presented a robust design for slowness oracles, in the sense that they allow adaptation to occur without requiring any difficult deployment decisions to be taken. Moreover, we showed how to match failure detection and slowness oracles in a more complete consensus architecture, in which both oracles follow a predictorbased approach [14, 15, 16].

The proposed architecture was applied to a classical consensus protocol, defined by Chandra and Toueg in [1] (*CT-consensus*). The performance of the resulting adaptive protocol, *P-ACT-consensus*, was analyzed by means of simulation and experiments in a real system, in the latter case, using a distributed scheduler application based on a consensus service. The performance of this application was compared with the performance of its corresponding non-adaptive protocol and other implementations of adaptive consensus protocols based on slowness oracles. The results we obtained show that *P-ACT-consensus* can be as much as 53.5% better than *CT-consensus*. Moreover, the proposed protocol is more robust, than the adaptive protocols based on slowness oracles published so far [17].

Acknowledgements

We would like to thank André Luis Moreira for his contribution on the implementation of the consensus protocols and the scheduler application presented in this paper. Authors would also like to thank the valuable comments from the anonymous referees and the financial support from CAPES/Brazil (grant 478.752/01) and CNPq/Brazil (grant 300.646/96). This work was partially developed in collaboration with HP Brazil R&D.

References

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

[2] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.

[3] W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. In *International Conference on Dependable Systems and Networks (DSN'2000)*, pages 191–200, New York, USA, Jun 2000. IEEE Computer Society.

[4] A. Coccoli, P. Urbán, A. Bondavalli, and A. Schiper. Performance analysis of a consensus algorithm combining stochastic activity networks and measurements. In *International Conference on Dependable Systems and Networks (DSN'2002)*, pages 551–560, Washington, D.C., USA, June 2002. IEEE Computer Society.

[5] P. Felber, R. Guerraoui, X. Défago, and P. Oser. Failure detector as first class objects. In *International Symposium on Distributed Objects and Applications*, pages 132–141, Edinburgh, Scotland, September 1999. IEEE Computer Society.

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

[7] R. Guerraoui and M. Raynal. The information structure of indulgent consensus. *IEEE Transactions on Computers*, 53(4):453–466, April 2004.

[8] R. Guerraoui and A. Schiper. The generic consensus service. *IEEE Transactions on Software Engineering*, 27(1):29–41, January 2001.

[9] M. Hurfin, R. Macêdo, M. Raynal, and F. Tronel. A generic framework to solve agreement problems. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS'99)*, pages 56–65, Lausanne, Switzerland, October 1999. IEEE Computer Society.

[10] M. Hurfin, J.P. Le Narzul, J. Pley, and Ph. Raipin Parvédy. A fault-tolerant protocol for resource allocation in a grid dedicated to genomic applications. In *Proceedings of the 5th International Conference on Parallel Processing and Applied Mathematics, Special Session on Parallel and Distributed Bioinformatic Applications (PPAM-03)*, volume 3019 of *LNCS*, pages 1154–1161, Czestochowa, Poland, September 2003. Springer.

[11] I. Keidar and S. Rajsbaum. Open questions on consensus performance. In *Future Directions in Distributed Computing, LNCS-2584, Springer*, volume 2584 of *LNCS*, pages 35–39. Springer, 2003.

[12] L. Lamport. Paxos made simple. *SIGACT News*, 32(4):18–25, December 2001.

[13] A. Mostéfaoui and M. Raynal. Leader-based consensus. *Parallel Processing Letters*, 11(1):95–107, March 2001.

[14] R. C. Nunes and I. Jansch-Pôrto. Modelling communication delays in distributed systems using time series. In *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems (SRDS'2002)*, pages 268–273, Osaka, Japan, 2002.

[15] R. C. Nunes and I. Jansch-Pôrto. A lightweight interface to predict communication delays. In *Proceedings of the First Latin American Symposium (LADC'2003)*, volume 2847 of *LNCS*, pages 245–263, S˜ao Paulo, Brazil, October 2003. Springer.

[16] R. C. Nunes and I. Jansch-Pôrto. Qos of timeout-based self-tuned failure detectors: the effects of the communication delay predictor and the safety margin. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'2004)*, pages 753–761, Florence, Italy, June 2004. IEEE Computer Society.

[17] L. M. R. Sampaio, F. V. Brasileiro, W. da C. Cirne, and J. C. A. de Figueiredo. How bad are wrong suspicions? towards adaptive distributed protocols. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'2003)*, pages 551–560, San Francisco, California, USA, June 2003. IEEE Computer Society.

[18] A. Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3):149–157, April 1997.

[19] N. Sergent, X. Défago, and A. Schiper. Impact of a failure detection mechanism on the performance of consensus. In *Proceedings of the 2001 Pacific Rim International Symposium on Dependable Computing (PRDC'2001)*, pages 137–145, Seoul, Korea, December 2001. IEEE Computer Society.

[20] J. Turek and D. Shasha. The many faces of consensus in distributed systems. *IEEE Computer*, 25(6):8–17, June 1992.

[21] P. Urbán, X. Défago, and A. Schiper. Contention-aware metrics for distributed algorithms: comparison of atomic broadcast algorithms. In *Proceedings of the 9th IEEE International Conference on Computer Communications and Networks (IC3N'2000)*, pages 80–92, Las Vegas, Nevada, USA, October 2000. IEEE Computer Society.

[22] P. Urbán, X. Défago, and A. Schiper. Neko: a single environment to simulate and prototype distributed algorithms. In *Proceeding of the 15th International Conference on Information Networking (ICOIN-15)*, pages 503–511, Beppu City, Japan, February 2001. IEEE Computer Society.

[23] P. Urbán, N. Hayashibara, A. Schiper, and T. Katayama. Performance comparison of a rotating coordinator and a leader based consensus algorithm. In *Proceedings of the 23rd Symposium on Reliable Distributed Systems (SRDS'2004)*, pages 4–17, Florianópolis, Brazil, October 2004. IEEE Computer Society.