Performance Analysis of a Consensus Algorithm Combining Stochastic Activity Networks and Measurements[*]

Andrea Coccoli[†] a.coccoli@guest.cnuce.cnr.it
Péter Urbán[‡] peter.urban@epfl.ch
Andrea Bondavalli[⋆] a.bondavalli@dsi.unifi.it
André Schiper[‡] andre.schiper@epfl.ch

[†]*CNUCE-CNR, Via Alfieri 1, I-56010 Ghezzano, Pisa, Italy*
[‡]*École Polytechnique Fédérale de Lausanne (EPFL), CH-1015 Lausanne, Switzerland*
[⋆]*Università di Firenze, Via Lombroso 6/17, 50134 Firenze, Italy*

Abstract

Protocols which solve agreement problems are essential building blocks for fault tolerant distributed applications. While many protocols have been published, little has been done to analyze their performance. This paper represents a starting point for such studies, by focusing on the consensus problem, a problem related to most other agreement problems. The paper analyzes the latency of a consensus algorithm designed for the asynchronous model with failure detectors, by combining experiments on a cluster of PCs and simulation using Stochastic Activity Networks. We evaluated the latency in runs (1) with no failures nor failure suspicions, (2) with failures but no wrong suspicions and (3) with no failures but with (wrong) failure suspicions. We validated the adequacy and the usability of the Stochastic Activity Network model by comparing experimental results with those obtained from the model. This has led us to identify limitations of the model and the measurements, and suggests new directions for evaluating the performance of agreement protocols.

Keywords: *quantitative analysis, distributed consensus, failure detectors, Stochastic Activity Networks, measurements*

1 Introduction

Agreement problems — such as atomic commitment, group membership, or total order broadcast — are essential building blocks for fault tolerant distributed applications, including transactional and time critical applications. These agreement problems have been extensively studied in various system models, and many protocols solving these problems have been published [1, 2]. However, these protocols have almost only been analyzed from the point of view of their safety and liveness properties, and very little has been done to analyze their *performance*. One of the reasons is probably that agreement protocols are complex, typically too complex for analytical approaches to performance evaluation. Nevertheless, a few papers have tried to analyze the performance of agreement protocols: [3] and [4] analyze quantitatively four different total order broadcast algorithms using discrete event simulation; [5] uses a contention-aware metric to compare analytically the performance of four total order broadcast algorithms; [6, 7] analyze atomic broadcast protocols for wireless networks, deriving assumption coverage and other performance related metrics; [8] presents an approach for probabilistically verifying a synchronous round-based consensus protocol; [9] evaluates the performability of a group-oriented multicast protocol; [10] compares the latency of a consensus algorithm by simulation, under different implementations of failure detectors.[1] In all these papers, except for [10, 8, 9], the protocols are only analyzed in failure free runs. This only gives a partial and incomplete understanding of their quantitative behavior. Moreover, in [3, 4] the authors model communication delays in a way that completely ignores contention on the network and the hosts: the communication delays are modeled using a distribution that was obtained independently from the agreement protocols analyzed. This approach does not account for the fact that the transmission delay of messages is greatly influenced by the message traffic that the algorithm generates itself. In fact, the transmission delay of messages cannot realistically be assumed to be independent of the algorithm that generates them.

A detailed quantitative performance analysis of agreement protocols represents a huge work. Where should such a work start? As most agreement problems are related to the abstract consensus problem [11, 12, 13] it seems natural to start by a performance analysis of a consensus algorithm, and to extend the work of [10]. This is the goal of this paper, which analyzes a consensus algorithm.

The consensus problem is defined over a set of processes. Informally, each process in this set proposes a value initially, and the processes must decide on the *same* value, chosen among the proposed values [11]. It has been shown that consensus cannot be solved deterministically in an asynchronous model [14]. A stronger system model is thus re-

[*]Research supported by a grant from the CSEM Swiss Center for Electronics and Microtechnology, Inc., Neuchâtel, and by a Short Term Mobility grant from CNR, the Italian National Research Council.

[1]The algorithm is the same as in this paper, but the failure detection techniques and the definition of latency differ.

quired to solve consensus. We have chosen the asynchronous model with unreliable failure detectors [11], and we have decided to analyze the Chandra-Toueg consensus algorithm based on the $\diamond\mathcal{S}$ failure detector.

The paper evaluates the performance of the Chandra-Toueg $\diamond\mathcal{S}$ consensus algorithm by combining the following two approaches: (1) experiments on a cluster of PCs and (2) simulation. The simulation was conducted using Stochastic Activity Networks (SANs), a class of timed Petri nets. Only the control aspect of the consensus algorithm had to be modeled for the simulation: the data aspect (e.g., the content of messages) could be ignored. The failure detectors were modeled in an abstract way, using the quality of service (QoS) metrics proposed by Chen et al. [15]. Communications were modeled in a way that takes contention on the network and on the hosts into account. The goal of our quantitative analysis was to determine the latency of the consensus protocol, i.e., the time elapsed from the the beginning of the algorithm until the first process decides. Moreover, we evaluated the latency in different classes of runs: (1) runs with no failures nor failure suspicions, (2) runs with failures but no wrong suspicions, and (3) runs with no failures but with (wrong) failure suspicions. We combined the results of the simulations and the measurements: some measurement results have been used to determine input parameters for the simulation model. Furthermore, a validation of the adequacy and the usability of the model has been made by comparing experimental results with those obtained from the model. This validation activity led us to determine some limitations of the model and new ideas for the experimental measurements, and suggests us new directions for evaluating the performance of agreement protocols.

The paper is structured as follows. Section 2 presents the context of our performance analysis: the algorithms, the performance measures, and the environment for running the algorithm. Section 3 describes the SAN model of the consensus algorithm and its environment. Section 4 mentions interesting points of the implementation and the measurements. We present and discuss our results in Section 5, and conclude the paper in Section 6.

2 Context of our performance analysis

As stated in the previous section, the goal of the paper is the performance analysis of a consensus algorithm. In Section 2.1, we start by describing the consensus algorithm that we want to analyze. As this algorithm uses failure detectors, we give in Section 2.2 the algorithm used to implement failure detectors. Then, in Section 2.3 we explain what performance measures we want to obtain from our analysis. Obviously, these measures vary from one run to another, depending on the failure behavior of the processes, and the output of the failure detectors. In Section 2.4, we introduce the different classes of runs for which we obtain performance measures. Finally, in Section 2.5 we describe the hardware and software environment in which our consensus algorithm is supposed to run.

2.1 The $\diamond\mathcal{S}$ consensus algorithm

The consensus problem is defined over a set of n processes $p_1, \ldots, p_n$. Each process p_i starts with an initial value v_i and the processes have to decide on a common value v that is the initial value of one of the processes. We consider in our study the consensus algorithm based on the failure detector $\diamond\mathcal{S}$ proposed by Chandra and Toueg [11]. The algorithm requires a majority of correct processes. We describe below the algorithm, up to the level of detail sufficient to understand the experiments that we have conducted. The algorithm assumes an asynchronous system model augmented with (unreliable) failure detectors. In that model, each process has a local *failure detector module*, which maintains a list of processes that are suspected to have crashed. A process p_i can query its local failure detector module to learn whether some other process p_j is currently suspected or not. Roughly speaking, the $\diamond\mathcal{S}$ failure detector ensures that (1) every crashed process is eventually suspected for ever by every correct process (*completeness* property), and (2) eventually there exists a correct process that is no more suspected by any correct process (*accuracy* property).

The $\diamond\mathcal{S}$ consensus algorithm is based on the rotating coordinator paradigm: each process proceeds in a sequence of asynchronous *rounds* (i.e., not all processes necessarily execute the same round at a given time t), and in each round one process assumes the role of the *coordinator* (p_i is the coordinator for the rounds $kn + i$). All the processes have an a priori knowledge of the identity of the coordinator of a given round. Processes that are not coordinator in a given round are called *participants*. In each round, every message is sent either by the participants to the coordinator or by the coordinator to the participants. The role of the coordinator is to impose a decision value. If it succeeds, the consensus algorithm terminates. It it fails, a new round with a new coordinator starts, in which the new coordinator will in turn try to impose a decision value, etc. Knowing the details of the execution of one round is not necessary for understanding this paper. We refer the interested reader to [11].

2.2 Failure detection algorithm

As explained above, each process has a failure detector module, which manages a list of processes that are suspected to have crashed. The failure detector modules are *unreliable*, in the sense that they can make mistakes by incorrectly suspecting a correct process and also by not suspecting a crashed process.

A variety of techniques exist for implementing a failure detector: they are usually qualified as push and pull techniques. In push techniques, a given process periodically sends a message (called heartbeat message) to inform the failure detector module of other processes that it is alive. In pull techniques, the failure detector module periodically sends a ping message to other processes, and waits for a reply.

We chose a push-style failure detector implemented using heartbeat messages (Figure 1): each process periodically sends a heartbeat message to all other processes. Failure detection is parameterized with a timeout value T and a heartbeat period T_h. Process p starts suspecting process q if it has not received any message from q (heartbeat or application message) for a period longer than T. Process p stops suspecting process q upon reception of any message from q (heartbeat or application message). The reception of any message from q resets the timer for the timeout T.

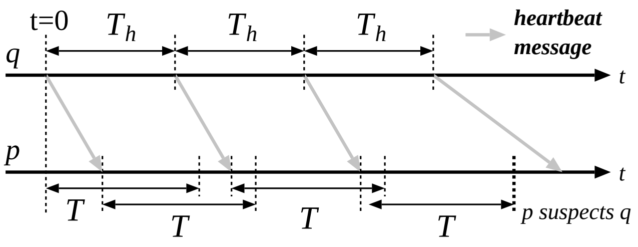


Figure 1. Heartbeat failure detection.

There exist heartbeat failure detectors with possibly better characteristics [15]. Our choice has the advantage that it is very simple to implement and control. In particular, it does not rely on synchronized clocks.

2.3 Latency as our performance measure

Latency and throughput are meaningful measures of the performance of algorithms. Roughly speaking, *latency* measures the time elapsed between the beginning and the end of the execution of an algorithm, while *throughput* measures the maximum number of times that a given algorithm can be executed per second.

Our study focuses on the *latency* of the consensus protocol, defined exactly as follows. We assume that all participants propose values at the same time t_0, and let t_1 be the time at which the first process decides. We define the latency as $t_1 - t_0$. This is a reasonable measure for the following reason. Consider a service replicated for fault tolerance using active replication [16]. Clients of this service send their requests to the server replicas using Atomic Broadcast [17] (which guarantees that all replicas see all requests *in the same order*). Atomic Broadcast can be solved by using a consensus algorithm [11]: a client request can be delivered at a server s_i as soon as s_i decides in the consensus algorithm. Once a request is delivered, the server replica processes the client request, and sends back a reply. The client waits for the first reply, and discards the other ones (identical to the first one). If we assume that the time to service a request is the same on all replicas, and the time to send the response from a server to the client is the same for all servers, then first response received by the client is the response sent by the server that has first decided in the consensus algorithm.

Studying the throughput of the $\diamond\mathcal{S}$ consensus algorithm will be one of the subjects of our future work. Throughput should be considered in a scenario where a sequence of consensus is executed, i.e., on each process, consensus #(k+1) starts immediately after consensus #k has decided. Note that, unlike in the definition of latency, not all processes necessarily start consensus at the same time.

2.4 Classes of runs considered

The latency of a consensus algorithm varies for different number of processes. However, given n, the latency can also vary from one run to another, depending on (1) the different delays experienced by messages, (2) the failure pattern of processes and (3) the failure detector history (i.e., the output of the failure detectors). We have considered the following classes of runs:

1. All processes are correct, and the failure detectors are accurate, i.e., they do not suspect any process.
   This is the scenario that one expects to happen most of the time. It assumes a failure detection mechanism that does not incorrectly suspects correct processes. There is a price to pay for the accuracy of the failure detector, though: the failure detection timeout T must be high to avoid wrong suspicions, thus detecting failures relatively slowly, or the heartbeat period T_h must be shortened thus increasing the network load.

2. One process is initially crashed, and the failure detectors are complete and accurate: the crashed process is suspected forever from the beginning, and correct processes are not suspected. We have further distinguished the following cases: (i) the first coordinator is initially crashed, and (ii) another process is initially crashed.

3. All processes are correct and the failure detectors are not accurate, i.e., they wrongly suspect some processes. We obtained the histories of the failure detectors by implementing the failure detector algorithm described in Section 2.2 and conducting measurements for different values of the parameters T_h and T. The failure detector histories obtained this way allowed us to estimate the failure detector quality of service (QoS) metrics defined in [15] (see Section 3.4). This metrics was then used in the simulation of the $\diamond\mathcal{S}$ consensus algorithm.

We performed experiments for each of these classes of runs.

2.5 Hardware and software environment

All experiments were run on a cluster of 12 PCs running Red Hat Linux 7.0 (kernel 2.2.19). The hosts have Intel Pentium III 766 MHz processors and 128 MB of RAM.

They are interconnected by a simplex 100 Base-TX Ethernet hub. The algorithms were implemented in Java (Sun's JDK 1.4.0 beta 2) on top of the Neko development framework [18]. All messages were transmitted using TCP/IP; connections between each pair of machines were established at the beginning of the test. The size of a typical message is around 100 bytes.

3 The SAN model for our performance analysis

3.1 Background: Stochastic activity networks

Stochastic activity networks (SANs) [19, 20] were developed for the purpose of performability evaluation: evaluations of performance and dependability. They belong to the broad family of Timed Petri Nets and have a very rich and powerful syntax thanks to primitives like activities, places, input gates, and output gates, thus allowing the specification of complex stochastic processes.

We used the UltraSAN tool [21] for solving SAN models. This tool provides a very general framework for building performability and/or dependability models. It supports a wide variety of analytical and simulative evaluation techniques with steady state and transient analysis. Timed Activities can have different kinds of distributions: exponential, deterministic, uniform, Weibull, etc., though the use of non-exponential distributions restricts the choice of the solvers to simulative ones (as it happened in our approach). Moreover, UltraSAN supports modular modeling: through the operators REP and JOIN different submodels may be replicated and joined together with common places. This allows an easier and faster modeling and reuse of previously built submodels.

3.2 Overview of the SAN model

As described in Section 2.1, the $\Diamond\mathcal{S}$ consensus algorithm cyclically evolves through rounds in which every process plays, in turn, the role of coordinator. The fact that all the messages in one round are exchanged with one process, different for each round, forced us to renounce to model the system using a parametric replication of the model of one single process. We needed to build a different submodel for every process involved in the algorithm execution. These submodels were composed together using the 'Join' facility offered by UltraSAN. The models for processes differ only in a few details, however. For this reason, we describe the model of just one process. Due to space constraints and the size of the model, we can only give an overview here. A detailed description of the model can be found in [22].

The model for a process (P1) represents the state machine that underlies one round of the the consensus algorithm: each state of the state machine is represented by one place and each state transition by an activity. Only the place that corresponds to the current state is marked. The model is subdivided into 5 submodels (see Fig. 2). Submodel P1C describes the actions of the process when it acts as a coordinator: it first waits for a majority of estimates, and then elaborates a proposal which is sent to the participants. Then it waits for a majority of acknowledgments and if they are all positive acknowledgments, it broadcasts the decision message. Otherwise (if it receives at least one negative ack) it passes to the next round. Submodels P1A1, P1A2a and P1A2b describe the actions of the process when it acts as a participant. In particular, during action P1A1 the process sends an estimate to the coordinator, then it waits for the coordinator's proposal. If it receives the proposal (action P1A2a), it replies with a positive acknowledgment, otherwise (if, while the process is waiting, the failure detector signals that it suspects the coordinator to have crashed; action P1A2b) it sends a negative acknowledgment.

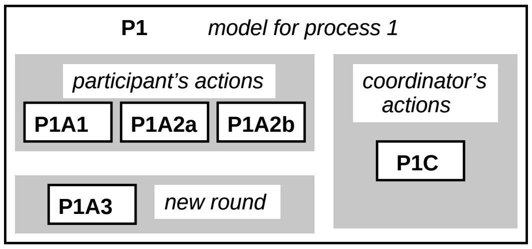


Figure 2. Overview of the SAN model of one process.

The submodel P1A3 is responsible for starting a new round and deserves a more detailed description. It contains a place which holds the current round number of the process modulo n, where n is the number of processes. The marking of this place controls if the process becomes coordinator or is a simple participant in the next round. The fact that the round number only holds the round number modulo n is a simplification of the algorithm. The effect of this simplification is that the algorithm only takes the messages of the last $n - 1$ rounds into account. While it is possible in the consensus algorithm that two processes are n or more rounds apart, this is rather improbable if a single instance of consensus is executed. For this reason, this simplification does not make our model less realistic.

The remaining places and activities (not implementing the state machine) are related to communication using messages and failure detectors. They are described in detail in the subsequent two sections.

3.3 The network model

We now describe how we modeled the transmission of messages. Our model is inspired from simple models of Ethernet networks [23, 24, 5]. The key point in the model is that it accounts for *resource contention*. This point is important as resource contention is often a limiting factor for the performance of distributed algorithms. In a distributed system, the key resources are (1) the CPUs and (2) the network medium, any of which is a potential bottleneck. For example, the CPUs may limit performance when a process has to receive information from a lot of other processes, and the network may limit performance when a lot of processes try to send messages at the same time.

The transmission of a message from a sending process p_i to a destination process p_j involves two kinds of resources. There is one network resource (shared among all processes) which represents the transmission medium. Only one process can use this resource for message transmission at any given point in time. Additionally, there is one CPU resource attached to each process. These CPU resources represent the processing performed by the network controllers and the communication layers, during the emission and the reception of a message (the cost of running the distributed algorithm is neglected, hence this does not require any CPU resource). The transmission of a message m occurs in the following steps (see Fig. 3):

1. m enters the *sending queue* of the sending host, waiting for CPU_i to be available.
2. m takes and uses the resource CPU_i for some time t_{send}.
3. m enters the *network queue* of the sending host and waits until the network is available for transmission.
4. m takes and uses the network resource for some time t_{net}.
5. m enters the *receiving queue* of the destination host and waits until CPU_j is available.
6. m takes and uses the resource CPU_j of the destination host for some time $t_{receive}$.
7. Message m is received by p_j in the algorithm.

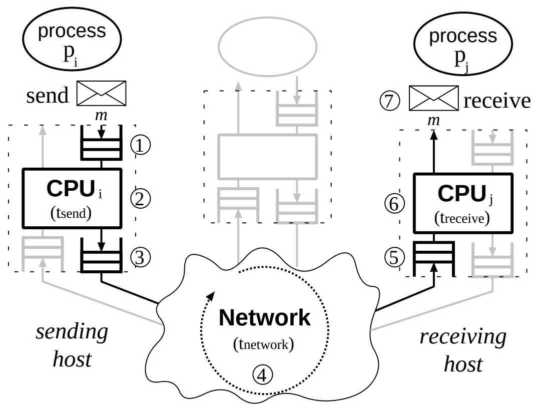


Figure 3. Decomposition of the end-to-end delay.

Parameters. The model defined needs to be fed with the three parameters t_{send}, t_{net} and $t_{receive}$. Due to the difficulties to measure their values on the prototype (see Section 5.1) we assumed (following [5]) t_{send} and $t_{receive}$ to be constants, with $t_{send} = t_{receive}$. The parameters t_{net}, t_{send} and $t_{receive}$ have been derived from the results of measurements (Section 5.1 and 5.2). We observed that a bi-modal distribution (described in Section 5.1) was a good fit of the results for t_{net}.

3.4 Failure detection model

One approach to modeling a failure detector is to build a model of the failure detection algorithm. However, this approach would complicate the model to a great extent (we would have to model the messages used for failure detection). Such a level of detail is not justified as we model other, more important components of the system (e.g., the network) in much less detail. For this reason, we chose a very simple model for failure detectors. Each process monitors every other process, thus each process has $n-1$ failure detectors (n is the number of processes). Consider any pair of processes p and q and the failure detector at q that monitors p. Each of these failure detectors is modelled as a process with two states, which alternates between states meaning "q trusts p" and "q suspects p". Note that the underlying assumption is that the behavior of each failure detector is independent of the others. This constitutes a major simplification. Indeed, as the heartbeat messages are affected by contention (either because of other heartbeats or other messages of the consensus algorithm) the outputs of the failure detectors at a given time are expected to be correlated.

The question remains how to set the transition rates in this simple model. We adopted the following solution. We measured some quality of service (QoS) metrics of failure detectors in experiments on our cluster, and then adjusted the transition rates in the model such that the model of the failure detector has the same average value for the QoS metrics as the real failure detector. QoS metrics for failure detectors were introduced in [15]. The authors consider the failure detector at a process q that monitors another process p, and identify the following three primary QoS metrics (see Fig. 3.4):

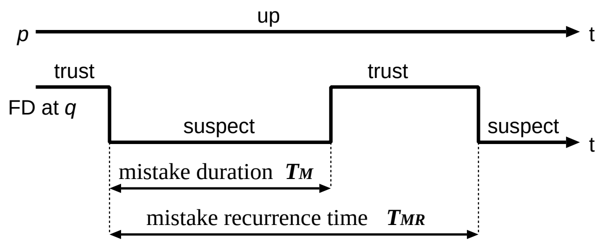


Figure 4. Quality of service metrics for failure detectors. Process q monitors process p.

Detection time T_D: The time that elapses from p's crash to the time when q starts suspecting p permanently.

Mistake recurrence time T_{MR}: The time between two consecutive mistakes (q wrongly suspecting p), given that p did not crash.

Mistake duration T_M: The time it takes a failure detector component to correct a mistake, i.e., to trust p again (given that p did not crash).

These QoS metrics are random variables. In our experiments on the cluster, we estimate the mean values for T_{MR} and T_M and use these values to configure the failure detector model (in future work, we plan to improve the model by estimating the distributions of these metrics and incorporating these distributions into the model). We considered two different time distributions for the transition from one state to the other: a deterministic and an exponential distribution, so to have, for the same mean value, a distribution with the minimum variance (0) and a distribution with a high variance.

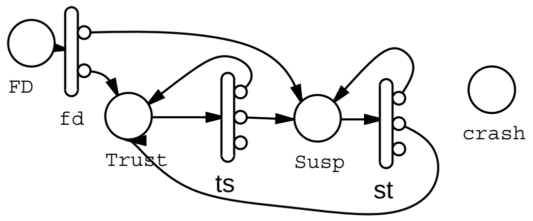


Figure 5. SAN model of the local failure detector module.

Figure 5 shows the way the local failure detector module has been modelled. As it can be seen, the two states are represented by the places Trust and Susp, while the (deterministic or exponential) transitions from one state to the other is managed by the activities ts and st. The activities present three possible outputs. The first two (in descending order) manage the alternance between the two states. The third one makes the activity of the failure detector stop when a decision is taken or when no more processes are alive; this is needed to stop the execution of the model once all interesting events took place. At the beginning of the simulation, an instantaneous activity fd determines the initial state of the failure detector module according to the probabilities associated to its outputs.

In this work, we did not model the contention on the network due to failure detectors. This is a choice we did on the basis of several measurements where the extra load generated did not affect the latency (the network bandwidth managed heartbeat and other messages without any problem). We reserve to take into account this phenomenon in a future and more refined model.

4 Implementation issues

In this section, we discuss some issues related to the $\diamond\mathcal{S}$ consensus experiments on the cluster.

Measuring latency. Since the latency values to be measured are rather small (sometimes < 1 ms) we had to measure time extremely precisely. The resolution of Java's clock (1 ms) was not sufficient, i.e., we had to implement a clock with a higher resolution (1 μs) in native C code. Also, the clocks of the hosts had to be synchronized precisely, in order to start the consensus algorithm of all the processes at the same time t_0 (see Section 2.3). We were able to achieve clock synchronization with a precision of $\ll 50\,\mu s$, using the NTP daemon [25] which provides advanced clock synchronization algorithms at minimal cost. This is far less than the transmission time of a message ($\approx 180\,\mu s$). This allowed us to have all processes start the consensus algorithm within a time window of $50\,\mu s$.

Isolation of multiple consensus executions. The latency of the $\diamond\mathcal{S}$ consensus algorithm was computed by averaging over a large number of (sequential) executions of the algorithm. However, with multiple executions of consensus it might happen that messages of consensus #k interfere with messages of consensus #$(k+1)$. In order to isolate completely the execution of two consensus algorithms, we have separated the beginning of two consecutive consensus executions by 10 ms, a value that was sufficient to avoid interferences.[2]

Measuring the QoS parameters of the failure detector. The failure detector outputs are only used in the runs of class 3 (Section 2.4), i.e., in runs were all processes are correct and the failure detectors are not accurate. We estimate the QoS parameters of a failure detector from its history during the experiment, i.e., from the state transitions *trust-to-suspect* and *suspect-to-trust*, and the time when these transitions occur. Note that these transitions were recorded during the full duration of an experiment, which encompasses multiple executions of the consensus algorithm: one new execution every 10 ms. While the different executions of the consensus are isolated one from another, the failure detectors are not reset to some initial state at the beginning of each consensus — this would not make any sense, considering the short latency of the consensus algorithm. The consequence is that we had to measure the QoS parameters of the failure detector for the full duration of the experiment (multiple consensus), rather than the duration of one single consensus.

Let T_{exp} be the duration of the experiment (multiple consensus), and let us consider the pair of processes (p, q). Let T_S^{pq} the time the failure detector of process p spent suspecting process q, n_{TS}^{pq} the number of trust-to-suspect transitions of p w.r.t. q, and n_{ST}^{pq} the number of suspect-to-trust transitions. The QoS metrics described in Section 3.4, i.e., the average mistake duration T_M and the average mistake recurrence time T_{MR}, are computed for the pair of processes (p, q) from the two equations:

$$\frac{T_M^{pq}}{T_{MR}^{pq}} = \frac{T_S^{pq}}{T_{exp}} \quad \text{and} \quad T_{exp} = \frac{n_{TS}^{pq} + n_{ST}^{pq}}{2} \cdot T_{MR}^{pq}$$

[2]In the few experiments with extremely bad failure detection we observed latencies above 10 ms (see Section 5.4) and thus we had to increase the separation.

We obtain the QoS metrics T_M and T_{MR} for the failure detector by averaging over the values T_M^{pq} and T_{MR}^{pq} for all pairs (p, q).

5 Results

We now present the results of the measurements and the simulations (we chose simulation solvers instead of analytical ones because we had to account for non-exponential distributions which capture much better the actual behavior of the system). We executed the same kind of experiments on both the SAN simulation model and the cluster whenever that was possible. We present results using both approaches for 3 and 5 processes, and results obtained on the cluster for 7, 9 and 11 processes.[3]

5.1 Setting the parameters of the SAN model

As we described in Section 3.3, we model the transmission of messages by reproducing the contention for the processors and for the network. The network model has three parameters t_{send}, $t_{receive}$ and $t_{network}$ that determine the end-to-end delay of a message. Messages sent to all n processes are treated specially, in order to reduce the size of the SAN model. Whereas in the implementation they are $n-1$ unicast messages, in the model they appear as a single broadcast message, with a higher parameter $t_{network}$ than unicast messages.

We tuned the parameters on the basis of measurements that give end-to-end delay of unicast and broadcast messages (for $n = 3$ and $n = 5$). Figure 6 shows the cumulative distribution in each of these three cases. These distributions were approximated by using uniform distributions in a bi-modal fashion, thus giving, in the case of unicast message: U[0.1, 0.13] (with a probability of 0.8) and U[0.145, 0.35] (with a probability of 0.2), where U[x,y] stands for a uniform distribution between x and y. The values are to be considered as milliseconds. From the measurements in Figure 6 we determined the parameters as follows: (1) t_{send} and $t_{receive}$ are assumed constant and equal (as suggested in previous works [24, 5]), (2) experiments reported in Section 5.2 allowed us to obtain $t_{send} = t_{receive}$, and (3) $t_{network}$ is computed as the end-to-end delay minus $2 \cdot t_{send}$. In the model, this implies the use of an instantaneous activity for $t_{network}$ with two outputs — whose case probabilities are the probabilities of the bi-modal like distribution — followed by two uniform timed activities.

5.2 No failures, no suspicions

Our first results show the latency when no failures occur and no failure detector ever suspects another process. Figure 7(a) shows the cumulative distribution of all observed

[3]Running the algorithm with an even number of hosts is not worthwhile: the consensus algorithms tolerates k crashes both with $2k + 1$ and $2k + 2$ processes.

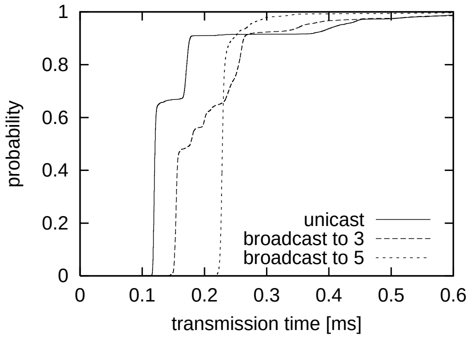


Figure 6. The cumulative distribution of the end-to-end delay of unicast and broadcast messages, averaged over the destinations.

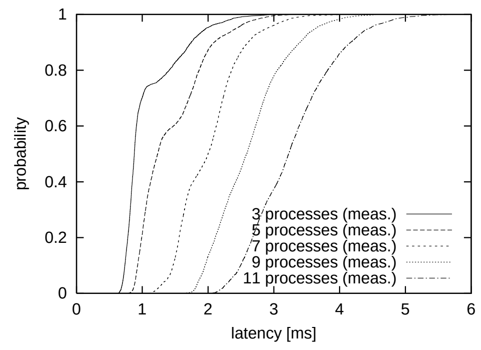
latency values, for a variety of values for n (the number of processes) obtained from measurements. The measurements come from 5000 consensus executions on the cluster for each value of n. Figure 7(b) reports the cumulative distribution of latency values for 5 processes obtained by simulation. Simulations were performed with the same end-to-end delay for message transmission but varying $t_{send} = t_{receive}$, and thus $t_{network}$. The figure shows that the simulation and measurement results match rather well when $t_{send} = 0.025$ ms, which suggests for that value a proper division between the different contributions of t_{send}, $t_{receive}$ and $t_{network}$ to the end-to-end delay. On the basis of these results we choose $t_{send} = t_{receive} = 0.025$ ms, and this value was used throughout all the simulations.

The mean values for the latency are the following: for $n = 3$, 1.06 ms (measurements) and 1.030 ms (simulation); for $n = 5$, 1.43 ms (measurements) and 1.442 ms (simulation); for $n = 7$, 2 ms (measurements); for $n = 9$, 2.62 ms (measurements); and for $n = 11$: 3.27 ms (measurements). The 90% confidence intervals for the measured means have a half-width smaller than 0.02 ms.

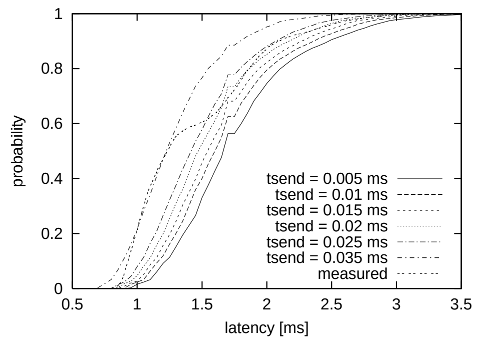
5.3 Failures, no incorrect suspicions

The next results were obtained for the case of one process crash. We assume that the crash occurs before the start of the consensus algorithm; the crashed process is suspected forever from the beginning, while the other (correct) processes are never suspected. We distinguish two cases: (1) the first coordinator (process 1) has crashed and thus the algorithm finishes in two rounds, and (2) a participant of the first round (process 2) crashed and thus the algorithm finishes in one round.

Our results are summarized in Table 1. We can see that the crash of the coordinator always increases the latency w.r.t. the crash-free case. The reason is that the consensus algorithm executes two rounds rather than one in that case. On the other hand, the crash of a participant has a more interesting influence: it decreases the latency for the consensus executions, except for the executions with 3 processes. The reason is that the crashed process does not gen-



(a) measurements for all n



(b) simulations with various settings for T_{send} and $n = 5$

Figure 7. The cumulative distribution of the latency.

erate messages, and thus there is less contention on the network and on the coordinator. The case $n = 3$ is special: the measurements show an increased latency. In this case, the number of messages exchanged is so small that the decreasing contention plays a secondary role. We can explain the latency increase as follows. The algorithm starts with the coordinator sending a message m to both participants: m is sent first to one participant p, and then to the other participant q. The reply of one participant is enough to come to a decision. Now, if p is crashed, q will reply, but the message m sent to p delays the sending of m to q.

In the simulation, the sending of message m is modeled as one single broadcast message. This explains that the special case $n = 3$ is not observed.

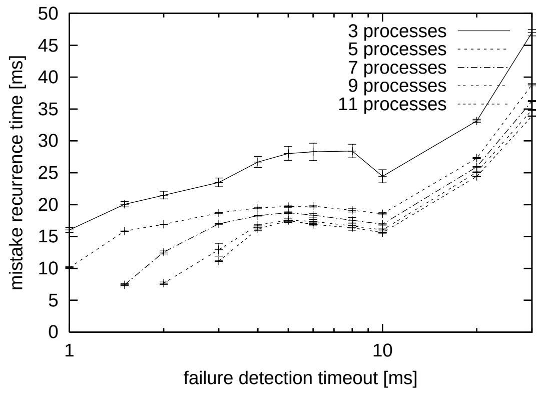
5.4 No failures, wrong suspicions

The next scenario we considered had no process crashes, but failure detectors sometimes wrongly suspected processes. We measured the quality of service metrics of the failure detectors (see Sect. 3.4) for a variety of values of the parameters T_h (heartbeat period) and T (timeout) (see Sect. 2.2). The QoS values served as input parameters for the SAN model. For both the quality of service metrics and the latency measurements, we executed 20 runs for each setting of the parameters T and T_h, where each run consisted of 1000 consensus executions. We computed the mean values and their 90% confidence intervals from the mean values measured in each of the runs.

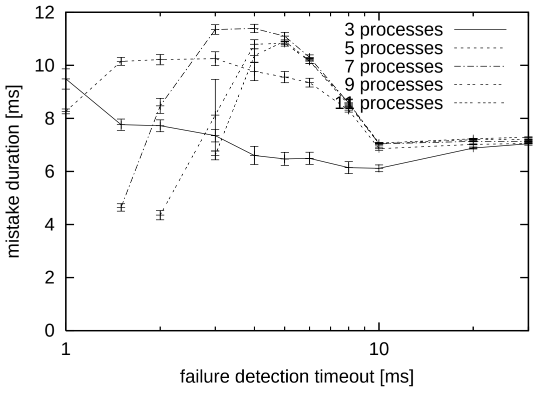
We present the quality of service metrics first, and the latency second, along with the SAN simulation results for latency.

Quality of service parameters. We found that modifying the heartbeat period T_h hardly influenced any of the quantities measured. For this reason, we treated only T as an independent parameter and we fixed T_h at $0.7 \cdot T$ for all experiments. The quality of service metrics T_{MR} and T_M are plotted in Fig. 8 as a function of T.

The mistake recurrence time curve (Fig. 8(a)) has an increasing tendency: suspicions occur more and more rarely. The curve only shows values up to $T = 30$ ms. At $T > 30$ ms, T_{MR} starts rising very fast: $T_{MR} > 190$ ms at $T = 40$ ms, and $T_{MR} > 5\,000$ ms at $T = 100$ ms, for each value of n (90% confidence).[4] The mistake duration curve (Fig. 8(b)) is less regular. It remains bounded (<12 ms) for all values of T.



(a) Mistake recurrence time T_{MR} as a function of T



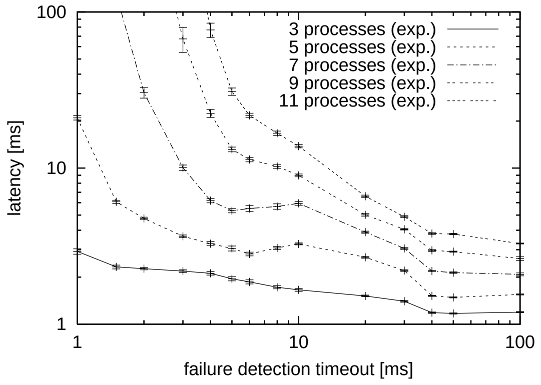
(b) Mistake duration T_M as a function of T

Figure 8. Quality of service metrics of the failure detector vs. the failure detection timeout T. No failures occur.

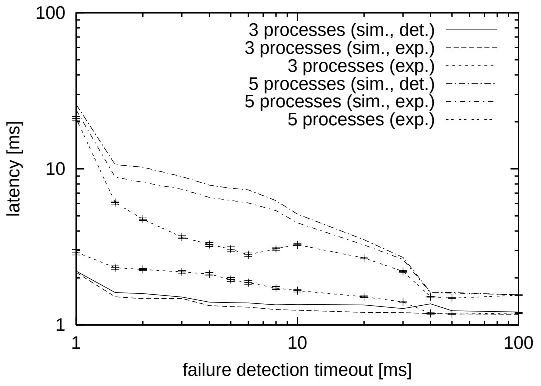
[4]Note that we do not need to determine T_{MR} (and T_M) precisely if T_{MR} is large, as the corresponding consensus latency values are nearly constant at those values.

latency [ms]	$n = 3$		$n = 5$		$n = 7$	$n = 9$	$n = 11$
	meas.	sim.	meas.	sim.	meas.	meas.	meas.
no crash	1.06	1.030	1.43	1.442	2.00	2.62	3.27
coordinator crash	1.568	1.336	2.245	2.295	2.739	3.101	3.469
participant crash	1.115	0.786	1.340	1.336	1.811	2.400	3.049

Table 1. Latency values (ms) for various crash scenarios from measurements and simulations.



(a) measurements



(b) measurements vs. simulation (det=deterministic distribution; exp=exponential distribution, see Sect. 3.4)

Figure 9. Latency vs. the failure detection timeout T. No failures occur.

Latency. Figure 9 shows the latency results obtained from both the measurements on the cluster and the simulations. Each latency curve starts at very high values, and decreases fast to the latency in the absence of suspicions. It is a decreasing curve, except for a small peak around $T = 10\,\mathrm{ms}$ at $n = 5$ and 7 in the measurement results. A possible explanation for this peak is interference with the Linux scheduler, in which the basic scheduling time unit is 10 ms (note also the behavior of the curves of the QoS metrics in Fig. 8 for $T = 10$ ms). The suspicions generated are likely to differ if the thread of the failure detector sleeps slightly more than 10 ms or slightly less than 10 ms.

By comparing the simulation and the measurement results, it is possible to notice some quite relevant differences. Actually, the SAN model is not able to perfectly catch the influence of the failure detectors when wrong suspicions are frequent (bad QoS). When the failure detectors' QoS is good — at high values for T — the results from the SAN model and measurements match. As we said in Section 3.4, each failure detector is assumed to be independent from the others. In reality, in case of contention on the system resources, there is likely to be correlation on false failure suspicions. The assumption of independence is thus not correct. The probability that two (or more) failure detectors see the expiring/respect of the timeout is not just the product of the individual probabilities of all failure detectors. Hence there should be a correlation between the states of all the failure detectors. Since correlation among failure detectors is relevant for the behavior of the protocol, further work will focus on accounting for that correlation, either by characterizing the QoS of the failure detectors in more detail and incorporating them into the model, or by modeling in detail the message flow of the failure detection algorithm.

6 Conclusion

In this paper we have applied a combined approach — modeling based simulations and experimental measurements — for the evaluation of the Chandra-Toueg $\diamond\mathcal{S}$ consensus algorithm (an algorithm that assumes an asynchronous system augmented with failure detectors). We identified the *latency* as a performance metric of interest, which reflects how much time the algorithm needs to reach a decision. For the failure detection we considered a simple heartbeat algorithm, and we tried to abstract its behavior in terms of appropriate quality of service (QoS) metrics. We investigated the latency of the consensus algorithm in three classes of runs, which differed in the behavior of the failure detectors and with respect to the presence or absence of crashes. These scenarios capture the most frequent operative situations of the algorithm.

We made measurements to determine input parameters for the simulation model. Furthermore, a validation of the adequacy and the usability of the simulation model has been made by comparing experimental results with those obtained from the model. This validation activity led us to determine some limitations of the model (e.g., the assumption about the independence of the failure detectors), and new directions for the measurements (e.g., extracting distributions for the QoS metrics of failure detectors).

Our efforts opened many interesting questions and directions, confirming how wide the problem is of providing quantitative analyses of distributed agreement algorithms. The work presented can be extended by introducing new performance metrics (e.g., throughput) and by investigating more deeply the behavior of the algorithm under particular conditions (e.g., transient behavior after crashes). Nevertheless, this work allowed us to gain insight on the behavior of the model and the implementation of the $\diamond\mathcal{S}$ consensus algorithm, which will be useful for further refinements. In order to give a complete evaluation of the Chandra and Toueg algorithm, i.e., to decide if it is a good one for a system like ours from a quantitative perspective, it is necessary to compare its performance with alternative solutions. This is our future plan: we will analyze alternative protocols and then we will be able to make statements about how good the protocols are by comparing the results. Such a work will also consolidate our framework for protocol analysis.

References

[1] M. Barborak, M. Malek, and A. Dahbura, "The consensus problem in distributed computing," *ACM Computing Surveys*, vol. 25, pp. 171–220, June 1993.

[2] X. Défago, A. Schiper, and P. Urbán, "Totally ordered broadcast and multicast algorithms: A comprehensive survey," Tech. Rep. DSC/2000/036, École Polytechnique Fédérale de Lausanne, Switzerland, Sept. 2000.

[3] F. Cristian, R. de Beijer, and S. Mishra, "A performance comparison of asynchronous atomic broadcast protocols," *Distributed Systems Engineering Journal*, vol. 1, pp. 177–201, June 1994.

[4] F. Cristian, S. Mishra, and G. Alvarez, "High-performance asynchronous atomic broadcast," *Distributed System Engineering Journal*, vol. 4, pp. 109–128, June 1997.

[5] P. Urbán, X. Défago, and A. Schiper, "Contention-aware metrics for distributed algorithms: Comparison of atomic broadcast algorithms," in *Proc. 9th IEEE Int'l Conf. on Computer Communications and Networks (IC3N 2000)*, Oct. 2000.

[6] A. Coccoli, S. Schemmer, F. D. Giandomenico, M. Mock, and A. Bondavalli, "Analysis of group communication protocols to assess quality of service properties," in *Proc. IEEE High Assurance System Engineering Symp. (HASE'00)*, (Albuquerque, NM, USA), pp. 247–256, Nov. 2000.

[7] A. Coccoli, A. Bondavalli, and F. D. Giandomenico, "Analysis and estimation of the quality of service of group communication protocols," in *Proc. 4th IEEE Int'l Symp. on Object-oriented Real-time Distributed Computing (ISORC'01)*, (Magdeburg, Germany), pp. 209–216, May 2001.

[8] H. Duggal, M. Cukier, and W. Sanders, "Probabilistic verification of a synchronous round-based consensus protocol," in *Proc. 16th IEEE Symp. on Reliable Distributed Systems (SRDS'97)*, (Durham, NC, USA), pp. 165–174, Oct. 1997.

[9] L. M. Malhis, W. H. Sanders, and R. D. Schlichting, "Numerical evaluation of a group-oriented multicast protocol using stochastic activity networks," in *Proc. 6th Int'l Workshop on Petri Nets and Performance Models*, (Durham, NC, USA), pp. 63–72, Oct. 1995.

[10] N. Sergent, X. Défago, and A. Schiper, "Impact of a failure detection mechanism on the performance of consensus," in *Proc. IEEE Pacific Rim Symp. on Dependable Computing (PRDC)*, (Seoul, Korea), Dec. 2001.

[11] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *J. ACM*, vol. 43, pp. 225–267, Mar. 1996.

[12] J. Turek and D. Shasha, "The many faces of consensus in distributed systems," *IEEE Computer*, vol. 25, pp. 8–17, June 1992.

[13] R. Guerraoui and A. Schiper, "The generic consensus service," *IEEE Transactions on Software Engineering*, vol. 27, pp. 29–41, Jan. 2001.

[14] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, pp. 374–382, Apr. 1985.

[15] W. Chen, S. Toueg, and M. K. Aguilera, "On the quality of service of failure detectors," in *Proc. Int'l Conf. on Dependable Systems and Networks (DSN)*, (New York, USA), pp. 191–200, June 2000.

[16] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: a tutorial," *ACM Computing Surveys*, vol. 22, pp. 299–319, Dec. 1990.

[17] V. Hadzilacos and S. Toueg, "Fault-tolerant broadcasts and related problems," in *Distributed Systems* (S. Mullender, ed.), ACM Press Books, ch. 5, pp. 97–146, Addison-Wesley, second ed., 1993.

[18] P. Urbán, X. Défago, and A. Schiper, "Neko: A single environment to simulate and prototype distributed algorithms," in *Proc. of the 15th Int'l Conf. on Information Networking (ICOIN-15)*, (Beppu City, Japan), Feb. 2001. Best Student Paper award.

[19] A. Movaghar and J. Meyer, "Performability modeling with stochastic activity networks," in *Proc. Real-Time Systems Symp.*, (Austin, TX, USA), Dec. 1984.

[20] J. F. Meyer, A. Movaghar, and W. H. Sanders, "Stochastic activity networks: structure, behaviour and applications," *Proc. International Workshop on Timed Petri Nets. Publ by IEEE, New York*, pp. 106–115, 1985.

[21] W. H. Sanders, W. D. O. II, M. A. Qureshi, and F. K. Widjanarko, "The UltraSAN modeling environment," *Performance Evaluation*, vol. 24, no. 1-2, pp. 89–115, 1995.

[22] A. Coccoli, *On Integrating Modelling and Experiments in Dependability and Performability Evaluation of Distributed Applications*. PhD thesis, University of Pisa, Italy, 2002. http://bonda.cnuce.cnr.it/Documentation/People/ACoccoli.html.

[23] K. Tindell, A. Burns, and A. J. Wellings, "Analysis of hard real-time communications," *Real-Time Systems*, vol. 9, pp. 147–171, Sept. 1995.

[24] N. Sergent, *Soft Real-Time Analysis of Asynchronous Agreement Algorithms Using Petri Nets*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 1998. Number 1808.

[25] D. Mills, "Network Time Protocol (version 3), specification, implementation and analysis," *IETF*, Mar. 1992.