Performance Comparison Between the Paxos and Chandra-Toueg Consensus Algorithms*

Naohiro Hayashibara† nao-haya@jaist.ac.jp
Péter Urbán‡ peter.urban@epfl.ch
André Schiper‡ andre.schiper@epfl.ch
Takuya Katayama† katayama@jaist.ac.jp
†**Japan Advanced Institute of Science and Technology**
Graduate School of Information Science
1-1 Asahidai, Tatsunokuchi, Ishikawa 923-1292, Japan
‡**École Polytechnique Fédérale de Lausanne**
Faculté Informatique et Communications
CH-1015 Lausanne, Switzerland

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 compares the latency of two consensus algorithms designed for the asynchronous model with failure detectors: the Paxos algorithm and the Chandra-Toueg algorithm. We varied the number of processes which take part in the execution. 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. We determined the latency by measurements on a cluster of PCs interconnected with a 100 Mbps Ethernet network. We found that the Paxos algorithm is more efficient than the Chandra-Toueg algorithm when the process that coordinates the first round of the protocol crashes. The two algorithms have almost the same performance in all other cases.

Keywords: Paxos consensus algorithm, Chandra-Toueg consensus algorithm, performance comparison

*Research supported by a grant from the CSEM Swiss Center for Electronics and Microtechnology, Inc., Neuchâtel.

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, 12]. 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*.

Nevertheless, a few papers have tried to analyze the performance of agreement protocols: [10] and [11] analyze quantitatively four different total order broadcast algorithms using discrete event simulation; [23] uses a contention-aware metric to compare analytically the performance of four total order broadcast algorithms; [8, 7] analyze atomic broadcast protocols for wireless networks, deriving assumption coverage and other performance related metrics; [13] presents an approach for probabilistically verifying a synchronous round-based consensus protocol; [18] evaluates the performability of a group-oriented multicast protocol; [21] analyzes the latency of a consensus algorithm by simulation, in order to compare under different implementations of failure detectors; and [9] analyzes the latency of a consensus algorithm focusing on the quality of service offered by the failure detectors. In all these papers, except

for [9, 21, 13, 18], the protocols are only analyzed in failure free runs. This only gives a partial and incomplete understanding of their quantitative behavior.

A detailed quantitative performance analysis of agreement protocols represents a huge work. Where should such a work start? Most agreement problems are related to the abstract consensus problem [6, 22, 15], defined over a set of processes: each process in this set proposes a value initially, and the processes must decide on the *same* value, chosen among the proposed values. For this reason, it seems natural to start by a performance analysis of consensus algorithms, and to extend the work of [9]. This is the goal of this paper, which compares the performance of two consensus algorithms. We chose two algorithms with rather similar characteristics: Paxos [19, 17] is a consensus algorithm of the *leader-based paradigm* [3] using the failure detector Ω [5], and the Chandra-Toueg consensus algorithm [6] is based on the *rotating coordinator paradigm* and uses the failure detector $\diamond\mathcal{S}$. Common points of these algorithms are that they work in asynchronous systems (extended with some oracles), that they need that a majority of processes is correct, and that they have a similar structure: they execute a sequence of rounds whereby each round has a leader which tries to impose a decision.

In our quantitative analysis, we determined the latency of the consensus algorithms, i.e., the time that elapses from the the beginning of the algorithm until the first process decides. We compared the algorithms by comparing their latencies in a variety of benchmarks. We varied the number of processes which take part in the execution. 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. We implemented the algorithms and ran measurements on a cluster of PCs interconnected with a 100 Mbps Ethernet network. We found that the Paxos algorithm is more efficient than the Chandra-Toueg algorithm when the process that coordinates the first round of the protocol crashes. The two algorithms have almost the same performance in all other cases.

The remainder of this paper is organized as follows. Section 2 defines the distributed system model, the consensus problem and unreliable failure detectors. Section 3 reviews the Paxos and Chandra-Toueg consensus algorithms. In Section 4, we present the objective of the experiments, as well as the hardware and the software environment. We present and discuss our results in Section 5. Finally, Section 6 concludes the paper.

2 System Model & Definitions

2.1 System model

A distributed system is modeled as a set of processes $\{p_1, p_2, \ldots, p_n\}$ that communicate by exchanging messages. We assume this system to be asynchronous, i.e., no bounds exist on either communication delays or the relative speed of processes. We further assume that every pair of processes is connected by quasi-reliable communication channels [2], whereby quasi-reliable means that a channel (1) never loses a message (unless the sender or the receiver crashes), (2) never corrupts a message, and (3) never generates spurious messages. We consider that processes may only fail by crashing, and that a crashed process never recovers.

2.2 The consensus problem

The consensus problem is defined over a set of processes. Each process executes two primitives: $propose(v_i)$ by which a process proposes its initial value, and $decide(v)$ by which a process decides a value. The decision must satisfy the following conditions:

(TERMINATION) Every correct process eventually decides.

(VALIDITY) If a process decides v, v is the initial value of some process.

(AGREEMENT) Two correct processes can not decide differently.

Consensus does not have a deterministic solution in the asynchronous model with process crashes [14], but this impossibility result can be circumvented by augmenting the system with an oracle. The algorithms analyze in this paper assume unreliable failure detectors [6, 5]. These oracles are discussed in Section 2.3.

2.3 Failure Detectors

A failure detector is a module attached to every process which gives some information about which process has crashed in the system. Failure detectors are unreliable, i.e., at a given time they can give an incorrect view of the system: correct processes might be suspected as crashed and crashed processes might be trusted as correct.

The Chandra-Toueg consensus algorithm uses the $\diamond\mathcal{S}$ failure detector. This failure detector outputs a list

of processes that it suspects to have crashed. It has to fulfill the following two properties:

(STRONG COMPLETENESS) Every incorrect(crashed) process is eventually suspected forever by every correct processes.

(EVENTUAL WEAK ACCURACY) There is a time t after which one correct process is no more suspected.

A variety of techniques exist for implementing such a failure detector. We chose a 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. We chose T sufficiently high to ensure that the implementation meets the properties of $\diamond\mathcal{S}$ failure detectors.

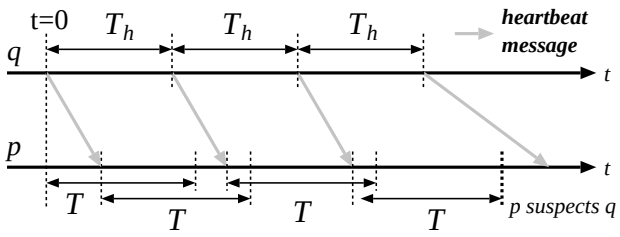

Figure 1. Heartbeat failure detection.

The Paxos consensus algorithm uses the Ω failure detector. An Ω failure detector outputs only one trusted process (in contrast to $\diamond\mathcal{S}$ which outputs a list of suspected processes). It has to fulfill the following property [5]:

(EVENTUAL LEADER) There is a time t after which exactly one correct process p_l is always trusted by every correct process.

We implemented the Ω failure detector using a $\diamond\mathcal{S}$ failure detector: the leader process output by the Ω failure detector is simply the process with the smallest identifier which is not suspected by the $\diamond\mathcal{S}$ failure detector [5].

3 The consensus algorithms

We compared the Paxos algorithm to the Chandra-Toueg consensus algorithm with the failure detector $\diamond\mathcal{S}$. We chose these algorithms because they have a lot of characteristic in common, and hence it is relatively easy to define a meaningful comparison. Common points between Paxos and Chandra-Toueg are that (1) an execution of these algorithms consists of asynchronous rounds; (2) each round has a leader process which tries to impose a decision on all processes, and if this fails, a new round is started with a new leader, (3) they require a majority of correct processes, (4) they work in the asynchronous system model, extended with $\diamond\mathcal{S}$ and Ω failure detectors, respectively (and these classes of failure detectors are equivalent [5]).

3.1 Paxos

Paxos [19, 17, 3] is a consensus algorithm of the *leader-based paradigm* [3] using the failure detector Ω [5]. This algorithm is actually called the "single-decree synod" protocol and its variation for multiple consensus is called the "multi-decree parliament" protocol [17]. We use the name Paxos for the single-decree protocol.

At any time, each process considers one process as the leader process. The leader process is decided by the failure detector Ω (see Section 2.3). This implies that if the leader process crashes, the failure detector Ω will eventually select another leader process.

If a process considers itself leader, it starts a new round. Process p_i will use round numbers $i, i+n, i+2n, \ldots$, in this order. This scheme ensures that all round numbers are unique (this is a requirement of the leader-based paradigm). Of course, several processes might consider themselves leader at the same time, so different processes might execute rounds with different numbers in parallel.

Each round consists of two phases (see Figure 2). In the first phase, the leader verifies whether all other participants have already decided some value with a higher round than the leader's one. In the second phase, it tries to decide a value.

First phase The leader process p_l sends a *prepare request* with the round number r_{p_l} to all other processes. The purpose of sending the prepare request is to ensure that other processes never again accept any round number less than r_{p_l}. When p_i receives the request, it ac-

cepts the proposal if (1) the proposal with the highest round number accepted so far has a round number less than r_{p_l} and (2) has not yet responded to a request with a round number greater than r_{p_l}. If p_i accepts the proposal, it sends an ACK message to the leader with its own current estimate of the decision and the number of the round when the estimate was updated. Otherwise (if p_i receives a prepare request with a round number less than the round number of any of the previously accepted requests) p_i rejects the proposal and sends a NACK message to the leader.

The leader waits for a majority of responses. If all responses are ACKs, the leader updates its current estimate of the decision value to the most recent estimate in the ACK messages. Otherwise, if it receives at least one NACK, the round aborts immediately and the leader moves to next round.

Second phase The leader sends an *accept request* to all other participants with the updated estimate. This request is handled the same way as the prepare request: the other processes might update their estimates and reply with an ACK or a NACK message (however, the ACK message contains only the round number, not the estimate). If the leader receives only ACKs, it sends a *decision message* with the decision value using reliable broadcast. Upon receipt of this message, the other processes decide immediately. If the leader receives at least one NACK, it aborts the round and moves to the next round.

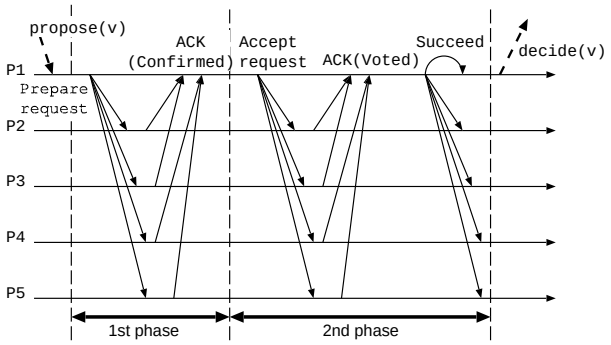

Figure 2. Phases in a round of the Paxos algorithm.

Figure 3 illustrates a *good run* of the algorithm. We define a good run as a run in which the leader does not crash. In a good run, the algorithm solves consensus directly with only the second phase (see Figure 3), because it does not need to check whether all other participants already decided some value with a higher round than the leader's one (round 0). However, the leader has to execute two phases, if it has aborted more than one round already.

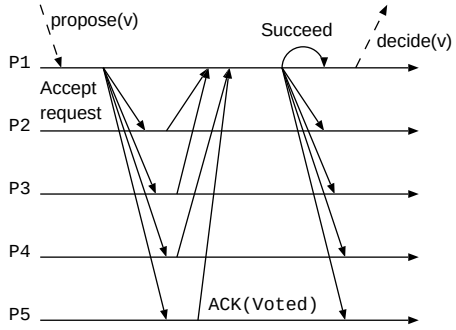

Figure 3. A good run of the Paxos algorithm (p_1 is leader for round 0).

There is an easy optimization to Paxos [3, 4]. Normally, each round is divided into two phases. However, we can omit the first phase (like Figure 3). The leader process p_l can safely execute operations of the second phase to decide a value in the first round without the first phase. In other rounds, we can modify the scenario as follows: (1)When some process without the leader responds a NACK, the process also sends it with the latest round number among they have accepted proposals as a value max to the leader, and then, (2)p_l aborts the round by receiving a NACK message and immediately increments the round number r_{p_l} to $x \times p_l > max$. Therefore, p_l move to the round "$x \times p_l > max$" when it aborts a round. This modification can save the first phase and p_l can solve consensus with only the second phase. In this case, consensus can be reached significantly faster than the standard one. We run the experiments with the optimized version of Paxos.

3.2 The Chandra and Toueg Algorithm

The Chandra-Toueg algorithm [6] is based on the rotating coordinator paradigm and uses the failure detector $\diamond\mathcal{S}$. In this paradigm, each process executes rounds with numbers 1, 2, etc. The leader process (called coordinator in [6]) is decided by the following expression:

$$c_{r_p} \leftarrow (r_p \; mod \; n) + 1$$

where r_p is the round number, and $p_{c_{r_p}}$ is the leader process of that round, and n the number of participants in the consensus.

Each round is composed of four phases (see Figure 4). In the first phase, every process sends its current estimate for the decision to the current leader c_{r_p}, along with the number of the round when the estimate was last updated. In the second phase, when the leader gets a majority of such estimates, it selects the newest estimate and sends it to all the processes. In the third phase, when p_i receives the estimate from the leader, sends an acknowledgment(*ACK*) to the leader to indicate that it adopted the estimate. If p_i's failure detector suspects the leader before receiving the estimate from the leader otherwise, sends a denial(*NACK*) to the leader. Finally, in the fourth phase, the leader waits for ACKs from a majority of processes or one NACK. If it received a majority of ACKs, it decides on its estimate and reliably broadcasts the decision. When the others receive the decision, they immediately decide. On the other hand, if the leader received a NACK, it moves on to the next round without deciding or sending messages.

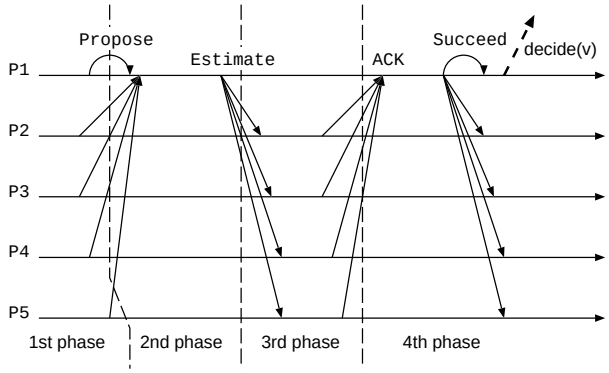

Figure 4. Phases in a round of the Chandra-Toueg algorithm

If there are no failure suspicions, the algorithm terminates in one round. Similarly to Paxos, we can omit the first phase of the first round to speed up the algorithm.

4 Experiments

4.1 Objective

Latency and throughput are meaningful measures of the performance of algorithms. Roughly speaking, *latency* measures the time that elapses 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 definition of latency is a reasonable performance measure for the following reason. Consider a service replicated for fault tolerance using active replication [20]. Clients of this service send their requests to the server replicas using Atomic Broadcast [16] (which guarantees that all replicas see all requests *in the same order*). Atomic Broadcast can be solved by using a consensus algorithm [6]: 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 the 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.

4.2 Scenarios and parameters

The latency of a consensus algorithm varies for different numbers of processes (n). In each of the benchmarks, we changed n from 2 to 10. However, given n, the latency also depends 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 scenarios:

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 suspect 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, and thus failures are detected relatively slowly, or the heartbeat period T_h must be shortened, and thus the network load increases.

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 performed experiments for each of these scenarios.

In the experiments, each process proposes a single integer value for the consensus algorithm. For this reason, all the messages sent by the consensus algorithm are short (less than 100 bytes), and thus the delay experienced by messages varies less than if we had run consensus with long proposals.

4.3 Environment

We used a cluster of 12 PCs running Red Hat Linux 7.2 (kernel 2.4.9) to run the experiments. Each node has a Intel Pentium III 766 MHz processor 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) on top of the Neko development framework; Neko is a platform for prototyping and simulating distributed algorithms [24]. All messages were transmitted using TCP/IP. Connections between each pair of machines were established at the beginning of the test.

5 Results

5.1 No process crashes

Figure 5 shows the mean latency of the two algorithms versus the number of processes, in the scenario in which no processes crash. The graph also shows the 95% confidence interval for the mean latency. We performed measurements with 2, 3, ..., 10 processes.

The results show that the latency of the two algorithms are largely identical.

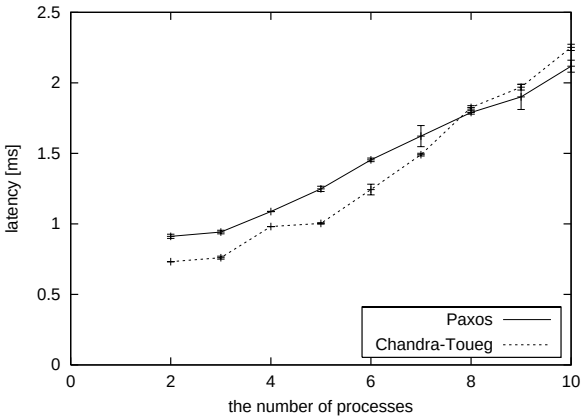

Figure 5. Latency of consensus vs. number of processes, with no crashes.

5.2 One crashed process

We also performed measurements in the scenario where one process is crashed before the measurements. The number of processes varies from 3 to 10 (measurements with 2 processes was omitted as the algorithms do not tolerate any crashes in this case). We also varied which process we crashed. We found that the latency was higher if the first leader process crashed than if any of the others crashed. Figure 6 shows this case. We can see that the Paxos algorithm is more efficient here than the Chandra-Toueg algorithm. We observed nearly identical latencies for a given algorithm and number of processes if any of the other processes (different from the first leader) crashed. These results are shown in Fig. 7.

5.3 Discussion

In the scenario where no processes crash, we observed a largely identical latency. The reason is that both algorithms need only the first round to decide the value. Also, first phase of the first round is optimized out in the case of both algorithms. Hence the algorithms generate exactly the same number of messages and have the same interactions (see Fig. 3, 4).

The same argument explains why the latencies are identical when a process which is not leader (any except the first process) crashes: the algorithms can still decide in the first round, and the pattern of message exchange is the same, as the leaders always collect acknowledgements from only a majority of all processes.

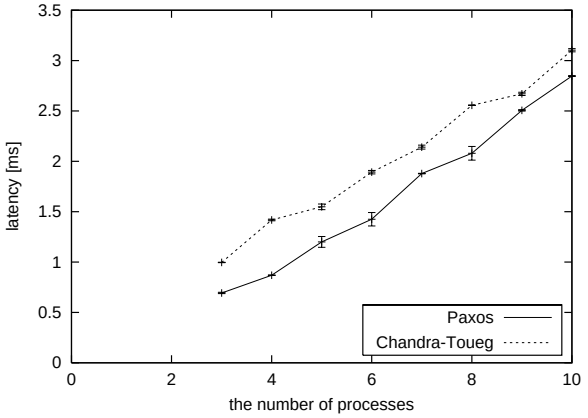

Figure 6. Worst case latency of consensus in the case of one crash (crash of the leader).

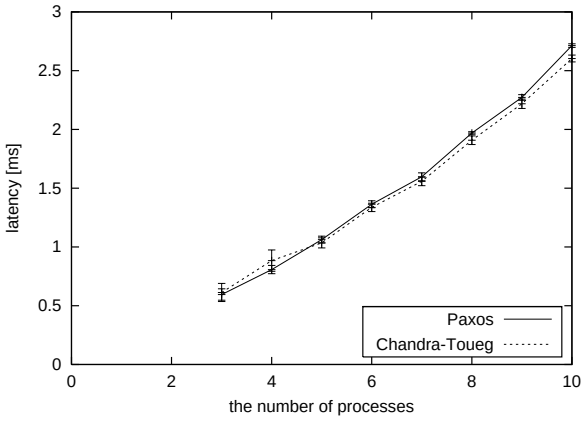

Figure 7. Latency of consensus, with the crash of one process which is not leader.

We observed a difference when the first leader process crashed. In this case, the Chandra-Toueg algorithm must execute four phases after the first round, while the Paxos algorithm can reach a decision within a smaller number of rounds (see Sect. 3.1). We can consider that these factors lead to the result. Thus, Paxos has an advantage when crashed processes are present.

The performance of consensus algorithms is strongly impacted by the number of messages. Therefore, the Paxos algorithm is more efficient than the Chandra-Toueg algorithm when the leader process immediately crashes after starting the round of the consensus algorithm.

6 Conclusion

In this work, we compared the Paxos and Chandra-Toueg consensus algorithms from the point of view of performance.

Our results show that Paxos is more efficient than Chandra-Toueg if the first leader process crashed, but, both algorithms have almost same latencies in any other cases.

As future work, we will compare the performance of the algorithms in runs with no failures but with (wrong) failure suspicions, i.e., when failure detectors mistakenly suspect correct processes from time to time. This happens when the timeouts used in the failure detector implementation are small. Such wrong failure suspicions increase the latency of the algorithms, as the algorithms are forced to take more rounds until completion. On the other hand, small failure detection timeouts may lead to a faster reaction to failures. We plan to study this detection time / wrong suspicions tradeoff in detail.

Also, we will measure the performance of these algorithms with more than 10 nodes, such as 500 nodes or more, as we think that it is important to consider applying consensus in realistic large scale distributed applications.

References

[1] M. Barborak, M. Malek, and A. Dahbura. The consensus problem in distributed computing. *ACM Computing Surveys*, 25(2):171–220, June 1993.

[2] A. Basu, B. Charron-Bost, and S. Toueg. Solving problems in the presence of process crashes and lossy links. Technical Report TR96-1609, Cornell University, USA, Sept. 1996.

[3] R. Boichat. *Reliable and Total Order Broadcast in the Crash Recovery Model*. PhD thesis, École Polytechnique Fédérale Lausanne, Switzerland, Nov. 2001.

[4] R. Boichat, P. Dutta, S. Frolund, and R. Guerraoui. Deconstructing paxos. Technical Report DSC-200106, EPFL, Lausanne, Switzerland, Jan. 2001.

[5] 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.

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

[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)*, pages 209–216, Magdeburg, Germany, May 2001.

[8] 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)*, pages 247–256, Albuquerque, NM, USA, Nov. 2000.

[9] A. Coccoli, P. Urbán, A. Bondavalli, and A. Schiper. Performance analysis of a consensus algorithm combining Stochastic Activity Networks and measurements. In *Proc. Int'l Performance and Dependability Symp.*, Washington, DC, USA, June 2002. Accepted for publication.

[10] F. Cristian, R. de Beijer, and S. Mishra. A performance comparison of asynchronous atomic broadcast protocols. *Distributed Systems Engineering Journal*, 1(4):177–201, June 1994.

[11] F. Cristian, S. Mishra, and G. Alvarez. High-performance asynchronous atomic broadcast. *Distributed System Engineering Journal*, 4(2):109–128, June 1997.

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

[13] H. Duggal, M. Cukier, and W. Sanders. Probabilistic verification of a synchronous round-based consensus protocol. In *Proceedings of The 16th Symposium on Reliable Distributed Systems (SRDS '97)*, pages 165–174, Washington - Brussels - Tokyo, Oct. 1997. IEEE.

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

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

[16] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. TR 94-1425, Dept. of Computer Science, Cornell University, Ithaca, NY, USA, May 1994.

[17] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[18] 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*, pages 63–72, Durham, NC, USA, Oct. 1995.

[19] R. D. Prisco, B. Lampson, and N. Lynch. Revisiting the PAXOS algorithm. *Theoretical Computer Science*, 243(1–2):35–91, July 2000.

[20] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.

[21] 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)*, pages 137–145, Seoul, Korea, Dec. 2001.

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

[23] 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.

[24] 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.