Comparing Distributed Consensus Algorithms[*]

Péter Urbán
Japan Advanced Institute of Science and Technology (JAIST)
1-1 Asahidai, Tatsunokuchi, Nomi, Ishikawa 923-1292, Japan
Email: urban@jaist.ac.jp

André Schiper
École Polytechnique Fédérale de Lausanne (EPFL)
CH-1015 Lausanne, Switzerland
Email: andre.schiper@epfl.ch

Abstract

Protocols that solve agreement problems are essential building blocks for fault tolerant distributed systems. While many protocols have been published, little has been done to analyze their performance, especially the performance of their fault tolerance mechanisms. In this paper, we compare two consensus algorithms with different communication schemes: one is centralized and the other decentralized. The elements of the simulation study form a generic methodology for evaluating consensus algorithms. The results show that the centralized algorithm performs better in some environments, in spite of the fact that the decentralized algorithm finishes in fewer communication steps. The reason is that it generates less contention.

Key words: simulation tools and techniques, Java-based simulation, distributed consensus, benchmarks

1 Introduction

Agreement problems — such as consensus, atomic broadcast or atomic commitment — are essential building blocks for fault tolerant distributed applications, including transactional and time critical applications. They have been extensively studied in various system models, and many protocols solving these problems have been published [1, 2]. However, the focus has been on analyzing the safety and liveness properties of protocols, and little has been done to analyze their *performance*. Also, most papers focus on analyzing failure free runs, thus neglecting the performance aspects of failure handling. In our view, the limited understanding of performance aspects, in both failure free scenarios and scenarios with failure handling, is an obstacle for adopting such protocols in practice. This paper represents a starting point for such studies, by focusing on the consensus problem, a problem related to most other agreement problems [3].

We present a comparison study of two well-known consensus algorithms. One algorithm (due to Chandra and Toueg [4]) uses a centralized communication pattern, while the other (due to Mostéfaoui and Raynal [5]) uses a decentralized communication pattern. Other aspects of the algorithms are very similar.

The paper also proposes a generic methodology for evaluating consensus algorithms. We next describe the elements of this methodology. The two consensus algorithms are analyzed in a system in which processes send atomic broadcasts to each other. Since the atomic broadcast algorithm that we use [4] leads to the execution of a sequence of consensus to decide the delivery order of messages, evaluating the performance of atomic broadcast is a good way of evaluating the performance of the underlying consensus algorithm in a realistic usage scenario. In our study, the atomic broadcast algorithm uses either of the two consensus algorithms. We study the system using simulation, which allows us to compare the algorithms in a variety of different environments. We model message exchange by taking into account contention on the network and the hosts, using the metrics described in [6, 7]. We model failure detectors in an abstract way, using the quality of service (QoS) metrics proposed by Chen et al. [8]. We compare the algorithms using the benchmarks proposed in [7, 9] (which are stated in terms of the system under study, i.e., atomic broadcast). Our performance metric for atomic broadcast is *early latency*, the time that elapses between the sending of a message m and the earliest delivery of m. We use symmetric workloads. We evaluate both (1) the steady state latency in runs with neither failures nor suspicions and (2) the transient latency after a process crash.

The centralized algorithm requires three communication steps under the most favorable conditions, while the decentralized one needs only two. Hence it is often believed that the decentralized algorithm is more efficient. Our results show that, contrary to these expectations, the centralized algorithm performs better under a variety of settings. The reason is that the centralized algorithm generates less contention, which often offsets the costs of the additional communication step.

As the problem of choosing between a decentralized and a centralized variant of an agreement algorithm recurs often in distributed systems (e.g., two and three phase commit protocols have variants of both kinds), we expect that our results are useful in other settings than the ones assumed in this paper.

The rest of the paper is structured as follows. Section 2 presents related work. Section 3 defines the system model and the agreement problems used in this paper. We introduce the algorithms in Section 4. The methodology is presented next: Section 5 describes the benchmarks we used, followed by our simulation model for the network and the

[*]Research supported by the Japan Society for the Promotion of Science, a MEXT Grant-in-Aid for JSPS Fellows, the Swiss National Science Foundation, and the CSEM Swiss Center for Electronics and Microtechnology, Inc., Neuchâtel.

failure detectors in Section 6. Our results are presented in Section 7, and the paper concludes in Section 8.

2 Related work

Most of the time, consensus algorithms are evaluated using simple metrics like time complexity (number of communication steps) and message complexity (number of messages). This gives, however, little information on the real performance of those algorithms. A few papers provide a more detailed performance analysis. Ref. [10] compares the impact of different implementations of failure detectors on the Chandra-Toueg consensus algorithm; Ref. [11] and [12] analyze the latency of the same algorithm, concentrating mostly on the effect of wrong failure suspicions; All these papers consider only isolated consensus executions, which are a special case of our workloads, corresponding to a very low setting for the throughput. Other papers [9,13] consider a consensus algorithm embedded in an atomic broadcast algorithm, and also consider more complex workloads, but they do not aim at comparing consensus algorithms. Note also that the performance of atomic broadcast algorithms is studied more extensively in the literature than the performance of consensus algorithms (see [7] for a summary).

Most papers on the performance of agreement algorithms only consider failure free executions (our normal-steady faultload), which only gives a partial and incomplete understanding of the behavior of the algorithms. We only note a few interesting exceptions here. The transient effects of a crash are studied in [9, 10, 14], but the faultload in [10, 14] is different from our crash-transient faultload. Ref. [10] assumes that the crash occurs at the worst possible moment during execution, leading to the worst case latency. In contrast to our faultload, this faultload requires a detailed knowledge of the execution, which is only available if one considers very simple workloads. The other paper [14] measures the latency of the group membership service used by the algorithm to tolerate crash failures;[1] it is thus based on an implementation detail of the algorithm, unlike our faultload.

There are other faultloads describing process crashes and their detection, studying steady-state performance in the presence of (1) crashes (e.g., [9]) and (2) wrong suspicions [9, 11, 12]. We do not consider such faultloads in this paper because the steady-state performance of the two consensus algorithms already shows significant differences with the normal-steady faultload.

3 Definitions

We consider a widely accepted model for distributed systems. It consists of processes that communicate only by exchanging messages. The system is asynchronous, i.e., we make no assumptions on its timing behavior. The network is quasi-reliable: it does not lose, alter nor duplicate messages. In practice, this is easily achieved by retransmitting lost messages. We consider that processes only fail by crashing. Crashed processes do not send any further messages.

The consensus algorithms used in this paper use *failure detectors* to tolerate process crashes. A failure detector maintains a list of processes it suspects to have crashed.To make sure that the consensus algorithms terminate, we need some assumptions on the behavior of the failure detectors ($\Diamond\mathcal{S}$; see [15]), easily fulfilled in practice [13].

We next give informal definitions of the agreement problems needed for understanding this paper; see [4, 16] for more formal definitions.

In the consensus problem, each process proposes an initial value. Uniform consensus (considered here) ensures that all processes decide the same value, which is one (any one) of the proposals.

Atomic broadcast is defined in terms of two primitives called *A-broadcast*(m) and *A-deliver*(m), where m is some message. Uniform atomic broadcast (considered here) guarantees that (1) if a message is A-broadcast by a process, then all correct processes eventually A-deliver it, and (2) all processes A-deliver messages in the same order.

4 Algorithms

This section sketches the two consensus algorithms, concentrating on their common points and their differences. We then introduce the atomic broadcast algorithm built on top of consensus. More detailed descriptions of the algorithms can be found in [7].

The consensus algorithms. For solving consensus, we use the Chandra-Toueg $\Diamond\mathcal{S}$ algorithm [4] and the Mostéfaoui-Raynal $\Diamond\mathcal{S}$ algorithm [5]. Henceforth, we shall refer to the algorithms as *CT algorithm* and *MR algorithm*, respectively.[2]

Common points. The algorithms share a lot of assumptions and characteristics, which makes them ideal candidates for a performance comparison. In particular, both algorithms are designed for the asynchronous model with $\Diamond\mathcal{S}$ failure detectors (see Section 3). Both tolerate $f < n/2$ crash failures. Both are based on the rotating coordinator paradigm: each process executes a sequence of asynchronous rounds (i.e., not all processes necessarily execute the same round at a given time t), and in each round a process takes the role of *coordinator* (p_i is coordinator for rounds $kn + i$). The role of the coordinator is to impose a decision value on all processes. If it succeeds, the consensus algorithm terminates. It may fail if some processes

[1]Certain kinds of Byzantine failures are also injected.

[2]We also use these names to refer to the atomic broadcast algorithm used with the corresponding consensus algorithm if no confusion arises from doing so.

suspect the coordinator to have crashed (whether the coordinator really crashed or not). In this case, a new round is started.

Execution of a round. In each round of a consensus execution, the CT algorithm uses a centralized communication scheme (see Fig. 1) whereas them MR algorithm uses a decentralized communication scheme (see Fig. 2). We now sketch the execution of one round in each of the two algorithms. We suppose that the coordinator is not suspected. Further details of the execution are not necessary for understanding the rest of the paper. The interested reader is referred to [7].

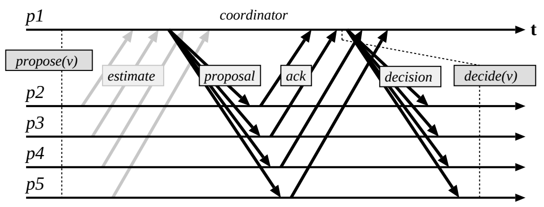


Figure 1. Example run of the CT consensus algorithm.

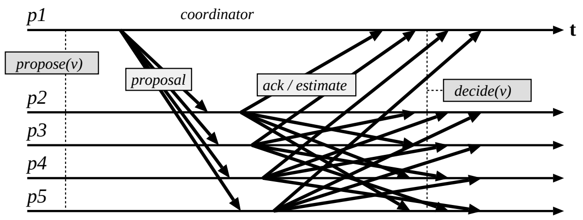


Figure 2. Example run of the MR consensus algorithm.

- In the CT algorithm, the coordinator first gathers estimates for the decision value from a majority of processes (*estimate* messages in Fig. 1) to choose its proposal from. This phase is only necessary in the second round and later; this is why the messages are grayed out in Fig. 1.
- In both algorithms, the coordinator sends a proposal to all (*proposal* messages in Fig. 1 and 2).
- Upon receiving the proposal, processes send an acknowledgment (*ack* messages). In the CT algorithm, acks are sent to the coordinator only. In the MR algorithm, the ack is sent to all. Moreover, processes in the MR algorithm piggyback their current estimate on the *ack* message, in order to allow the coordinator of the next round to choose a proposal. This is why the MR algorithm does not require a separate phase to send estimate messages. Piggybacking estimates in a similar way is not possible in the CT algorithm, as the coordinator of the next round does not receive the *ack* messages.
- Upon receiving acks from a majority of processes, the coordinator (in the CT algorithm) and all processes (in the MR algorithm) decide. The coordinator in the CT algorithm needs to send its decision to all (decision message in Fig. 1). This is not necessary in the MR algorithm, because each process decides independently.

Crashes are handled in the following way: if a process suspects the coordinator, it sends a negative ack to the coordinator, which results in a new round with another coordinator.

The Chandra-Toueg atomic broadcast algorithm. In this algorithm [4], a process executes A-broadcast by sending a message to all processes.[3] When a process receives such a message, it buffers it until the delivery order is decided. The delivery order is decided by a sequence of consensus numbered 1, 2, etc. The value proposed initially and the decision value of each consensus are *sequences of message identifiers*. The delivery order is given by the concatenation of the sequences coming from consensus 1, 2, etc.

The algorithm inherits the system model and any fault tolerance guarantees from the underlying consensus algorithm. We use this atomic broadcast algorithm with both the CT and MR consensus algorithms.

5 Benchmarks

This section describes our benchmarks [7, 9], consisting of performance metrics, workloads and faultloads. In order to get meaningful results, we state the benchmarks in terms of the system under study (processes sending atomic broadcasts) rather than in terms of the component under study (consensus).

Performance metrics. Our main performance metric is the *early latency* of atomic broadcast [7, 9]. Early latency L is defined for a single atomic broadcast as follows. Let *A-broadcast(m)* occur at time t_0, and *A-deliver(m)* on p_i at time t_i, for each $i = 1, \ldots, n$. Then latency is defined as the time elapsed until the first A-delivery of m, i.e., $L \overset{\text{def}}{=} (\min_{i=1,\ldots,n} t_i) - t_0$. In our study, we compute the mean for L over a lot of messages and several executions.

This metric is meaningful in practice: it reflects the performance of a service replicated using atomic broadcast (see [7] for details).

Workloads.. Latency is always measured under a certain workload. We chose simple workloads: (1) all destination processes send atomic broadcast messages at the same constant rate, and (2) the A-broadcast events come from a Poisson stochastic process. We call the overall rate of atomic broadcast messages *throughput*, denoted by T. In general,

[3]This message is sent using reliable broadcast; see [7] for a discussion.

we determine how the latency L depends on the throughput T.

Faultloads. The faultload is the part of the workload that describes failure-related events that occur during an experiment [17]. We concentrate on (1) crash failures of processes, and (2) the behavior of unreliable failure detectors. We evaluate the latency of the atomic broadcast algorithms with two different faultloads.

Normal-steady faultload. With this faultload, we have neither crashes nor wrong suspicions in the experiment. We measure latency after the system reaches its steady state (a sufficiently long time after startup). Parameters that influence latency under this faultload are the algorithm (A), the number of processes (n) and the throughput (T).

Crash-transient faultload. With this faultload, we inject a crash after the system reached a steady state. After the crash, we can expect a halt or a significant slowdown of the system for a short period. We would like to capture how the latency changes in atomic broadcasts directly affected by the crash. Our faultload definition represents the simplest possible choice: we determine the latency of an atomic broadcast sent at the moment of the crash (by a process other than the crashing process). Of course, the latency of this atomic broadcast (L) may depend on the choice for the sender process (p) and the crashing process (q). In order to reduce the number of parameters, we consider the worst case, i.e., the case that increases latency the most: $L_{crash} \overset{\text{def}}{=} \max_{p,q \in P} L(p, q)$.

Parameters that influence latency under this faultload are the algorithm (A), the number of processes (n) and the throughput (T), just as under the normal-steady faultload. An additional parameter describes how fast failure detectors detect the crash. This parameter is discussed in Section 6.

6 Simulation models

Our approach to performance evaluation is simulation, which allowed for more general results as would have been feasible to obtain with measurements in a real system (we can use a parameter in our network model to simulate a variety of different environments). We used the Neko prototyping and simulation framework [18], written in Java, to conduct our experiments.

Modeling the execution environment. We now describe how we modeled the transmission of messages. We use the model of [6, 7], inspired from simple models of Ethernet networks, and validated in [7]. 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. Both a host and the network itself can be a bottleneck. These two kinds of resources appear in the model (see Fig. 3): the network resource (shared among all processes) represents the transmission medium, and the CPU resources (one per process) represent the processing performed by the network controllers and the layers of the networking stack, during the emission and the reception of a message (the cost of running the algorithm is negligible). A message m transmitted for process p_i to process p_j uses the resources (1) CPU_i, (2) network, and (3) CPU_j, in this order. Message m is put in a waiting queue before each stage if the corresponding resource is busy. The time spent on the network resource is one time unit. The time spent on each CPU resource is λ time units; the underlying assumption is that sending and receiving a message has a roughly equal cost.

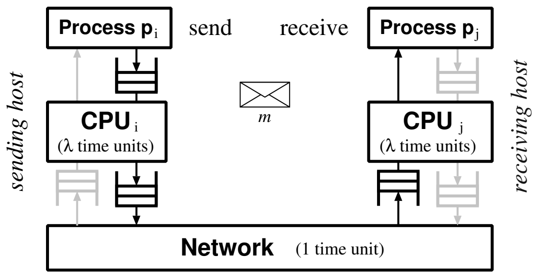


Figure 3. Transmission of a message in our network model.

The λ parameter ($0 \leq \lambda$) shows the relative speed of processing a message on a host compared to transmitting it over the network. Different values model different networking environments. We conducted experiments with a variety of settings for λ. Also, we conducted experiments with two variants of the model: one that supports multicast messages and one that supports only unicast messages. In the latter variant, multicast messages are sent as several unicast messages, and thus put a higher load on the network and the sending host.

Modeling failure detectors. One approach to examine the behavior of a failure detector is implementing it and using the implementation in the experiments. However, it is not justified to model the failure detector in so much detail, as other components of the system are modeled much more coarsely. We built a more abstract model instead, using the notion of quality of service (QoS) of failure detectors introduced in [8]. Only one of the QoS metrics is relevant with our faultloads: the detection time (T_D), which measures the time that elapses from the crash of the monitored process until the monitoring process starts suspecting p permanently. The definition is illustrated in Fig. 4.

To keep our model simple, we assume that the detection time T_D is the same constant on all processes. This choice only represents a starting point, as we are not aware of any previous work we could build on (apart from [8] that makes similar assumptions). We will refine our models as we gain more experience.

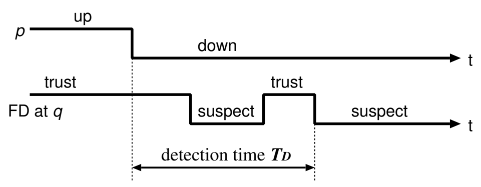


Figure 4. Quality of service metric of failure detectors. Process q monitors process p.

Finally, note that this abstract model for failure detectors neglects that failure detectors and their messages put a load on system components. This simplification is justified in a variety of systems, in which a rather good QoS can be achieved with failure detectors that send messages infrequently.

7 Results

We now present our results for both faultloads and a variety of network models. We obtained results for a variety of representative settings for λ: 0.1, 1 and 10. The settings $\lambda = 0.1$ and 10 correspond to systems where communication generates contention mostly on the network (at $\lambda = 0.1$) and the hosts (at $\lambda = 10$), respectively, while 1 is an intermediate setting. Due to lack of space, we only present results for $\lambda = 1$ here; see [7] for the full set of results. We obtained results with both the point-to-point and the broadcast variant of the network model.

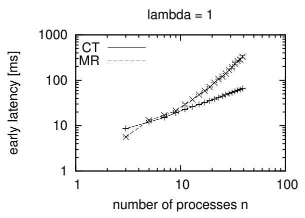
Most graphs show latency vs. throughput (some show latency vs. the number of processes). The rightmost point shown corresponds to the highest throughput at which each process is still able to deliver all messages. We set the time unit of the network simulation model to 1 ms, to make sure that the reader is not distracted by an unfamiliar presentation of time/frequency values. Any other value could have been used. 95% confidence intervals are shown.

Normal-steady faultload, scalability study (Fig. 5(a)). In each graph, latency is shown as a function of the number of processes n.[4] Atomic broadcast are sent at a very low rate (0.1 requests/s). At this throughput, executions of subsequent atomic broadcasts do not influence each other.

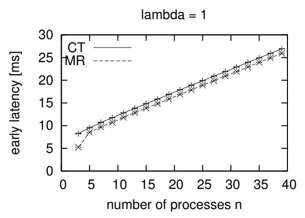
Fig. 5(a) shows the results for the point-to-point model. Logarithmic scales are used on both axes, to visualize a big range of latency and to emphasize small values of n. The graph can be divided into three regions:

- The MR algorithm always performs better at $n = 3$. The reason is that decentralized coordination (MR algorithm) requires one communication step fewer than centralized coordination (CT algorithm; see Figures 1 and 2).
- At high values of n ($n \geq 11$) the MR algorithm performs much worse. The graphs also show that the latency of the CT algorithm scales linearly with n whereas the latency of the MR algorithm scales quadratically: the slopes of the latency curves in the log-log graph are about 1 and 2, respectively. The reason is that the CT algorithm uses $O(n)$ messages, whereas the MR algorithm uses $O(n^2)$ messages, though each process only handles $O(n)$ messages in both algorithms. This makes the MR algorithm network bound at high values of n, and the effect of a quadratic number of messages shows directly.
- At intermediate settings for n, the two algorithms perform roughly the same. The reason is that the higher resource utilization of the network resource starts to show (unlike at $n = 3$) but both algorithms are still CPU bound (unlike at high values of n).

[4]The two algorithms were always run with an odd number of processes. The reason is that the same number of crash failures k ($k = 1, 2, \ldots$) is tolerated if the algorithms are run with $2k+1$ and $2k+2$ processes; thus adding a process to a system with an odd number of processes does not increase the resiliency of the system.



(a) point-to-point model



(b) broadcast model

Figure 5. Latency vs. number of processes with the normal-steady faultload.

The results are different in the broadcast model; see Fig. 5(b) (linear scales are used on both axes). One can see that the MR algorithm offers a slightly lower latency. Moreover, the difference in latency does not depend on n. The reason is that in the broadcast model, the MR algorithm terminates in one communication step fewer, and that the most heavily loaded resources (the network and the CPU of the coordinator) process one message fewer per consensus.

Normal-steady faultload, algorithms under load (Fig. 6 and 7). In each figure, two latency vs. throughput graphs are shown, one for $n = 3$ and one for $n = 7$.
With the point-to-point model (Fig. 6) one can observe two different behaviors:

- The CT algorithm has worse performance at $n = 3$ (shown here) and also at $n = 7$ when $\lambda = 10$ (see [7]). The reason is that the CT algorithm loads the coordinator much more than the MR algorithm: beside providing a proposal and collecting acks, it must also send the decision, as shown in Fig. 1.
- The MR algorithm has worse performance at $n = 7$ when $\lambda = 1$ (shown here) or 0.1 (see [7]). The

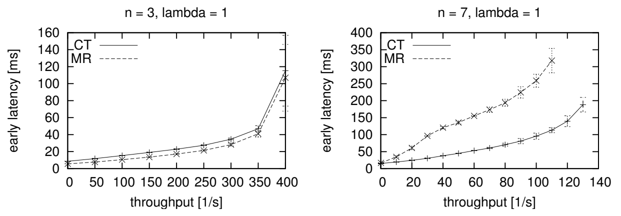


Figure 6. Latency vs. throughput with the normal-steady faultload (point-to-point model).

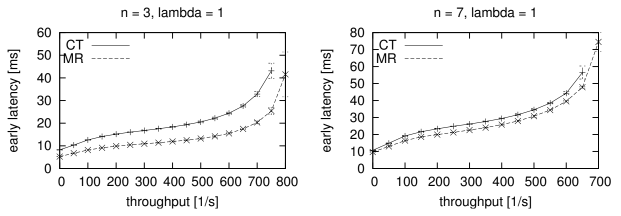


Figure 7. Latency vs. throughput with the normal-steady faultload (broadcast model).

performance difference is roughly proportional to the throughput. The reason for this behavior is that the load on the CPUs does not matter, unlike in the previous case. Instead, the determining factor is that the MR algorithm loads the network more. Also, increasing the throughput leads to higher queuing times in the network buffers of the model (see Section 6).

With the broadcast model (Fig. 7) the MR algorithm performs better at any load. The reason is that in the broadcast model, the most heavily loaded resources (the network and the CPU of the coordinator) process one message fewer per consensus.

Crash-transient faultload (Fig. 8 and 9). With this faultload, we only present the latency after the crash of the coordinator, as this is the case resulting in the highest transient latency (and the most interesting comparison).

The figures show the *latency overhead*, i.e., the latency minus the detection time T_D, rather than the latency. Graphs showing the latency overhead are more illustrative; note that the latency is always greater than the detection time T_D with this faultload, as no atomic broadcast can finish until the crash of the coordinator is detected. The

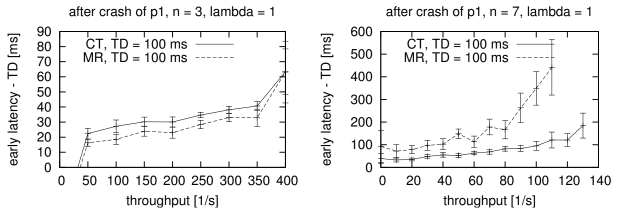


Figure 8. Latency overhead vs. throughput with the crash-transient faultload (point-to-point model).

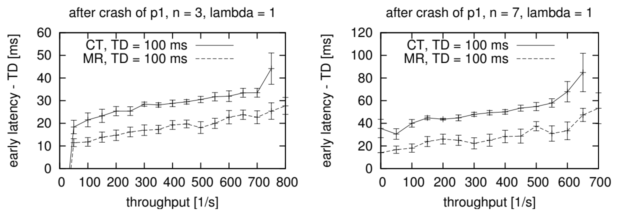


Figure 9. Latency overhead vs. throughput with the crash-transient faultload (broadcast model).

arrangement of the graphs is the same as in Fig. 6 and 7.

We set the failure detection timeout T_D to 100 ms. This choice models a reasonable trade-off for the failure detector: the latency overhead is comparable to T_D, to make sure that the failure detector does not degrade performance catastrophically when a crash occurs. On the other hand, the detection time is high enough (a high multiple of the roundtrip time at low loads) to avoid that failure detectors suspect correct processes.

In the point-to-point model, the results are very similar to the previous set of results, as can be seen by comparing Fig. 6 with Fig. 8. The same observations and explanations apply. The reason is that the differences identified with the normal-steady faultload dominate with the crash-transient faultload as well.

In the broadcast model, the performance of the CT algorithm is much worse, at all settings of n (and λ; see [7]). The reason is that, in addition to the differences observed with the normal-steady faultload, the CT algorithm takes one communication step more (the first phase of the second round; see the gray *estimate* messages in Fig. 1) than the MR algorithm. These *estimate* messages are piggybacked on *ack* messages in the MR algorithm, as discussed in Section 4. The fact that piggybacking is possible is an advantage of the decentralized structure of the MR algorithm.

8 Conclusion

We have investigated two asynchronous consensus algorithms designed for the same system model. Also, both algorithms are based on the rotating coordinator paradigm. The main difference is that, in each round, the CT algorithm uses a centralized communication pattern and the MR algorithm a decentralized one.

We now summarize the results of the study as a list of observations. These observations can be used by implementors when deciding which algorithm to deploy in a given system.

1. In a network model with point-to-point messages only, the MR algorithm performs much worse both when the number of processes n or the load on the system is high.
2. In a network model with broadcast messages, the MR algorithm performs slightly better. The difference in latency does not depend on the number of processes.

3. In a network model with broadcast messages, the MR algorithm reacts much faster to failures.
4. Frequently, only one crash failure needs to be tolerated. If this is the case, i.e., the consensus algorithm runs on three processes, the MR algorithm is a better choice regardless of whether the network supports broadcast messages or not.

Beside the actual performance study, the paper also presented a generic simulation methodology for evaluating consensus algorithms. The main characteristics of the methodology are the following: (1) we consider a sequence of consensus executions, corresponding to a realistic usage scenario; (2) we define repeatable benchmarks, i.e., scenarios specifying the workload, the occurrence of crashes and suspicions, and the performance metrics of interest; (3) the benchmarks include scenarios with crashes and suspicions; (4) we describe failure detectors using quality of service (QoS) metrics; (5) we have a simple one-parameter model for message exchange that accounts for resource contention. The methodology allowed us to obtain rather general results for the two algorithms, as only a small number of parameters were involved in describing the environment.

References

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

[2] X. Défago, A. Schiper, & P. Urbán, Total order broadcast and multicast algorithms: Taxonomy and survey, Japan Advanced Institute of Science and Technology, Ishikawa, Japan, Research Report IS-RR-2003-009, 2003.

[3] R. Guerraoui & A. Schiper, The generic consensus service, *IEEE Trans. on Software Engineering, 27*(1), 2001, 29–41. [Online]. Available: http://lsrwww.epfl.ch/Publications/ById/282.html

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

[5] A. Mostéfaoui & M. Raynal, Solving consensus using Chandra-Toueg's unreliable failure detectors: A general quorum-based approach, *Proc. 13th Int'l Symp. on Distributed Computing (DISC)*, Bratislava, Slovak Republic, 1999, 49–63.

[6] P. Urbán, X. Défago, & A. Schiper, Contention-aware metrics for distributed algorithms: Comparison of atomic broadcast algorithms, *Proc. 9th IEEE Int'l Conf. on Computer Communications and Networks (IC3N 2000)*, 2000, 582–589.

[7] P. Urbán, Evaluating the performance of distributed agreement algorithms: Tools, methodology and case studies, Ph.D. dissertation, École Polytechnique Fédérale de Lausanne, Switzerland, 2003, number 2824.

[8] W. Chen, S. Toueg, & M. K. Aguilera, On the quality of service of failure detectors, *IEEE Trans. on Computers, 51*(2), 2002, 561–580.

[9] P. Urbán, I. Shnayderman, & A. Schiper, Comparison of failure detectors and group membership: Performance study of two atomic broadcast algorithms, *Proc. Int'l Conf. on Dependable Systems and Networks*, San Francisco, CA, USA, 2003, 645–654.

[10] N. Sergent, X. Défago, & A. Schiper, Impact of a failure detection mechanism on the performance of consensus, *Proc. IEEE Pacific Rim Symp. on Dependable Computing (PRDC)*, Seoul, Korea, 2001, 137–145. [Online]. Available: http://lsrwww.epfl.ch/Publications/ById/292.html

[11] A. Coccoli, P. Urbán, A. Bondavalli, & A. Schiper, Performance analysis of a consensus algorithm combining Stochastic Activity Networks and measurements, *Proc. Int'l Performance and Dependability Symp.*, Washington, DC, USA, 2002, 551–560.

[12] L. Sampaio, F. V. Brasileiro, W. d. C. Cirne, & J. de Figueiredo, How bad are wrong suspicious: Towards adaptive distributed protocols, *Proc. Int'l Conf. on Dependable Systems and Networks (DSN)*, San Francisco, CA, USA, 2003.

[13] P. Urbán, X. Défago, & A. Schiper, Chasing the FLP impossibility result in a LAN or how robust can a fault tolerant server be? *Proc. 20th IEEE Symp. on Reliable Distributed Systems (SRDS)*, New Orleans, LA, USA, 2001, 190–193.

[14] H. Ramasamy, P. Pandey, J. Lyons, M. Cukier, & W. Sanders, Quantifying the cost of providing intrusion tolerance in group communication systems, *Proc. 2002 Int'l Conf. on Dependable Systems and Networks (DSN-2002)*, Washington, DC, USA, 2002, 229–238.

[15] T. D. Chandra, V. Hadzilacos, & S. Toueg, The weakest failure detector for solving consensus, *Journal of the ACM, 43*(4), 1996, 685–722. [Online]. Available: http://www.acm.org/pubs/toc/Abstracts/jacm/234549.html

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

[17] H. Madeira, K. Kanoun, J. Arlat, Y. Crouzet, A. Johansson, R. Lindström, *et al.*, Preliminarily dependability benchmark framework, Dependability Benchmarking project (DBench), EC IST-2000-25425, Project deliverable CF2, 2001. [Online]. Available: http://www.laas.fr/DBench/

[18] P. Urbán, X. Défago, & A. Schiper, Neko: A single environment to simulate and prototype distributed algorithms, *Journal of Information Science and Engineering, 18*(6), 2002, 981–997.