Performance Comparison of a Rotating Coordinator and a Leader Based Consensus Algorithm

Péter Urbán[1], Naohiro Hayashibara[1],
André Schiper[2], and Takuya Katayama[1]

[1]School of Information Science, Japan Advanced Institute of Science and Technology
[2]Swiss Federal Institute of Technology in Lausanne (EPFL)

August 4, 2004
IS-RR-2004-016

Research Report

JAIST

School of Information Science
Japan Advanced Institute of Science and Technology

ISSN 0918-7553

Performance Comparison of a Rotating Coordinator and a Leader Based Consensus Algorithm (extended version)*

Péter Urbán[†] Naohiro Hayashibara[†] André Schiper[‡] Takuya Katayama[†]

[†]Japan Advanced Institute of Science and Technology
1-1 Asahidai, Tatsunokuchi, Nomi, Ishikawa 923-1292, Japan
Email: {urban,nao-haya,katayama}@jaist.ac.jp

[‡]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 well-known asynchronous consensus algorithms. In both algorithms, a leader process tries to impose a decision, and another leader retries if the leader fails doing so. The algorithms elect leaders differently: the Chandra-Toueg algorithm has a rotating leader, whereas processes in the Paxos algorithm elect leaders directly. We investigate the performance implications of this difference.

In the system under study, processes send atomic broadcasts to each other. Consensus is used to decide the delivery order of messages. We evaluate the steady state latency in (1) runs with neither crashes nor suspicions, (2) runs with crashes and (3) runs with no crashes in which correct processes are wrongly suspected to have crashed, as well as the transient latency after (4) one crash, (5) multiple simultaneous crashes and (6) multiple sequenced crashes. The results show that the Paxos algorithm tolerates frequent wrong suspicions (3) and correlated crashes that occur within a short time (5) better, while the performance is comparable in all other scenarios.

Keywords: *simulation, consensus, atomic broadcast, rotating coordinator, leader, asynchronous, failure detector*

* Research supported by the Japan Society for the Promotion of Science, a Grant-in-Aid for JSPS Fellows from the Japanese Ministry of Education, Culture, Sports, Science and Technology, the Swiss National Science Foundation, and the CSEM Swiss Center for Electronics and Microtechnology, Inc., Neuchâtel.

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. These agreement problems have been extensively studied in various system models, and many protocols solving these problems have been published [2,10], offering different levels of guarantees. However, these protocols have mostly been analyzed from the point of view of their safety and liveness properties, and very 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 presents a performance study focusing on consensus, a problem related to most other agreement problems [15], in scenarios that involve failure handling.

The two algorithms. We present a study comparing the performance of two consensus algorithms: the Chandra-Toueg [5] and Paxos [18,21] algorithms. These well-known algorithms are representative of consensus algorithms designed for the asynchronous system model (with minimal extensions necessary to solve consensus). They are important because of their robustness: regardless of their execution environment, they never violate their safety properties. Also, they have the highest possible resiliency in such a system: they tolerate $f < n/2$ crashes in a system with n processes. Moreover, there is an ongoing informal debate in the community about their relative performance. We hope that our comparison will bring some objective arguments to this debate.

The algorithms follow a common pattern by structuring their execution into rounds. In each round, a process called *leader*[1] tries to impose a decision. A round may fail because of failures or uncertainty about failures. The algorithms differ in how they choose a leader for the next round: processes in the Chandra-Toueg algorithm rotate the leader role among all processes, whereas processes in the Paxos algorithm elect leaders directly in an uncoordinated manner. These two approaches are often referred to as rotating coordinator paradigm and leader based paradigm, respectively. In this paper, we investigate the performance implications of this difference.

Elements of the performance study. 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 [5] 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 [29,30]. We model failure detectors in an abstract way, using the quality of service (QoS) metrics proposed by Chen et al. [6]. We compare the algorithms using the benchmarks proposed in [29, 33] (which are stated in terms of the system under study, i.e., atomic broadcast). Our main 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 the steady state latency in (1) runs with neither crashes nor suspicions, (2) runs with crashes and (3) runs with no crashes in which correct processes are wrongly suspected to have crashed, as well as the transient latency after (4) one crash, (5) multiple simultaneous crashes and (6) multiple sequenced crashes.

The results. Our main finding is that, although the two algorithms have comparable performance in scenarios (1), (2), (4) and (6), *the Paxos algorithm performs significantly better in scenarios 3 and 5.* With multiple correlated crashes that occur within a short time, the reason is that the Paxos algorithm elects a correct leader immediately after detecting the crashes. We found the largest difference when wrong failure suspicions were frequent and/or long lasting wrong failure suspicions. The reason is that the Paxos algorithm generates less contention: its leader election mechanism makes sure that only a small subset of all processes start concurrent rounds, whereas the rotating leader scheme in the Chandra-Toueg algorithm results in nearly all processes starting concurrent rounds. Therefore the leader based approach seems more suited to environments in which the failure detection service makes mistakes often.

Structure. 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. Section 5 describes the benchmarks we used, followed by our simulation model for the network and the failure detector and leader oracles in Section 6. Our results are presented in Section 7, and the paper concludes with a discussion 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. [27] compares the impact of different implementations of failure detectors on the Chandra-Toueg consensus algorithm, and Ref. [8] and [24] 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 [31, 33] consider a consensus algorithm embedded in an atomic broadcast algorithm, 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 [29] 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 [22, 27, 33], but the faultload in [22, 27] is different from our crash-transient faultload. Ref. [27] 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 (isolated executions of consensus in [27]) in an analytical or simulation model. The other paper [22] measures the latency of the group membership service used by the algorithm to tolerate crash failures.[2] This way of considering the transient

1 Ref. [5] uses the term *coordinator*. We stick to *leader* throughout the paper.

effects of a crash is less general compared to our faultload, as it is stated in terms of an implementation detail of the algorithm under study.

The assumptions and/or the algorithms used in all the studies listed are too different to allow a meaningful comparison of the results with those in this paper. Our previous work [17] would be an exception: it compares the same algorithms using measurements rather than simulation, and with fewer faultloads. However, bugs discovered and fixed since its publication invalidate the results presented there.

3. Definitions

3.1. System model

We consider a widely accepted system model. It consists of n processes $p_1, \ldots, p_n$ that communicate only by message passing. The system is asynchronous, i.e., we make no assumptions on its timing behavior: there are no bounds on the message transmission delays and the relative processing speed of processes. The network is quasi-reliable: it does not lose, alter nor duplicate messages (messages whose sender or recipient crashes might be lost). 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. Process crashes are rare, and process recovery is slow: both the time between crashes and time to repair are much greater than the latency of the algorithms investigated.

The consensus algorithms used in this paper use *oracles* to tolerate process crashes: the Chandra-Toueg algorithm (CT) uses *failure detector oracles* and the Paxos algorithm (Paxos) uses *leader oracles*. A failure detector oracle outputs a list of processes it suspects to have crashed. It might make mistakes: it might suspect correct processes and it might not suspect crashed processes immediately. A leader oracle outputs a single leader process that it trusts to be alive. All leader oracles in the system strive to output the same leader process. This oracle might make mistakes as well: it might elect crashed processes as leader, and different oracles might elect different leaders. To make sure that the consensus algorithms terminate, we need some assumptions on the behavior of the oracles: $\Diamond S$ for CT [5] and Ω for Paxos [4]. These assumptions are rather weak: they can usually be fulfilled in real systems by tuning implementation parameters of the oracles [11,31]. Also, they are equivalent: one can solve the same set of problems when using the asynchronous model with oracles fulfilling either of $\Diamond S$ and Ω [4].

2 Certain kinds of Byzantine failures are also injected.

3.2. Agreement problems

We next give informal definitions of the agreement problems needed for understanding this paper; see [5, 16] for more formal definitions.

In the consensus problem, each process proposes an initial value. Uniform consensus (considered here) ensures that no two processes decide differently, and that the decision value 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 correct process, then all correct processes eventually A-deliver it, (2) if a process A-delivers a message, then all correct processes eventually A-deliver it, and (3) all processes A-deliver messages in the same order.

The algorithms in this study use (non-uniform) reliable broadcast, which guarantees that if a message is broadcast or delivered by a correct process, then all correct processes eventually deliver it (even if the sender crashes).

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.

4.1. The consensus algorithms

For solving consensus, we use the Chandra-Toueg $\Diamond S$ algorithm [5] and the single-decree Synod algorithm from the Paxos paper [18,21]. Henceforth, we shall refer to the algorithms as *CT algorithm* and *Paxos algorithm*, respectively. 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.

4.1.1. 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 equally strong oracles: $\Diamond S$ failure detectors (CT algorithm; see Section 3.1) and Ω leader oracles (Paxos algorithm). Both tolerate $f < n/2$ crash failures. In both algorithms, processes execute a sequence of asynchronous rounds (i.e., not all processes necessarily execute the same round at a given time t). Each round has a *leader* (called *coordinator* in [5]), whose role is to try to impose a decision value on all processes. If it succeeds, the consensus algorithm terminates; if it fails, some additional rounds are executed with possibly a different leader. Moreover, leaders execute a very similar protocol in each round,[3] discussed in detail in Section 4.1.3.

4.1.2. Electing a leader The main difference between the algorithms is how the leaders are chosen. A new leader is necessary whenever the current round is not successful. A round may not be successful if one or more processes want a different leader, usually because they suspect the current leader to have crashed.

The CT algorithm is based on the rotating coordinator paradigm. Whenever the current leader is suspected, the leader is chosen to be the next process, in a round-robin fashion. In other words, each process executes a sequence of rounds $1, 2, \ldots$, and there is *a priori agreement* on the identity of the leader: process p_i is leader for rounds $kn + i$.

There is no such a priori agreement in the Paxos algorithm. A process p_i considers itself leader (and starts a new round) when its leader oracle outputs p_i. Other processes only start participating in this round when they receive a message from the leader. Leaders always choose unique increasing round numbers: process p_i is leader for rounds $kn + i$, just like in the CT algorithm. However, unlike in the CT algorithm, a given process hardly ever executes all of the rounds $1, 2, \ldots$: there are usually gaps in the sequence of rounds.

4.1.3. Execution of a round We now sketch the execution of one round in each of the two algorithms, illustrated in Fig. 1. Further details of the execution are not necessary for understanding the rest of the paper.

Read phase. Throughout the execution, processes maintain their current *estimate* of the decision value. Both algorithms start the round with a *read phase* whose purpose is to update the leader's estimate with a recent estimate. In the Paxos algorithm, the leader sends a *read* message to all processes, and all processes reply with their estimate (*estimate* messages). In the CT algorithm, the read message is not necessary, as all processes execute every round. In each of the two algorithms, the leader only waits for an estimate from a majority of all processes, and then updates its own estimate.

Write phase. In this phase, the leader sends its estimate to all, proposing its acceptance (*proposal* messages). A process accepts this estimate if it has not seen messages from a later round (in the case of the Paxos algorithm) or if it does not suspect the leader (CT algorithm).

When a process accepts a proposal, it updates its own estimate and sends back an *ack* message; otherwise, it sends back a *nack* message (not shown in Fig. 1). In the case of the CT algorithm, the nack message is sent *before* receiving the proposal.

3 This is why we chose the CT algorithm over other algorithms written for the same system model (e.g., [25] and [20]).

The leader waits for messages from a majority of all processes, and decides if it has received a majority of ack messages. In this case, it also sends a *decision* message to all using reliable broadcast. Upon receiving this message, the other processes decide as well. If the leader receives one nack message before deciding (this is not shown in Fig. 1) it finishes executing the round without deciding.

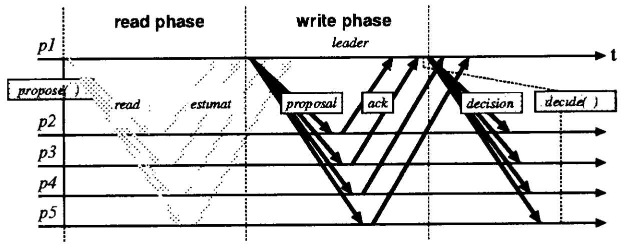

Figure 1. Example of a round in the CT and Paxos algorithms (CT does not send the *read* message)

4.2. Optimizations to the consensus algorithms

The consensus algorithms implemented contain several optimizations with respect to the published versions [5, 18, 21]. The goal of the optimizations is to reduce the number of messages in the most common scenario: when no process is suspected (CT algorithm) or when the leader is the same process (p_1) throughout the execution (Paxos algorithm).

- The read phase is not necessary in the first round, in either of the two algorithms. This is why its messages are gray in Fig. 1.
- In the original CT algorithm, the non-leader processes start the next round immediately after sending the *ack* message. This generates *estimate* messages which are not needed in the most common scenario. These messages degrade performance. To prevent this, non-leader processes wait for an *abort* message before starting the new round.[4] The *abort* message is sent by the leader if it receives *nack* messages.
- In the write phase, the leader stops the current round after receiving the first *nack* message, because it is known at this point already that the round has failed. The original algorithms always wait for (*ack* and *nack*) messages from a majority of processes.

4 The non-leader processes also start a new round if they start suspecting the leader.

- In both algorithms, the decision message must be sent using reliable broadcast (see Section 3.2). We use an efficient algorithm inspired by [13] that requires only one broadcast message if the sender is not suspected.
- The CT algorithm always starts with the same leader p_1. If p_1 crashes, this affects steady-state performance negatively. We fix this problem by having the consensus decide on the first leader of the next consensus (beside the order of messages) [9]. Processes propose the first process that their failure detector trusts as first leader. This choice makes sure that, eventually, crashed processes do not ever become first leaders.

4.3. The Chandra-Toueg atomic broadcast algorithm

In the Chandra-Toueg atomic broadcast algorithm [5], a process executes A-broadcast by sending a message to all processes.[5] 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 *sets of message identifiers*. Let msg_k be the set of message IDs decided by consensus $\#k$. The messages denoted by msg_k are A-delivered before the messages denoted by msg_{k+1}, and the messages denoted by msg_k are A-delivered according to a deterministic function, e.g., according to an order relation defined on their IDs.

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 Paxos consensus algorithms.

The performance of the algorithms can be improved by packing messages from subsequent consensus executions into one message. For the sake of simplicity, we did not perform such optimizations [1,3,12]. This decision affects the two algorithms in the same way, hence we introduce no bias in the performance study.

5. Benchmarks

This section describes our benchmarks, 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). Previous versions of the benchmarks are published in [29,33].

5 This message is sent using reliable broadcast. We use the efficient algorithm mentioned Section 4.2.

5.1. Performance metrics and workloads

Our main performance metric is the *early latency* of atomic broadcast. 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 that elapses until the first A-delivery of m, i.e., $L \overset{\mathrm{def}}{=} (\min_{i=1,\ldots,n} t_i) - t_0$. In our study, we compute the mean for L over a lot of messages and several executions.

This performance metric makes sense in practice. Consider a service replicated for fault tolerance using active replication [26]. Clients of this service send their requests to the server replicas using Atomic Broadcast. 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 to which the request was delivered first. Thus there is a direct link between the response time of the replicated server and the latency L.

Beside the early latency, we also compute the *late latency*, the time that elapses until the last A-delivery of a message m: $L_{late} \overset{\mathrm{def}}{=} (\max_{i=1,\ldots,n} t_i) - t_0$.

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

The system can only reach a steady state if the throughput is under some maximal throughput T_{max}. Beyond this throughput, some processes are left behind. We detect if the system reaches steady state by observing if the late latency stabilizes over time.

5.2. Faultloads

The faultload is the part of the workload that describes failure-related events that occur during an experiment [19]. We concentrate on (1) crash failures of processes, and (2) the behavior of unreliable failure detectors. We evaluate the performance of the algorithms with four different faultloads. We now describe each of them in detail, mentioning which parameters influence latency with each faultload.

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-steady faultload. One or more crashes occur before the experiment. We measure latency after the system reaches its steady state: a sufficiently long time after startup and any crashes. Beside A, n and T, an additional parameter is the set of crashed processes. In the steady state of the system, all failure detectors in the system permanently suspect all crashed processes at this point, and all leader oracles have elected the same correct process. No wrong suspicions occur, and the leader no longer changes.

Crash-transient faultload. With this faultload, we inject one or more crashes at some point in time after the system reached a steady state. Multiple crashes represent correlated failures. We model both simultaneous multiple crashes and crashes that happen in a sequenced manner, spaced apart by the *crash interval* T_C.

After the crashes, 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 crashes. Our faultload definition represents the simplest possible choice: we determine the latency of an atomic broadcast sent when the crashes start (by a process that does not crash). Of course, the latency of this atomic broadcast may depend on the choice for the sender and the crashing processes. In order to reduce the number of parameters, we consider the worst case, i.e., the case that increases latency the most.

The precise definition for the faultload is the following. Consider that a list of c processes $\mathcal{C}$ crashes at times $t, t + T_C, \dots, t + (c-1) \cdot T_C$, respectively, where $T_C \geq 0$ (no other crashes nor wrong suspicions occur). Let process p ($p \notin \mathcal{C}$) execute *A-broadcast*(m) at t. Let $L(p, \mathcal{C})$ be the mean latency of m, averaged over a lot of executions. Then $L_{crash} \overset{\text{def}}{=} \max_{p,\mathcal{C}} L(p, \mathcal{C})$, i.e., we choose the sender and the crashing processes such that latency increases the most.

Beside A, n, T, c and T_C, an additional parameter describes how fast failure detectors and leader oracles detect the crashes. This parameter is discussed in Section 6.2.

Suspicion-steady faultload. No crashes occur, but failure detectors generate wrong suspicions, and leader oracles change their mind about the leader. This causes the algorithms to take extra steps and thus increase latency. Beside A, n and T, additional parameters include how often wrong suspicions occur and how long they last. These parameters are discussed in Section 6.2.

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 [32] to conduct our experiments. We used the same models for our previous work [29, 33].

6.1. Modeling the execution environment

We now describe how we modeled the transmission of messages. We use a model inspired from simple models of Ethernet networks [28], and validated in [29]. 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. 2): 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.

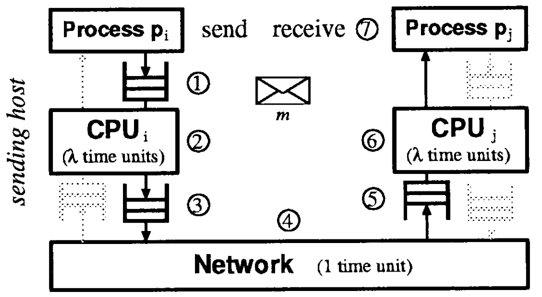

Figure 2. Transmission of a message in our network model.

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.

We model network-level multicasts: a message sent to several destinations is only processed once on the sending CPU resource and on the network resource.

Crashes are modeled as follows. If a process p_i crashes at time t, no messages can pass between p_i and CPU$_i$ after t; however, the messages on CPU$_i$ and the content of the attached queues are still sent, even after time t. In real systems, this corresponds to a (software) crash of the application process (operating system process), rather than a (hardware) crash of the host or a kernel panic. We chose to model software crashes because they are more frequent in most systems [14].

6.2. Modeling failure detectors

One approach to examine the behavior of a failure detector is implementing it and using the implementation in the experiments. However, this approach would restrict the generality of our performance study: another choice for the algorithm would likely give different results. Also, it is not justified to model the failure detector in so much detail, as other components of the system, like the execution environment, 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 [6]. The authors consider the failure detector at a process q that monitors another process p, and identify the following three primary QoS metrics:

- *Detection time T_D:* The time that elapses from p's crash to the time when q starts suspecting p permanently. The definition is illustrated in Fig. 3.

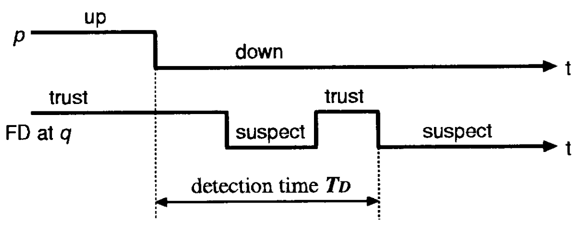

Figure 3. Quality of service metric expressing the speed of failure detection. Process q monitors process p.

- *Mistake recurrence time T_{MR}:* The time between two consecutive mistakes (q wrongly suspecting p), given that p did not crash; see Fig. 4.
- *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); see Fig. 4.

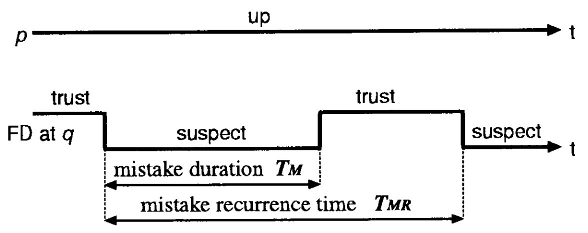

Figure 4. Quality of service metrics describing wrong suspicions made by failure detectors. Process q monitors process p.

Not all of these metrics are equally important in each of our faultloads (see Section 5.2). In the *normal-steady* faultload, the metrics are not relevant. The same holds in the *crash-steady* faultload, because we observe the system a sufficiently long time after all crashes, long enough to have all failure detectors to suspect the crashed processes permanently. In the *suspicion-steady* faultload no crash occurs, hence the latency of atomic broadcast only depends on T_{MR} and T_M (shown in Fig. 4). In the *crash-transient* faultload no wrong suspicions occur, hence T_D is the relevant metric (shown in Fig. 3).

In [6], the QoS metrics are random variables, defined on a pair of processes. In our system, where n processes monitor each other, we have thus $n(n-1)$ failure detectors in the sense of [6], each characterized with three random variables (T_D, T_{MR}, T_M). In order to have an executable model for the failure detectors, we have to define (1) how these random variables depend on each other, and (2) how the distribution of each random variable can be characterized. To keep our model simple, we assume that all failure detector modules are independent and the tuples of their random variables are identically distributed. Moreover, note that we do not need to model how T_{MR} and T_M depend on T_D, as the two former are only relevant in the suspicion-steady faultload, whereas T_D is only relevant in the crash-transient faultload. As for the distributions of the metrics, we took the simplest possible choices: T_D is a constant, and both T_{MR} and T_M are exponentially distributed with (different) constant parameters. This choice only represents a starting point, as we are not aware of any previous work we could build on (apart from [6] that makes similar assumptions). We will refine our models as we gain more experience.

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. This is the case whenever T_D and T_{MR} are not too small. Moreover, if this is not the case, it is fair to assume that the overhead of failure detection affects both algorithms, and furthermore, that the overhead affects the algorithm that already has performance problems to a greater extent. Thus it is unlikely that neglecting the load generated by failure detectors actually changes which algorithm performs better at any given setting (though it might change the absolute values of performance metrics).

6.3. Modeling leader oracles

Our leader oracles for the Paxos algorithm rely on failure detectors: at any point in time, the leader is the process with the smallest index of all processes trusted by the failure detector. We implemented leader oracles with failure detectors because a leader oracle must detect the crash of other processes.[6] The failure detectors underlying the leader oracles are modeled with their quality of service parameters as described in the previous section.

Recall from Section 3.1 that the CT algorithm requires a $\Diamond\mathcal{S}$ failure detector and the Paxos algorithm an Ω leader oracle. The reader might wonder why the transformation of $\Diamond\mathcal{S}$ to Ω, described in [7], is not used here. The reason is that we do not aim at modeling $\Diamond\mathcal{S}$ or Ω; instead, we aim at modeling the performance characteristics of failure detectors (following [6]).[7] One question might be whether our failure detector model ensures a better coverage of the assumptions of $\Diamond\mathcal{S}$ or those of Ω. However, this question is not relevant for our study. Oracles satisfying the assumptions of $\Diamond\mathcal{S}$ and Ω, respectively, ensure that the algorithms terminate, but there are runs in which the algorithms terminate, even though the assumptions are not satisfied. Moreover, we can use simulations to obtain coverage data directly.

7. Results

We now present our results for all four faultloads and a variety of network models. We obtained results at 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 $\lambda = 1$ is an intermediate setting. For example, in current local area networks, the time spent on the hosts is much higher than the time spent on the wire, and thus $\lambda = 10$ is probably the setting that corresponds best to such an environment.

6 The leader oracle has other potential uses, e.g., it can be used to implement load balancing among all correct processes (see Section 8). We intend to investigate this aspect in the future.

7 Another reason is that our transformation is more efficient ([7] uses additional messages).

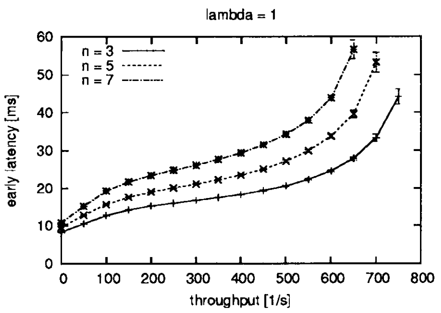

Figure 5. Latency vs. throughput with the normal-steady faultload, for both Paxos and CT.

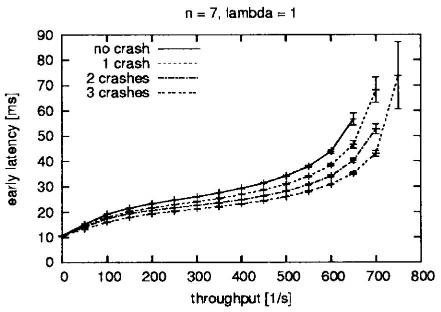

Figure 6. Latency vs. throughput with the crash-steady faultload, for both Paxos and CT.

Most graphs show the early latency vs. the throughput. Graphs showing the late latency are presented in the appendix only. Values of the late latency are slightly higher, but all other characteristics of the corresponding graphs are very similar. The reason is that if one process reaches a decision in either of the consensus algorithms, all other processes will soon follow, thanks to the decision message (see Section 4.1.3). The maximal throughput is approximately the highest throughput value, that is, the x coordinate of the rightmost point, in all graphs showing the steady-state latency; beyond this throughput, the late latency did not stabilize (see Section 5.1). 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 (one that refers to time units). Any other value could have been used. The 95% confidence interval is shown for each point in the graphs.

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

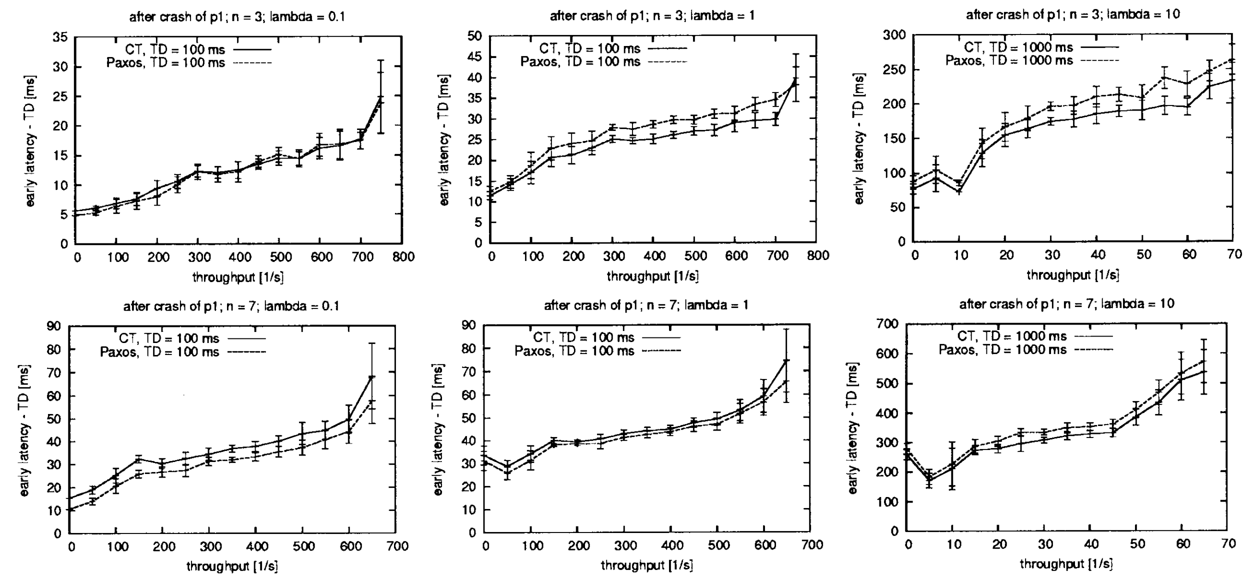

Figure 7. Latency overhead vs. throughput with the crash-transient faultload. One process crashes.

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. Also, we always ran the algorithms with seven or fewer processes. Studying the scalability of the algorithms did not seem worthwhile, because neither algorithm is especially scalable: processes often wait for messages from $\approx n/2$ processes, whereas scalable algorithms tend to synchronize much fewer processes (see, e.g, [23]). Also, it is questionable that using algorithms tolerating $\approx n/2$ failures makes sense when n is large.

7.1. Normal-steady and crash-steady faultloads (Figures 5 and 6, Appendixes B and C)

With these faultloads, the two algorithms have the same performance. Each curve thus shows the latency of *both* algorithms. For the sake of readability, we only present a subset of the results in Fig. 5 (normal-steady faultload) and Fig. 6 (crash-steady faultload). The full set of results is presented in Appendixes B and C. The latency increases with the throughput and with the number of processes. Somewhat surprisingly, the latency *decreases* with the number of crashes. The reason is that the crashed processes no longer load the network with messages.

The fact that the two algorithms have the same performance is not surprising. Their only important difference is the way of electing a new leader, and no new leader is elected with these faultloads (such that this influences the steady-state performance). In fact, we have deliberately chosen similar algorithms for this study, so that we can concentrate on the performance differences observed with the other faultloads.

7.2. Crash-transient faultload (Figures 7 to 10, Appendix D)

With this faultload and c crashes, we only present the latency after crashing the first c processes ($p_1, \ldots, p_c$), as this is the case resulting in the highest transient latency (and the most interesting comparison). The crash of any additional processes affects the two algorithms in the same way (slightly decreased latency; cf. Fig. 6).

We set the failure detection timeout T_D to 100 ms at $\lambda = 0.1$ or 1, and to 1000 ms at $\lambda = 10$. This choice models a reasonable trade-off for the failure detector. On the one hand, the detection time T_D is low enough (comparable to the latency overhead) 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 (it is a high multiple of the roundtrip time at low loads: $2 + 4\lambda$) to avoid that failure detectors suspect correct processes.[8]

All figures show the *latency overhead*, i.e., the latency minus the detection time T_D, rather than the latency.[9] Graphs showing the latency overhead are more illus-

8 As we use an abstract model for the failure detectors for the sake of generality, this does not appear directly in our simulations. The argumentation is about a hypothetical implementation. Given that this implementation can afford spending a high multiple of the roundtrip time before generating a suspicion, wrong suspicions will be rare.

9 Actually, Fig. 10 shows a derived quantity.

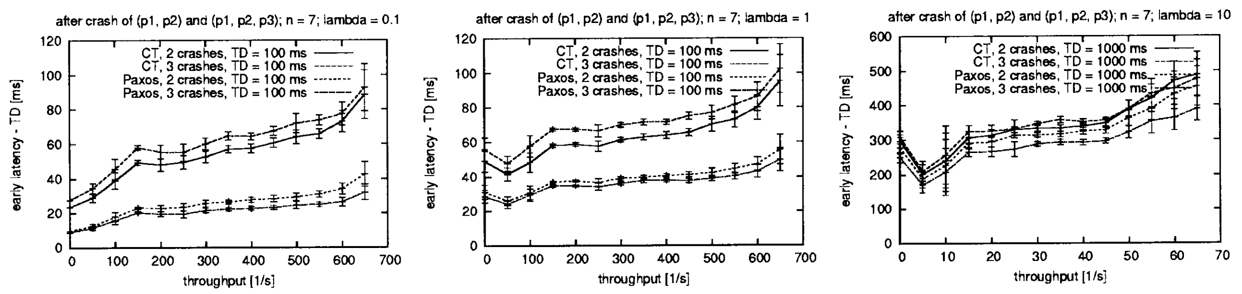

Figure 8. Latency overhead vs. throughput with the crash-transient faultload. Multiple processes crash simultaneously.

trative; 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 first leader is detected.

One crash (Fig. 7). We start by discussing the results for the case of one crash. The latency overhead of both algorithms is shown at $n = 3$ (top) and $n = 7$ (bottom) and a variety of values for λ (0.1, 1 and 10 from left to right).

The results show that (1) both algorithms perform rather well (the latency overhead of both algorithms is only a few times higher than the latency with the normal-steady faultload; see Fig. 5) and that (2) the algorithms perform roughly the same. The CT algorithm performs slightly better at $n = 3, \lambda \geq 1$ and $n = 7, \lambda = 10$, i.e., with a small number of processes and a high λ meaning a relatively fast network. The Paxos algorithm performs slightly better at $n = 7, \lambda \leq 1$, i.e., with a lot of processes and a small λ meaning a relatively slow network.

The differences can be explained by differences in the execution of the algorithms once the crash of the first leader is detected. In the CT algorithm, all processes send a *nack* message to the first leader. In the Paxos algorithm, the new leader sends a *read* message. The rest of the execution (from the *estimate* message of the second round) is the same. The CT algorithm thus uses fewer communication steps, but generates more contention on the network; moreover, the increase in network contention is proportional to the number of processes. This explains why the CT algorithm is favored by a fast network and a small number of processes.

Multiple simultaneous crashes (Fig. 8). For this case, the latency overhead of both algorithms is shown at $n = 7$, for 2 and 3 crashes (the algorithms do not tolerate more than 3 crashes) and a variety of values for λ (0.1, 1 and 10 from left to right).

The results are different from those obtained with one crash only: the Paxos algorithm always outperforms the CT algorithm. The reason is that the CT algorithm takes more rounds: it rotates over all crashed processes first, whereas the Paxos algorithm elects a correct leader after the first round.

The fact that the CT algorithm rotates over the crashed processes also explains why its latency increases with the number of crashes. The latency of the Paxos algorithm, however, decreases with the number of crashes. The reason is that fewer correct processes load the system with messages to a smaller extent (cf. Fig. 6).

Multiple sequenced crashes (Figures 9 and 10). Finally, we investigated what happens when multiple non-simultaneous crashes occur. The order in which processes crash is $p_1, \ldots, p_c$, so we obtain the highest transient latency. The time that elapses between two crashes is called crash interval T_C (thus all values shown in Fig. 8 were obtained at $T_C = 0$).

Fig. 9 shows the effect of T_C on the latency overhead. The curves were obtained with $n = 7$ processes, $c = 3$ crashes and a low load (0.1 s^{-1}) at a variety of values for λ (0.1, 1 and 10 from left to right). The characteristics of all other curves are similar (see Appendix D). Up to $T_C \approx T_D$, the latency of both algorithms increases according to $(c-1) \cdot T_C$; the reason is that each of the crashes is not yet detected when the following crash happens. This region is highlighted in Fig. 10, which plots the latency overhead minus $(c-1) \cdot T_C$ rather than the latency overhead. One can see that the performance advantage of Paxos decreases as T_C increases. The reason is that the Paxos algorithm perceives the crashes separately: it cannot elect a correct leader in one step as at small values of T_C (Fig. 8), and thus cannot maintain a significant performance advantage over the CT algorithm.

Except for small T_C values, the relative performance in this region (T_C below $\approx T_D$) depends on λ; higher values, i.e., a faster network, favor the CT algorithm. The explanation is the same as in the case of one crash (see above; Fig. 7): the CT algorithm uses fewer communication steps but generates more contention on the network.

As T_C grows beyond $\approx T_D$, both algorithms detect the

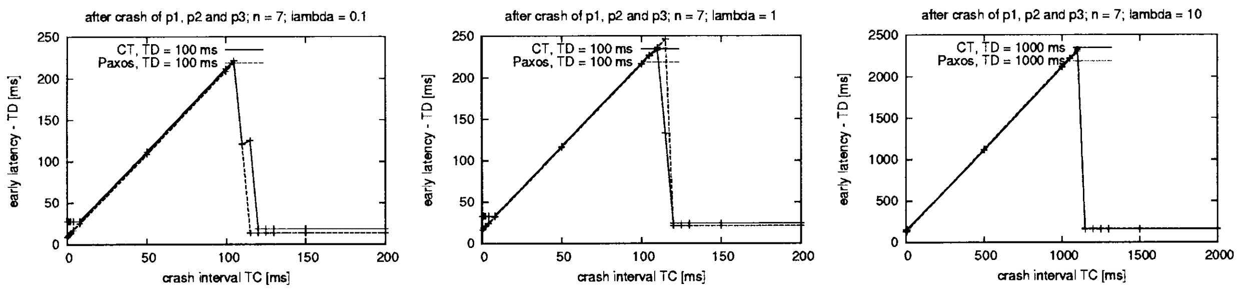

Figure 9. Latency overhead vs. crash interval with the crash-transient faultload. Multiple processes crash in a sequenced manner.

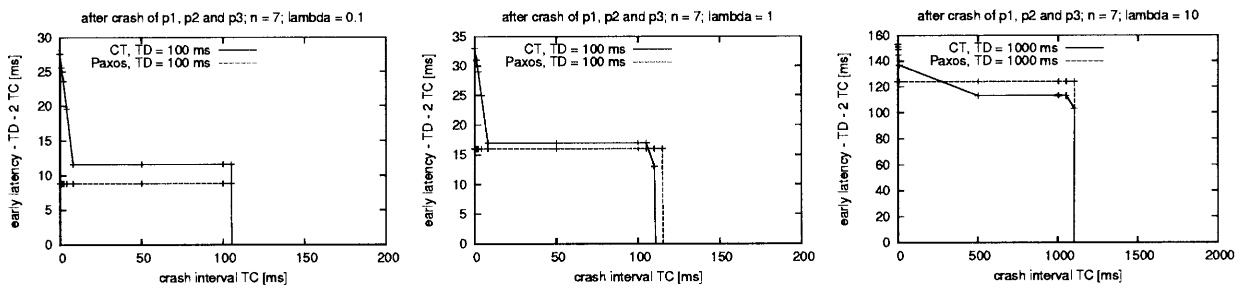

Figure 10. Latency overhead minus $2 \cdot T_C$ vs. crash interval T_C with the crash-transient faultload. Multiple processes crash in a sequenced manner.

first crash before the next ones happen, and can reach a decision. Therefore the transient latency quickly decreases until it is essentially the same as with a single crash (Fig. 7).

7.3. Suspicion-steady faultload (Figures 11 and 12, Appendix E)

The occurrence of wrong suspicions are quantified with the T_{MR} and T_M QoS metrics of the failure detectors. As this faultload does not involve crashes, we expect that the mistake duration T_M is short. In our first set of results (Fig. 11 for $\lambda = 1$; the results for $\lambda = 0.1$ and 10 are similar and are omitted here for better readability; see Appendix E for the full set of results) we hence set T_M to 0, and latency is shown as a function of T_{MR}. In each figure, we have four graphs: the left column shows results with 3 processes, the right column those with 7; the top row shows results at a low load (10 s^{-1}; 1 s^{-1} if $\lambda = 10$) and the bottom row at a moderate load (300 s^{-1}; 30 s^{-1} if $\lambda = 10$); the algorithms can take a throughput of about 700 s^{-1} (70 s^{-1} if $\lambda = 10$) in the absence of suspicions (i.e., with the normal-steady faultload; see Fig. 5 and Appendix B).

The results show that the CT algorithm is much more sensitive to wrong suspicions if these occur frequently. We illustrate this on Fig. 11: at $n = 3$ and $T = 10$ s^{-1}, that is, the settings at which the CT algorithm tolerates wrong suspicions most, the CT algorithm only works if $T_{MR} \geq 5$ ms, whereas the FD algorithm still works at the smallest T_{MR} value considered (1 ms); the latency of the two algorithms is only equal at $T_{MR} \geq 100$ ms. The CT algorithm breaks down at higher values of T_{MR} for all other settings, whereas the Paxos algorithm continues to work even with 1 ms.

The results can be explained by the difference in the mechanisms that the algorithms use to elect the next leader. The CT algorithm always chooses the next process (in a round-robin manner) as the next leader. Moreover, suspicions are likely to abort the current round. Therefore, if wrong suspicions occur frequently, a lot of rounds are needed to finish a consensus execution, and all processes become leaders, executing rounds that overlap. In contrast, the Paxos algorithm is run with a leader oracle that elects the process with the smallest index among all suspected processes. If suspicions are short ($T_M = 0$), the leader oracle will only ever elect p_1 and p_2 as leader. Only these two processes start overlapping rounds. Moreover, suspicions, even if they lead to a change in the output of the leader ora-

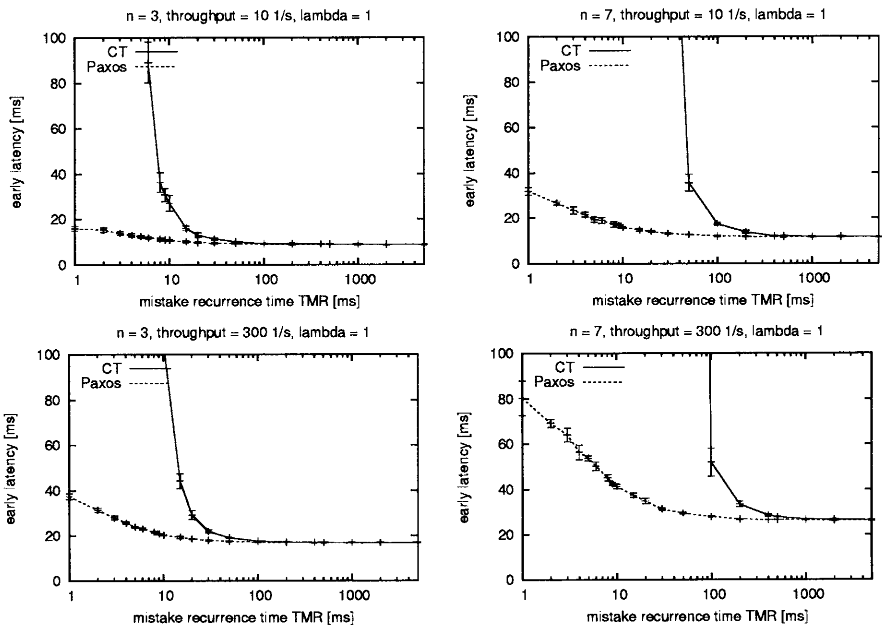

Figure 11. Latency vs. T_{MR} with the suspicion-steady faultload, with $T_M = 0$ ($\lambda = 1$).

cle, do not abort the current round directly; their only effect is to start other rounds in parallel that might conflict with the current round. Because of these differences, the CT algorithm generates much more contention on the hosts and the network: it is likely that n processes run rounds in parallel, whereas the Paxos algorithm only has two processes that run rounds in parallel. The increased contention of CT is the reason why the Paxos algorithm performs better with this faultload.

In the second set of results (Fig. 12 for $\lambda = 1$; the results for $\lambda = 0.1$ and 10 are similar and are omitted here for better readability; see Appendix E for the full set of results) T_{MR} is fixed and T_M is on the x axis. We chose T_{MR} such that the latency of the two algorithms is close to equal at $T_M = 0$. For example, with $\lambda = 1$ (Fig. 12), (i) $T_{MR} = 100$ ms for $n = 3$ and (ii) $T_{MR} = 1000$ ms for $n = 7$.

The results show that the CT algorithm is more sensitive to the mistake duration T_M as well, not just the mistake recurrence time T_{MR}. Once again, the difference can be attributed to the fact that the Paxos algorithm generates less contention: its leader oracle usually outputs only a small subset of all processes, hence only a few processes start rounds concurrently, whereas all processes are likely to do so in the CT algorithm.

8. Discussion

We have compared the performance of the Chandra-Toueg and Paxos consensus algorithms. These algorithms are representative for consensus algorithms designed for the asynchronous system model (with a minimal extension to allow us to solve the consensus problem) and $f < n/2$ process crashes (the highest f that the system model allows). Following a common pattern, the algorithms have a similar structure: they execute a sequence of rounds whereby each round has a leader that tries to impose a decision. They differ in how they tolerate (suspected) failures of the leader: processes in the Chandra-Toueg algorithm rotate the leader role among all processes, whereas processes in the Paxos algorithm elect leaders directly in an uncoordinated manner.

Not surprisingly, the two algorithms have the same steady-state performance if neither crashes nor wrong suspicions occur, or if crashes occur but wrong suspicions do not. In fact, the algorithms differ only in how they handle suspected crashes, and this difference does not come into play in these scenarios. This result allows us to state with confidence that performance differences observed in the other scenarios are due to the differences in failure handling and not other artifacts of the two algorithms.

As for the transient performance after one crash or multiple sequenced crashes, the performance differences are small, and the relative performance depends on the relative speed of the network and the hosts, as well as on the number of processes. The Paxos algorithm has better transient performance after multiple correlated crashes that happen simultaneously or within a short time, because its leader oracle elects a correct leader immediately after detecting the

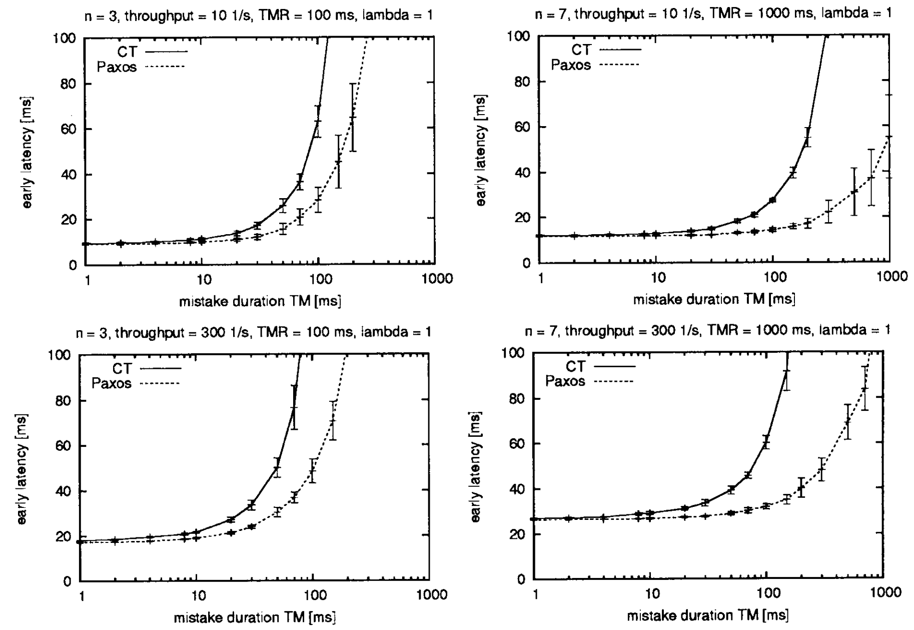

Figure 12. Latency vs. T_M with the suspicion-steady faultload, with T_{MR} fixed ($\lambda = 1$).

crashes. This advantage seems to be inherent to the leader-based approach that the Paxos algorithm follows.

We found the largest difference in scenarios with frequent or long lasting wrong failure suspicions. In such scenarios, the Paxos algorithm performs better. The reason is that it generates less contention: its leader oracle makes sure that only a small subset of all processes start concurrent rounds, whereas the rotating leader scheme in the Chandra-Toueg algorithm results in nearly all processes starting concurrent rounds. Once again, this advantage in environments in which the failure detection service makes mistakes often seems to be inherent to the leader-based approach.

We have chosen consensus algorithms with a centralized communication scheme, with one process coordinating the others. In the future, we would like to investigate algorithms with a decentralized communication scheme (e.g., [25] and [20]) as well. We would also like to investigate how results change in a load balanced configuration, e.g., in a configuration in which the first leader of subsequent consensus executions rotates among all processes that are alive. The coordinated fashion of electing the next leader in the Chandra-Toueg algorithm might provide performance benefits in such a configuration.

Acknowledgments

We would like to thank Pierre Metrailler for his help in implementing the algorithms and performing the simulations, the anonymous referees for their suggestions and Neeraj Suri for his help in shaping the final version of this paper.

References

[1] E. Anceaume. A lightweight solution to uniform atomic broadcast for asynchronous systems. In *Proc. 27th Int'l Symp. on Fault-Tolerant Computing (FTCS-27)*, pages 292–301, Seattle, WA, USA, June 1997.

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

[3] R. Boichat, P. Dutta, S. Frølund, and R. Guerraoui. Deconstructing Paxos. *SIGACT News*, 34(1):47–67, 2003.

[4] 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.

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

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

[7] F. Chu. Reducing Ω to $\Diamond W$. *Information Processing Letters*, 67(6):289–293, Sept. 1998.

[8] 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.*, pages 551–560, Washington, DC, USA, June 2002.

[9] X. Défago and A. Schiper. Specification of replication techniques, semi-passive replication, and lazy consensus. Research Report KS-RR-2002-001, Japan Advanced Institute of Science and Technology, Ishikawa, Japan, Feb. 2002.

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

[11] C. Fetzer, M. Raynal, and F. Tronel. An adaptive failure detection protocol. In *Proc. 8th IEEE Pacific Rim Symp. on Dependable Computing (PRDC-8)*, Seoul, Korea, Dec. 2001.

[12] R. Friedman and R. van Renesse. Packing messages as a tool for boosting the performance of total ordering protocols. TR 95-1527, Dept. of Computer Science, Cornell University, Ithaca, NY, USA, July 1995.

[13] S. Frolund and F. Pedone. Revisiting reliable broadcast. Technical Report HPL-2001-192, HP Laboratories, Palo Alto, CA, USA, Aug. 2001.

[14] J. Gray. Why do computers stop and what can be done about it ? In *Proc. 5th Symp. on Reliablity in Distributed Software and Database systems*, Jan. 1986.

[15] R. Guerraoui and A. Schiper. The generic consensus service. *IEEE Trans. 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] N. Hayashibara, P. Urbán, A. Schiper, and T. Katayama. Performance comparison between the Paxos and Chandra-Toueg consensus algorithms. In *Proc. Int'l Arab Conf. on Information Technology (ACIT 2002)*, pages 526–533, Doha, Qatar, Dec. 2002.

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

[19] H. Madeira, K. Kanoun, J. Arlat, Y. Crouzet, A. Johansson, R. Lindström, et al. Preliminarily dependability benchmark framework. Project deliverable CF2, Dependability Benchmarking project (DBench), EC IST-2000-25425, Aug. 2001.

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

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

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

[23] L. Rodrigues, H. Fonseca, and P. Veríssimo. Totally ordered multicast in large-scale systems. In *Proc. 16th Int'l Conf. on Distributed Computing Systems (ICDCS-16)*, pages 503–510, Hong Kong, May 1996.

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

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

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

[27] 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.

[28] K. Tindell, A. Burns, and A. J. Wellings. Analysis of hard real-time communications. *Real-Time Systems*, 9(2):147–171, Sept. 1995.

[29] P. Urbán. *Evaluating the Performance of Distributed Agreement Algorithms: Tools, Methodology and Case Studies.* PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, Aug. 2003. Number 2824.

[30] 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)*, pages 582–589, Oct. 2000.

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

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

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

A. Explanations for the full set of results

For the sake of readability, only a representative subset of our results appears in the body of the paper. In the appendix, we present the full set of results. This includes results for all combinations of the following:

- Performance metrics: early and late latency.
- Relative contention in the network model (λ): 0.1, 1 and 10.
- Number of processes (n): 3, 5 and 7.
- Number of crashes: 1 (for $n = 3$); 1 and 2 (for $n = 5$); 1, 2 and 3 (for $n = 7$).

B. Full set of results for the normal-steady faultload

B.1. Graphs showing the early latency

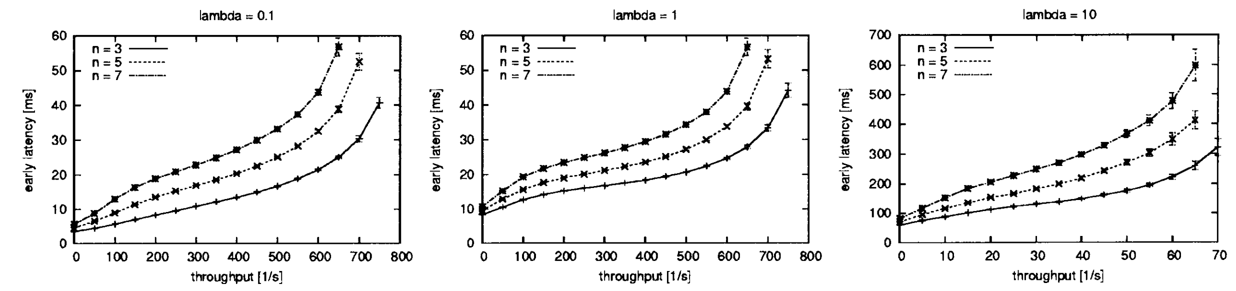

Figure 13. Latency vs. throughput with the normal-steady faultload.

B.2. Graphs showing the late latency

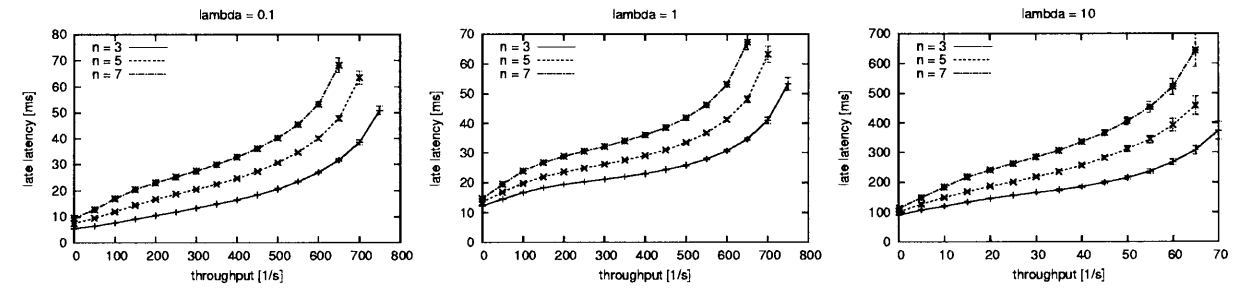

Figure 14. Latency vs. throughput with the normal-steady faultload.

C. Full set of results for the crash-steady faultload

C.1. Graphs showing the early latency

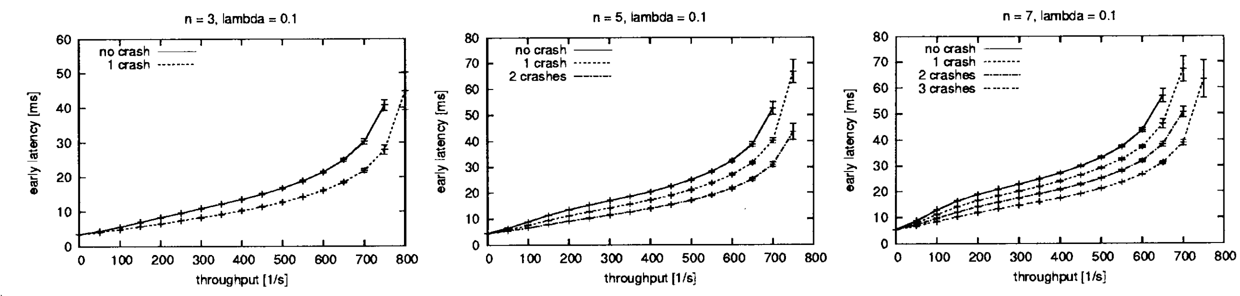

Figure 15. Latency vs. throughput with the crash-steady faultload ($\lambda = 0.1$).

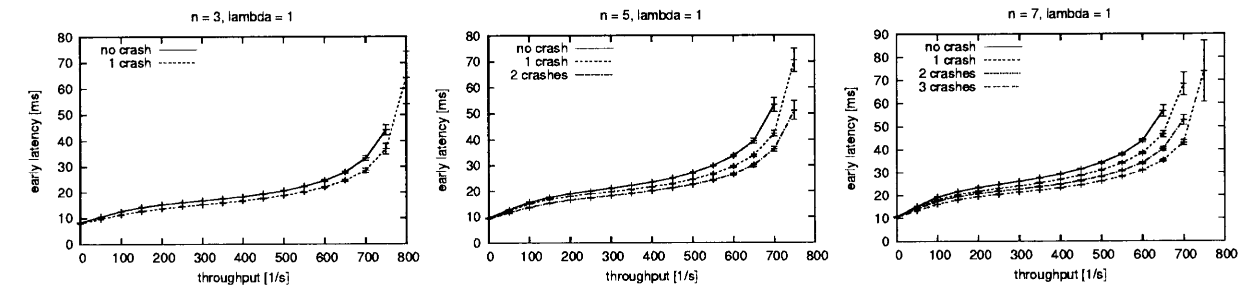

Figure 16. Latency vs. throughput with the crash-steady faultload ($\lambda = 1$).

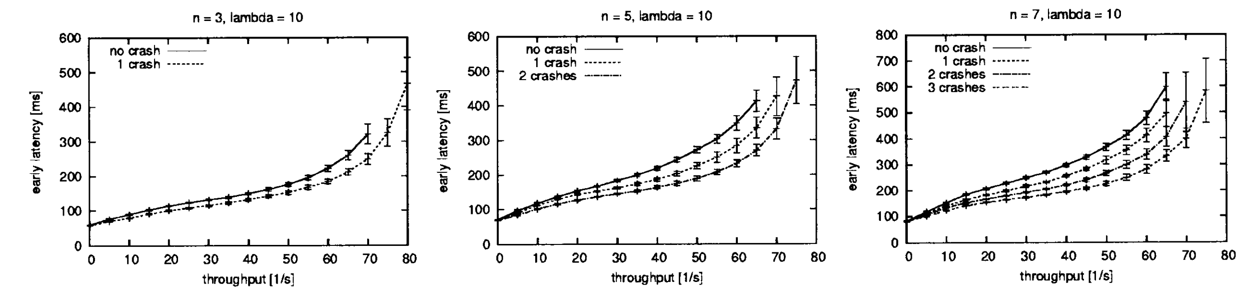

Figure 17. Latency vs. throughput with the crash-steady faultload ($\lambda = 10$).

C.2. Graphs showing the late latency

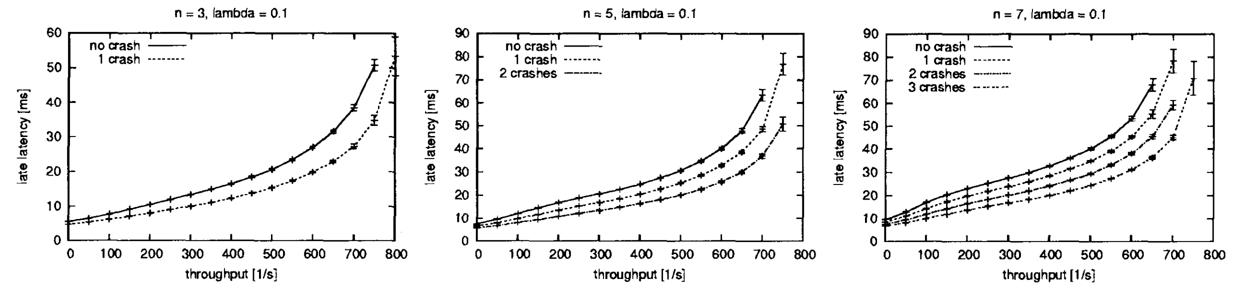

Figure 18. Latency vs. throughput with the crash-steady faultload ($\lambda = 0.1$).

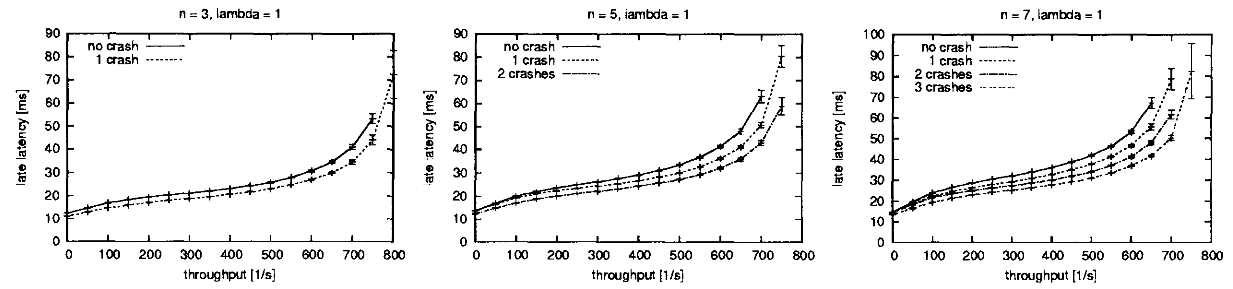

Figure 19. Latency vs. throughput with the crash-steady faultload ($\lambda = 1$).

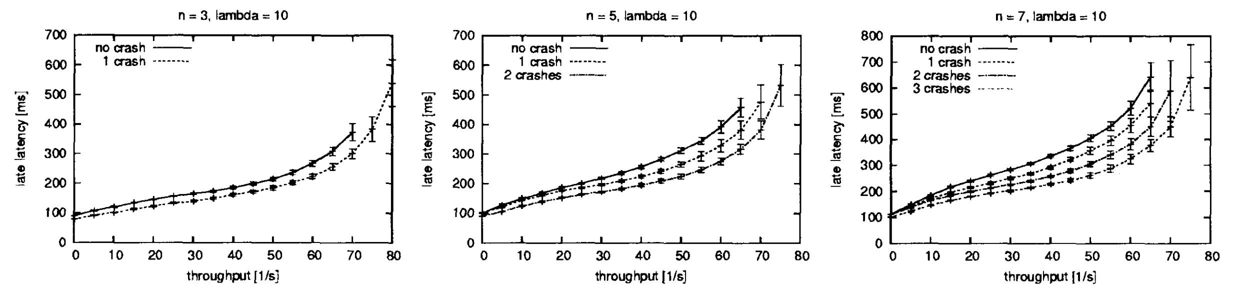

Figure 20. Latency vs. throughput with the crash-steady faultload ($\lambda = 10$).

D. Full set of results for the crash-transient faultload

D.1. Graphs showing the early latency overhead

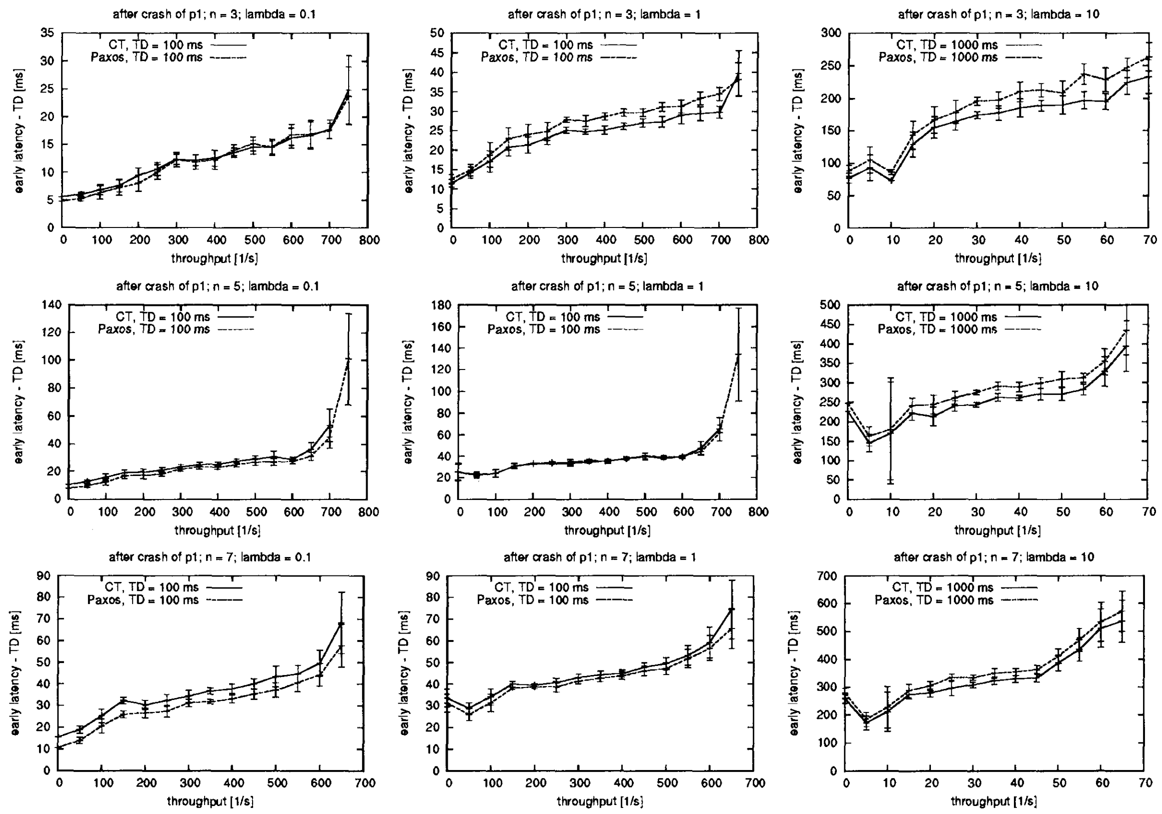

Figure 21. Latency overhead vs. throughput with the crash-transient faultload. One process crashes.

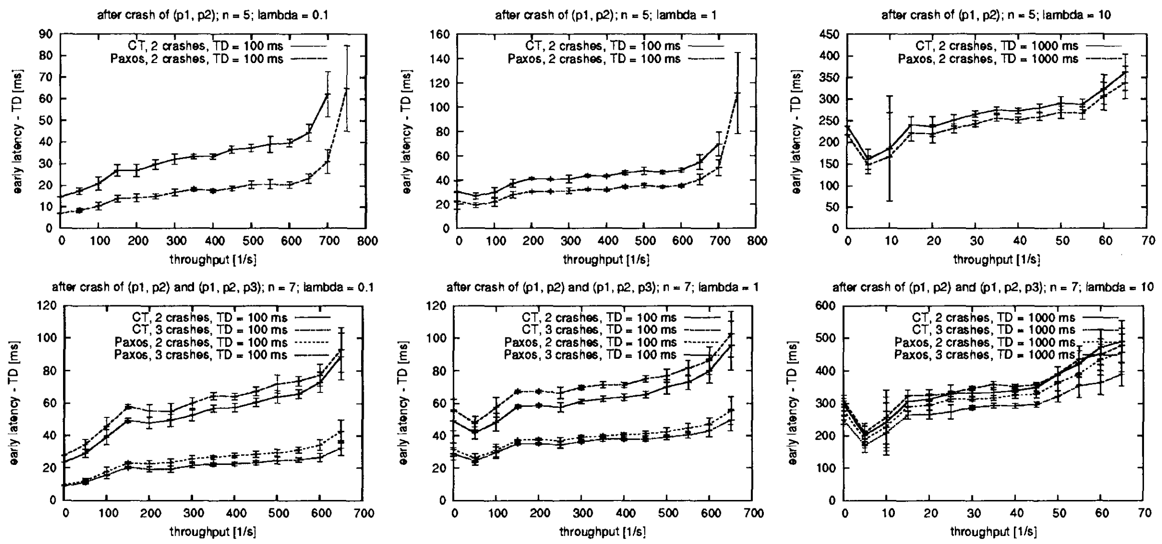

Figure 22. Latency overhead vs. throughput with the crash-transient faultload. Multiple processes crash simultaneously.

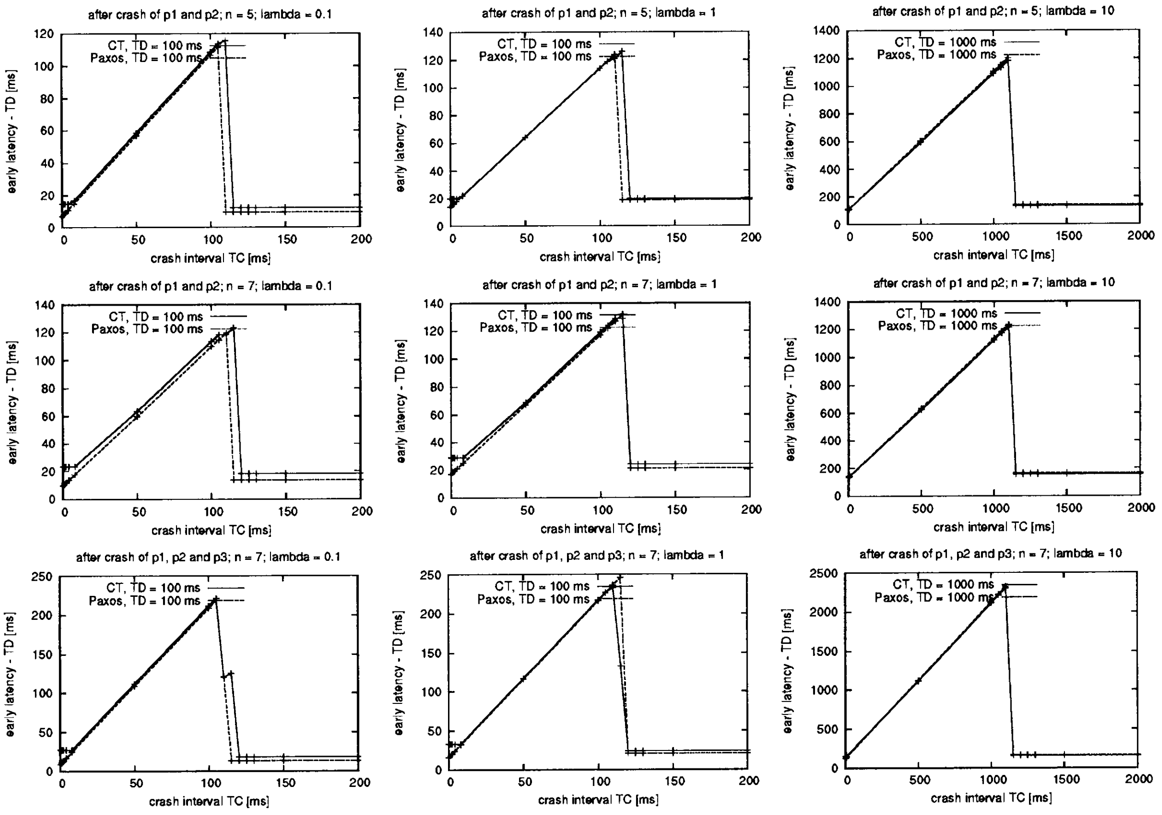

Figure 23. Latency overhead vs. crash-interval with the crash-transient faultload. Multiple processes crash in a sequenced manner.

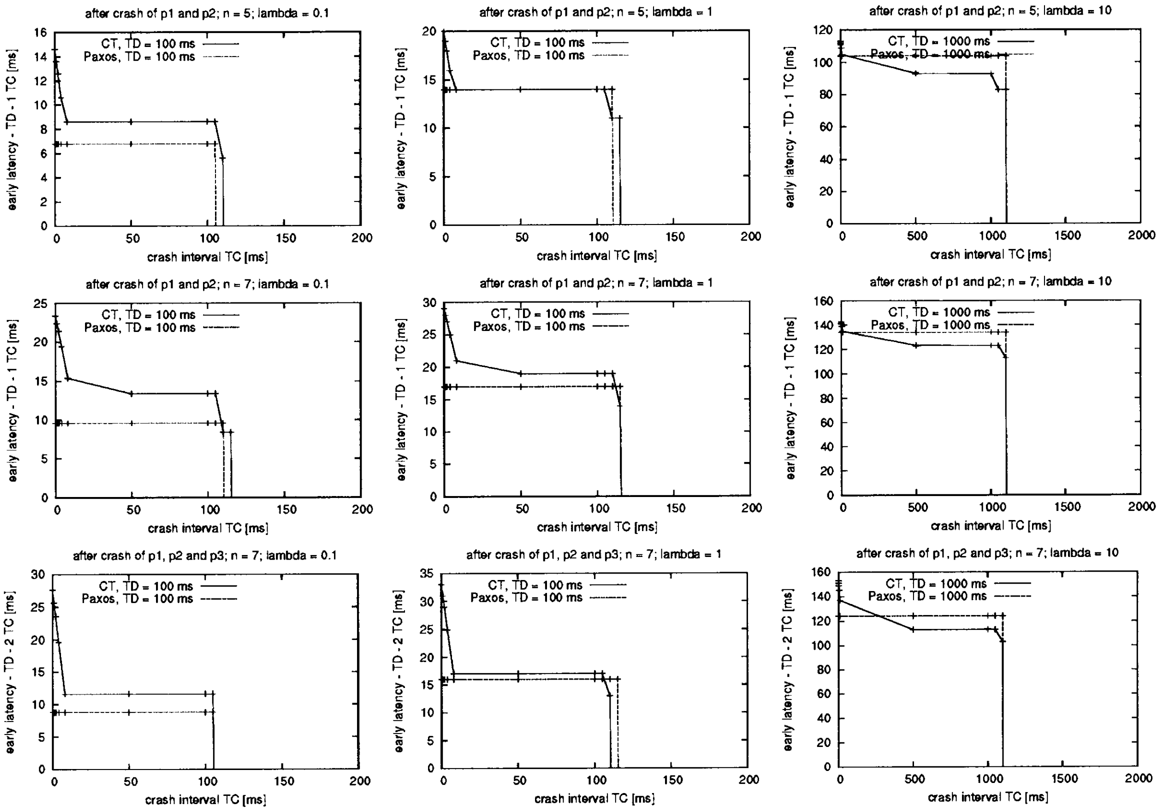

Figure 24. Latency overhead minus $(c-1) \cdot T_C$ vs. crash-interval with the crash-transient faultload. Multiple processes crash in a sequenced manner.

D.2. Graphs showing the late latency overhead

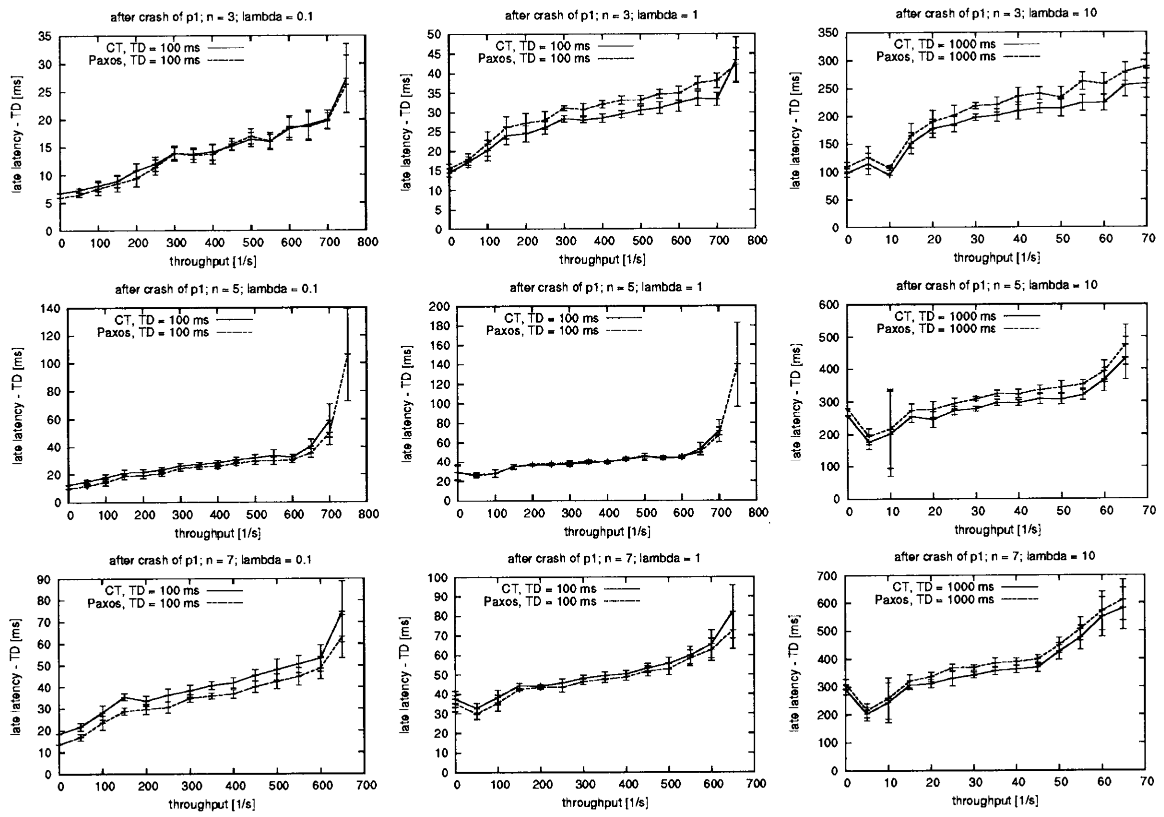

Figure 25. Latency overhead vs. throughput with the crash-transient faultload. One process crashes.

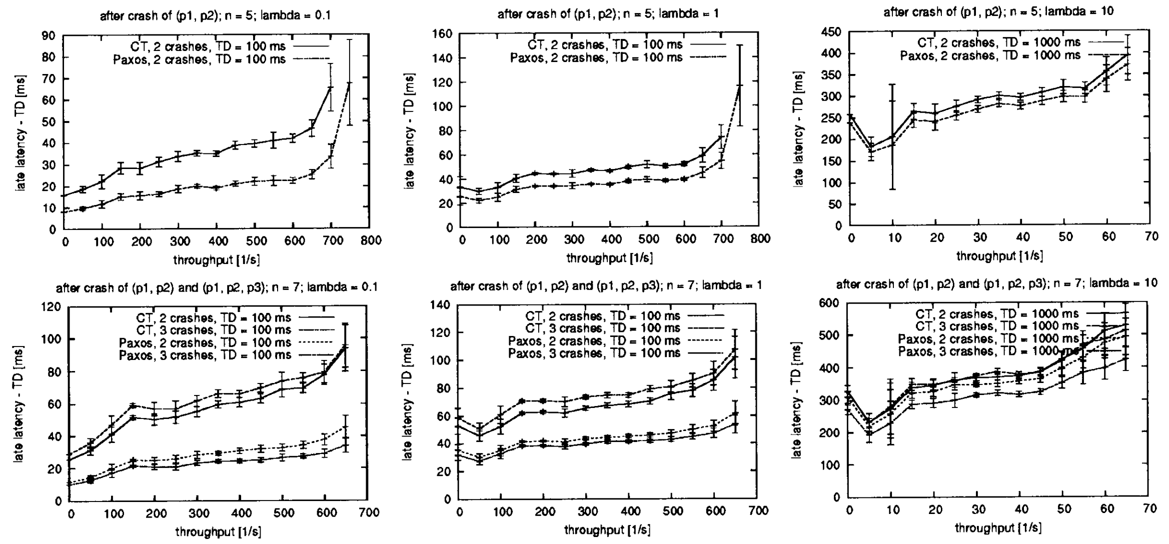

Figure 26. Latency overhead vs. throughput with the crash-transient faultload. Multiple processes crash simultaneously.

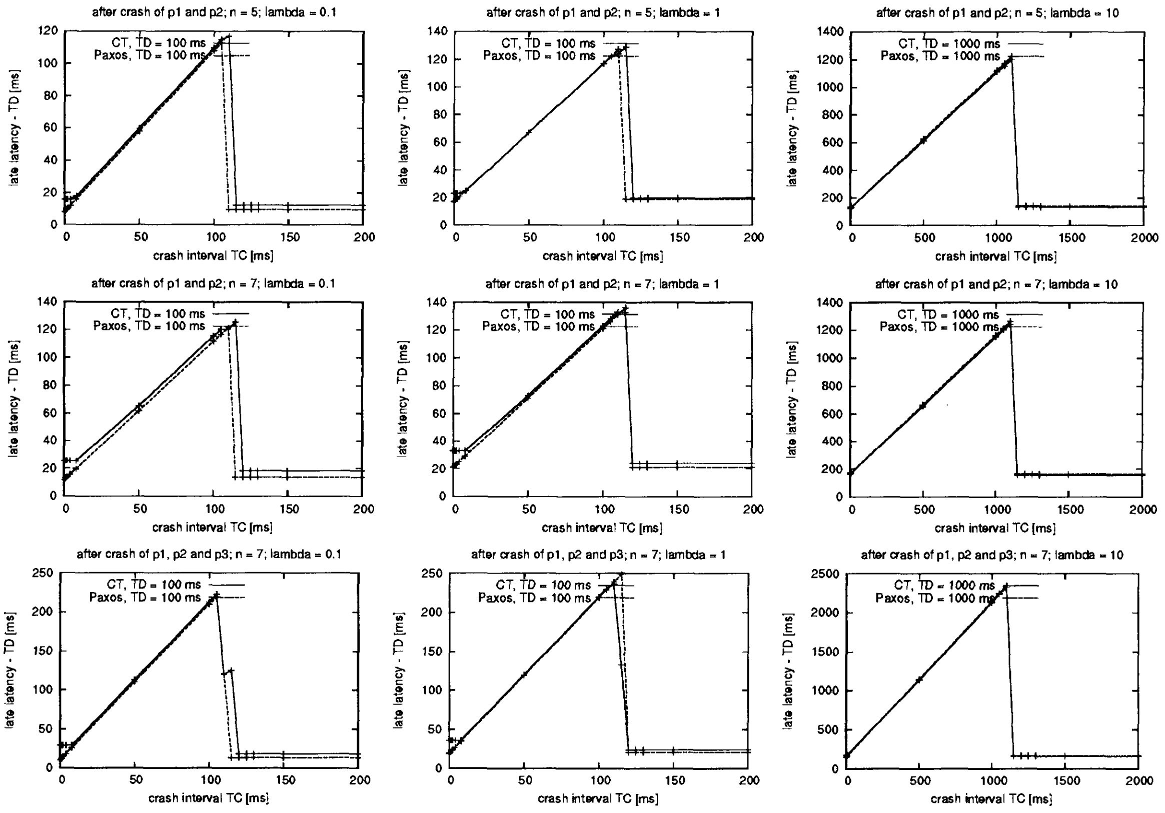

Figure 27. Latency overhead vs. crash-interval with the crash-transient faultload. Multiple processes crash in a sequenced manner.

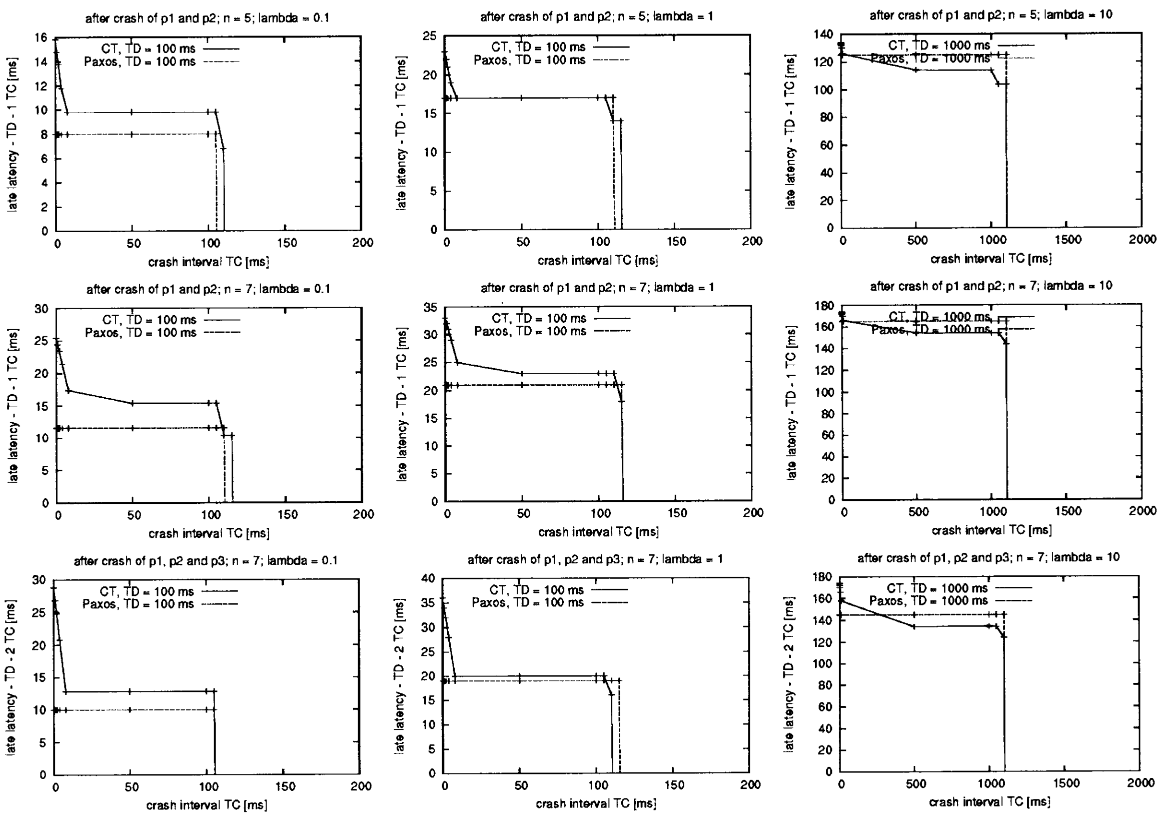

Figure 28. Latency overhead minus $(c-1) \cdot T_C$ vs. crash-interval with the crash-transient faultload. Multiple processes crash in a sequenced manner.

E. Full set of results for the suspicion-steady faultload

E.1. Graphs showing the early latency

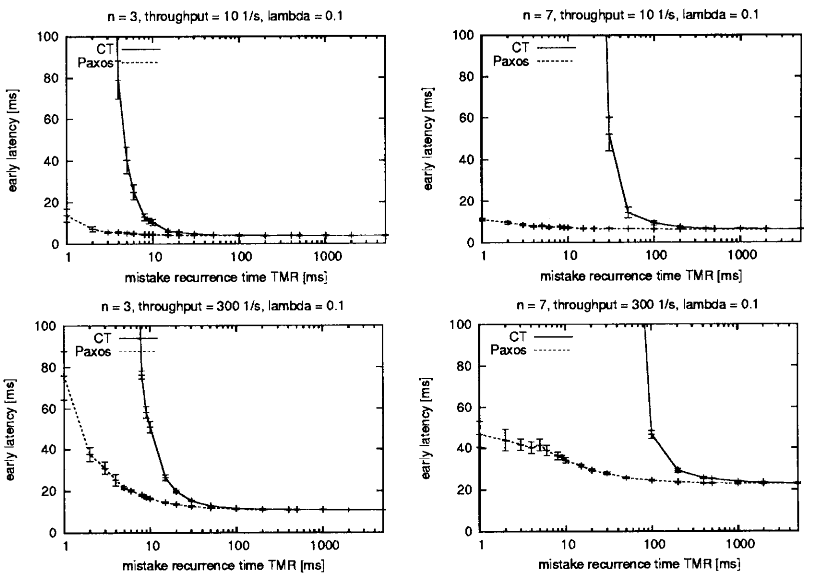

Figure 29. Latency vs. T_{MR} with the suspicion-steady faultload, with $T_M = 0$ ($\lambda = 0.1$).

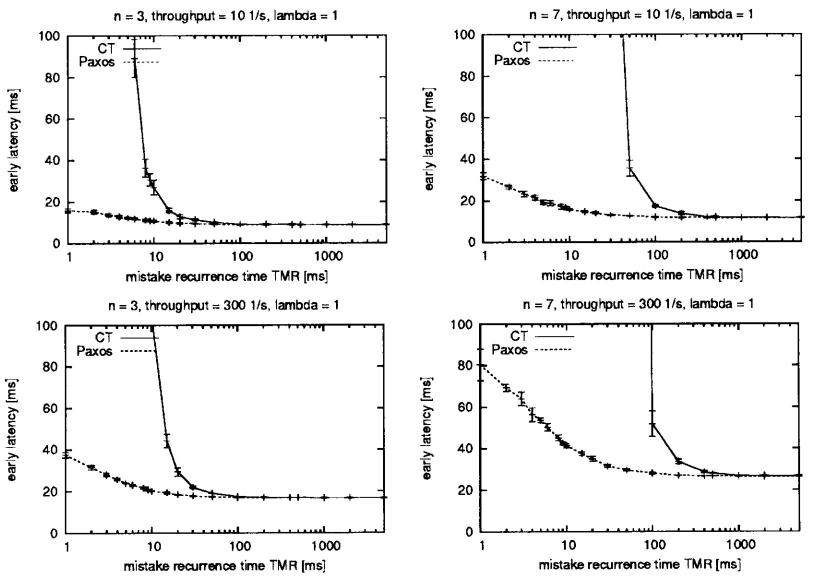

Figure 30. Latency vs. T_{MR} with the suspicion-steady faultload, with $T_M = 0$ ($\lambda = 1$).

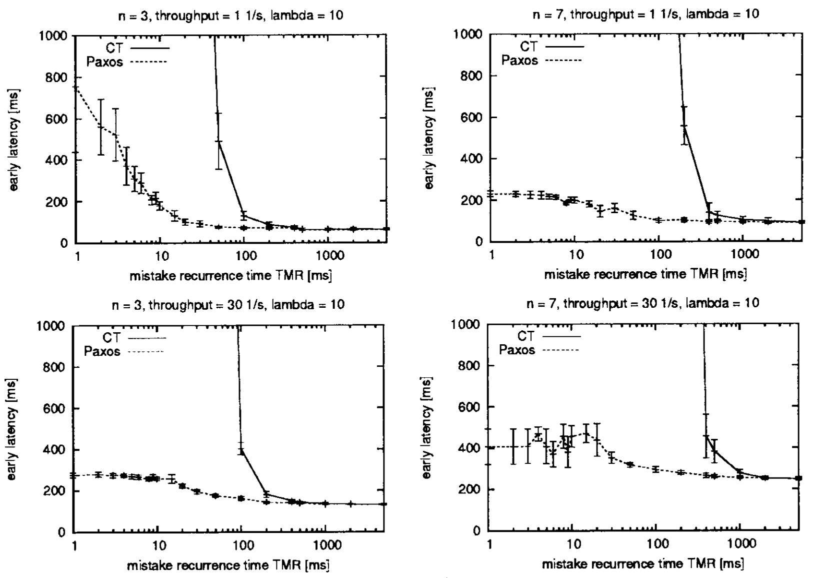

Figure 31. Latency vs. T_{MR} with the suspicion-steady faultload, with $T_M = 0$ ($\lambda = 10$).

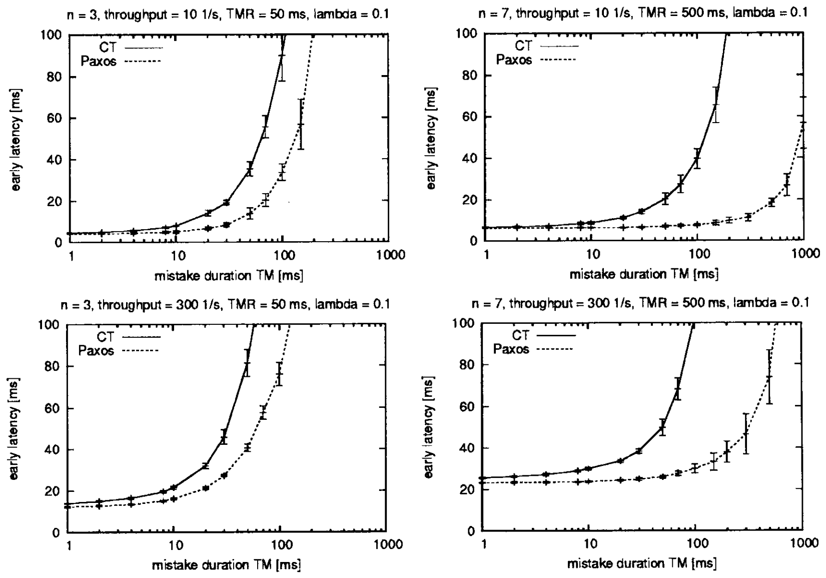

Figure 32. Latency vs. T_M with the suspicion-steady faultload, with T_{MR} fixed ($\lambda = 0.1$).

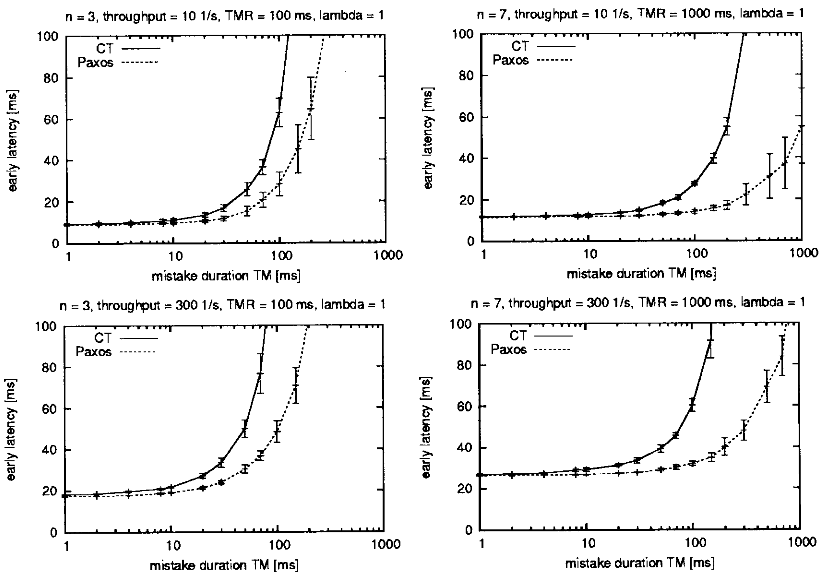

Figure 33. Latency vs. T_M with the suspicion-steady faultload, with T_{MR} fixed ($\lambda = 1$).

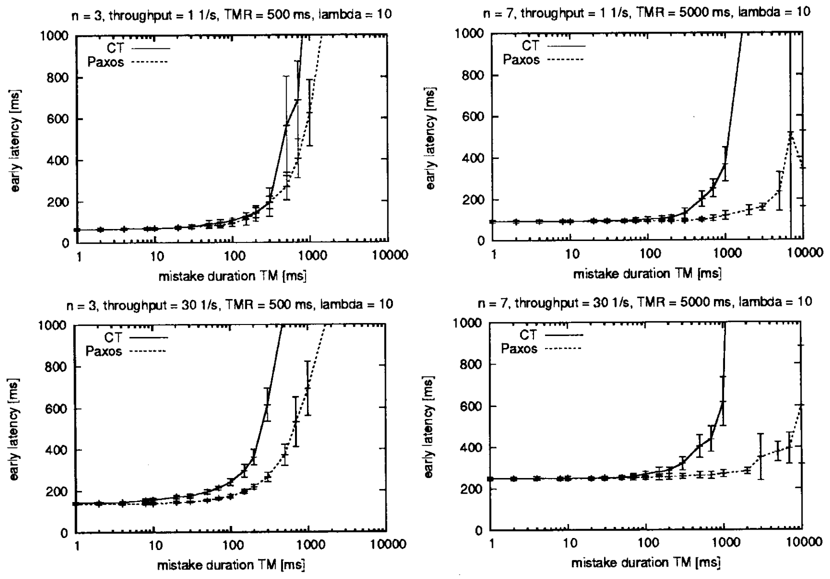

Figure 34. Latency vs. T_M with the suspicion-steady faultload, with T_{MR} fixed ($\lambda = 10$).

E.2. Graphs showing the late latency

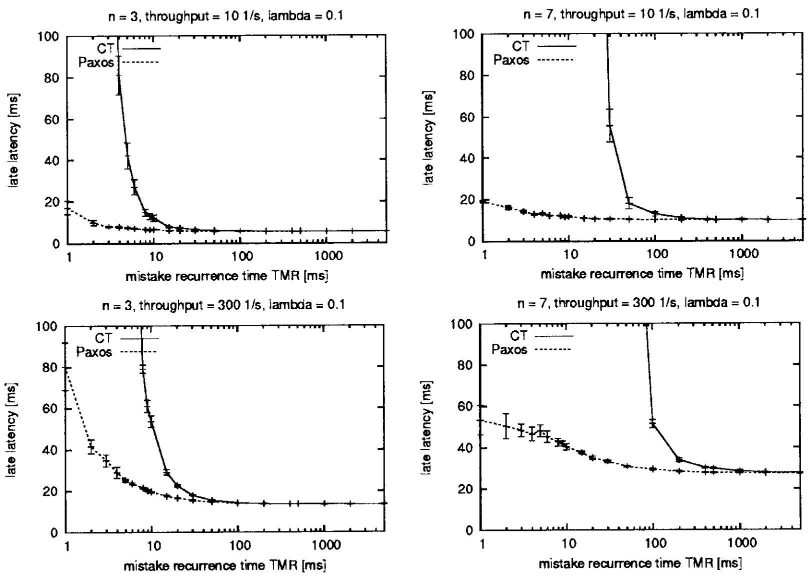

Figure 35. Latency vs. T_{MR} with the suspicion-steady faultload, with $T_M = 0$ ($\lambda = 0.1$).

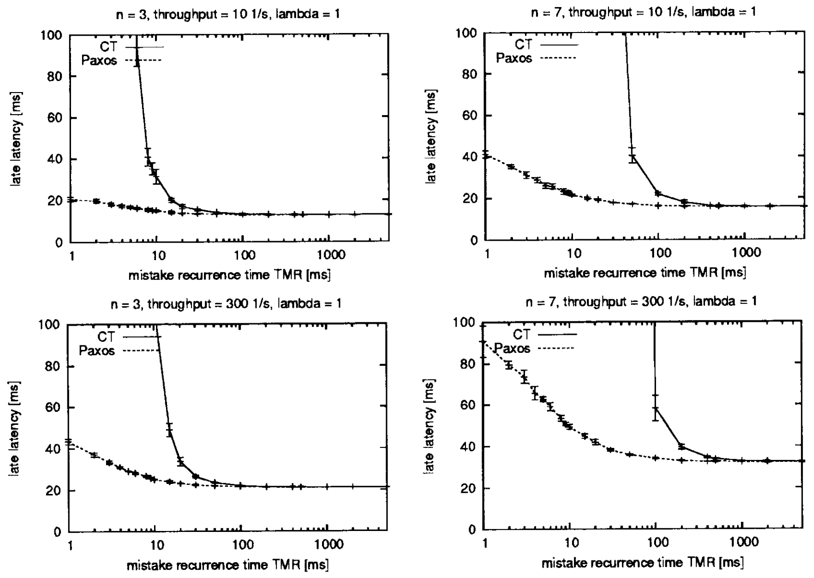

Figure 36. Latency vs. T_{MR} with the suspicion-steady faultload, with $T_M = 0$ ($\lambda = 1$).

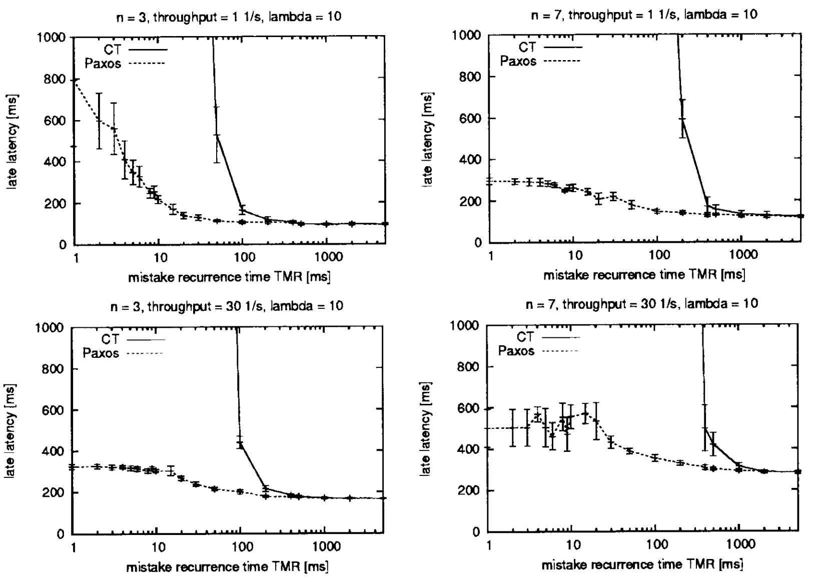

Figure 37. Latency vs. T_{MR} with the suspicion-steady faultload, with $T_M = 0$ ($\lambda = 10$).

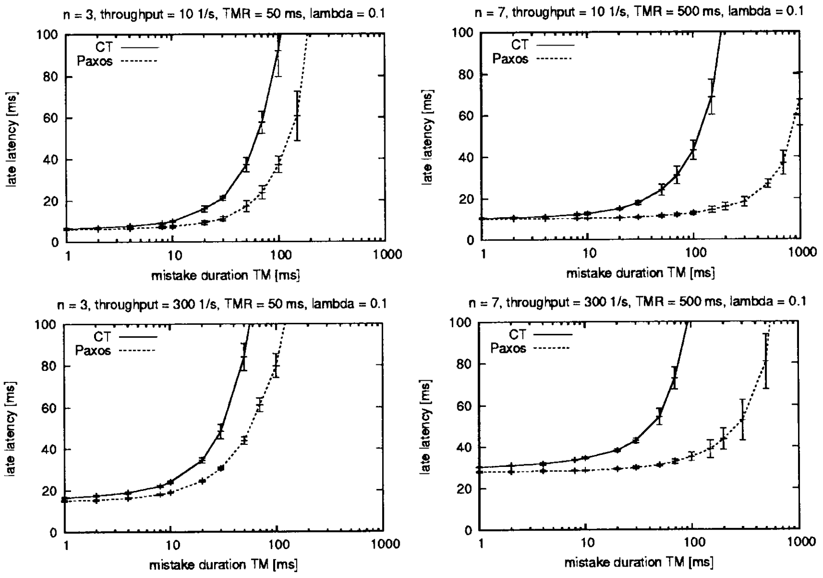

Figure 38. Latency vs. T_M with the suspicion-steady faultload, with T_{MR} fixed ($\lambda = 0.1$).

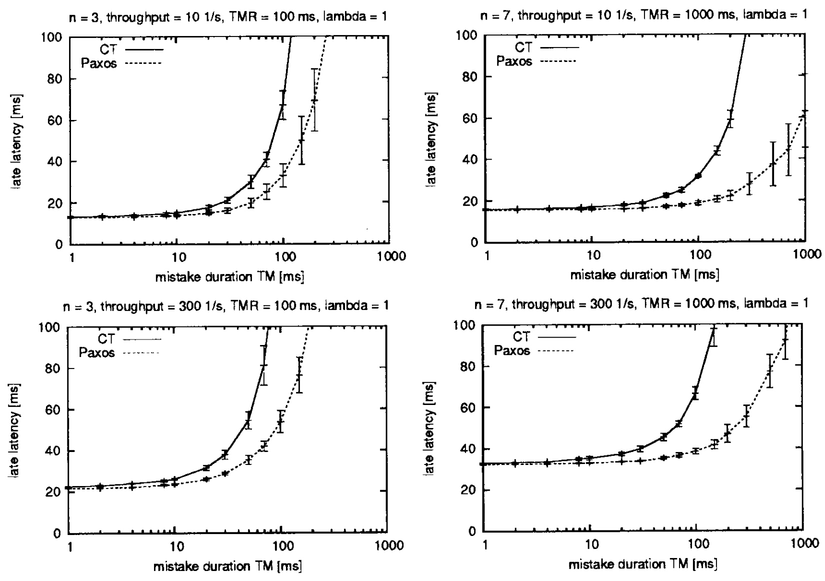

Figure 39. Latency vs. T_M with the suspicion-steady faultload, with T_{MR} fixed ($\lambda = 1$).

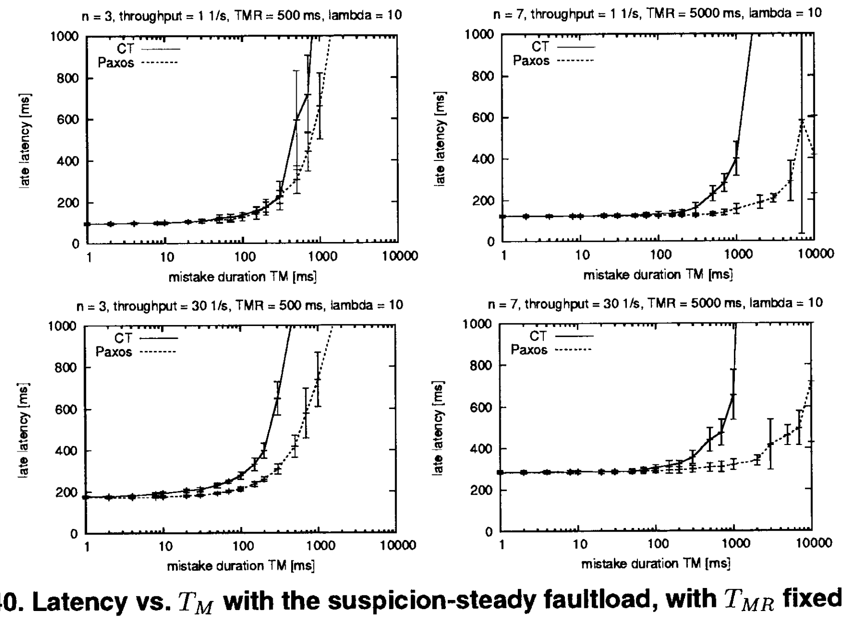

Figure 40. Latency vs. T_M with the suspicion-steady faultload, with T_{MR} fixed ($\lambda = 10$).