A Fault-Tolerant Token based Atomic Broadcast Algorithm

Richard Ekwall André Schiper
nilsrichard.ekwall@a3.epfl.ch andre.schiper@epfl.ch

***Abstract*— Many atomic broadcast algorithms have been published in the last twenty years. Token based algorithms represent a large class of these algorithms. Interestingly, all the token based atomic broadcast algorithms rely on a *group membership* service and none of them uses unreliable failure detectors directly. This paper presents the first token based atomic broadcast algorithm that uses an unreliable failure detector instead of a group membership service. It requires a system size that is quadratic in the number of supported failures. The special case of a single supported failure ($f = 1$) requires $n = 3$ processes.**

We *experimentally* evaluate the performance of this algorithm in local and wide area networks, in order to emphasize that atomic broadcast is efficiently implemented by combining a failure detector and a token based mechanism. The evaluation shows that the new token based algorithm surpasses the performance of the other algorithms in most small system settings.

I. Introduction

Atomic broadcast (or total order broadcast) is an important abstraction in fault-tolerant distributed computing. Atomic broadcast ensures that messages broadcast by different processes are delivered by all destination processes in the same order [22]. Atomic broadcast algorithms can be classified according to the mechanism used for message ordering [16]. *Token circulation* is one important ordering mechanism. In these algorithms, a token circulates among the processes, and the token holder has the privilege to order messages that have been broadcast. Additionally, sometimes only the token holder is allowed to broadcast messages.

However, the ordering mechanism is not the only key mechanism of an atomic broadcast algorithm. The mechanism used to tolerate failures is another important characteristic of these algorithms. If we consider asynchronous systems with crash failures, the two most widely used mechanisms to tolerate failures in the context of atomic broadcast algorithms are (i) *unreliable failure detectors* [9] and (ii) *group membership* [12]. For example, the atomic broadcast algorithm presented by Chandra and Toueg in [9] (together with a consensus algorithm using the failure detector $\diamond\mathcal{S}$ [9]) falls into the first category; the atomic broadcast algorithm by Birman *et al.* in [6] falls into the second category.

A. *Group membership* vs. *failure detector*

A (virtually synchronous) group membership service provides a consistent membership information to all the members of a group [29]. It can be used to *remove* processes that are suspected to have crashed (in the *primary-partition* membership). In contrast, an unreliable failure detector, *e.g.*, $\diamond\mathcal{S}$, does not provide consistent information about the failure status of processes. It can, for example, tell to process p that r has crashed, and to process q that r is alive, at the same time.

Both mechanisms can make mistakes, *e.g.*, by incorrectly suspecting correct processes. However, the overhead of a wrong failure suspicion is high when using a group membership: the suspected process is removed, a costly operation. *Depending on which process is suspected, this removal is absolutely necessary for the atomic broadcast algorithm that relies on the membership service: the notification of the removal allows the algorithm to avoid being blocked.* Moreover, to keep the same replication degree, the removal of a process is usually followed by the addition of another (or the same) process. So, with a group membership service, a wrong suspicion can lead to two costly membership operations: *removal* of a process followed by the *addition* of another process. With a failure detector, neither the removal, nor the addition is needed.

In an environment where wrong failure suspicions are frequent,[1] algorithms based on failure detectors thus have advantages over algorithms based on a group membership service. The cost difference has been evaluated through simulation in [35] in the context of two specific (not token based) atomic broadcast algorithms.

Atomic broadcast algorithms based on a failure detector have another important advantage over algorithms based on group membership: *they can be used to implement the group membership service*! Indeed, a (primary partition) group membership service can use atomic broadcast to order its views. This leads to a simple protocol stack design [27]. Such a design is not possible if atomic broadcast relies on group membership.

B. *Why token based algorithms?*

According to [3], [26], [37], token based atomic broadcast algorithms are extremely efficient in terms of throughput: the number of messages that can be delivered per time unit. The reason is that these algorithms manage to reduce network contention by using the token (1) to avoid the *ack* explosion problem (which happens if each broadcast message generates one acknowledgment per receiving process), or (2) to perform flow control (*e.g.*, a process is allowed to broadcast a message only when holding the token). However, none of the token based algorithms use failure detectors: they all rely on a group membership service (which does not necessarily appear explicitly in the algorithm, but can be implemented in an ad-hoc way, as in [26]). It is therefore interesting to try to design token based atomic broadcast algorithms that rely on failure detectors, in order to combine the advantage of failure detectors and of token based algorithms: good throughput

[1]For example, when setting timeouts used to suspect processes to small values (in the order of the average message transmission delay), to reduce the time needed to detect the crash of a process.

(without sacrificing latency) in stable environments, but adapted to frequent wrong failure suspicions.

C. Contributions of the paper.

The paper presents the first token based atomic broadcast algorithm that uses unreliable failure detectors instead of group membership. The algorithm requires a system size that is quadratic in the number of supported failures (rather than linear, as in typical atomic broadcast algorithms). In the special case of a single supported failure, only three processes are needed. The performance of this algorithm is evaluated experimentally and compared to other failure detector based atomic broadcast algorithms, both in local area networks and in wide area networks.

These results are obtained in several steps. We first give a new and more general definition for token based algorithms (Sect. II) and introduce a new failure detector, denoted by $\mathcal{R}$, adapted to token based algorithms (Sect. III). The failure detector $\mathcal{R}$ is shown to be strictly weaker than $\diamond\mathcal{P}$, and strictly stronger than $\diamond\mathcal{S}$. Although $\diamond\mathcal{S}$ is strong enough to solve consensus and atomic broadcast, $\mathcal{R}$ has an interesting feature: the failure detector module of a process p_i only needs to give information about the (estimated) state of p_{i-1}. For p_{i-1}, this can be done by sending *I am alive* messages to p_i only, which is extremely cheap compared to failure detectors where each process monitors all other processes. Moreover, in the case of three processes (a frequent case in practice, tolerating one crash), our token based algorithm works with $\diamond\mathcal{S}$.

Section IV concentrates on the consensus problem and presents a token based algorithm based on the failure detector $\mathcal{R}$. An algorithm that solves atomic broadcast is presented in Section V. The algorithm is inspired from the token based consensus algorithm of Section IV. Note that a standard solution consists in solving atomic broadcast by reduction to consensus [9]. However, this solution is not adequate here, because the resulting algorithm would be highly inefficient. Our atomic broadcast algorithm is derived from our consensus algorithm in a more complex manner. The detour through the consensus algorithm makes the explanation easier to understand.

Sections VI to IX present experimental performance comparisons between the token based atomic broadcast algorithm and several other failure detector based atomic broadcast algorithms. The evaluations are done in a local area network (Section VII), in wide area networks (Section VIII) and in large size systems (Section IX). These experimental evaluations show that the token based algorithm often surpasses the performance of the other algorithms in systems supporting one or two failures, both in the common case without failures and wrong suspicions (in local and wide area networks), and also in the case where wrong suspicions occur repeatedly.

II. System model

We assume an asynchronous system composed of n processes taken from the set $\Pi = \{p_0, \ldots, p_{n-1}\}$, with an implicit order on the processes. The k^{th} successor of a process p_i is $p_{(i+k) \ mod \ n}$, which is noted p_{i+k} for the sake of clarity. Similarly the k^{th} predecessor of p_i is simply denoted by p_{i-k}. The processes communicate by message passing over reliable channels. Processes can only fail by crashing. A process that never crashes is said to be *correct*, otherwise it is *faulty*. At most f processes are *faulty*. Finally, the system is augmented with unreliable failure detectors [9] (see below).

A. Agreement problems

The agreement problems considered in this paper are presented below.

1) Consensus: In the consensus problem, a group of processes Π have to agree on a common value based on proposals of the processes [9]. Consensus is defined by two primitives: $propose$ and $decide$. When a process p calls $propose(v)$, we say that p *proposes* v. Similarly, whenever p calls $decide(v)$, it *decides* v.

As in [9], we specify the (uniform) consensus problem by the four following properties: (1) *Termination:* Every correct process eventually decides some value, (2) *Uniform integrity:* Every process decides at most once, (3) *Uniform agreement:* No two processes (correct or not) decide a different value, and (4) *Uniform validity:* If a process decides v, then v was proposed by some process in Π.

2) Atomic broadcast: In the atomic broadcast problem, defined by the primitives $abroadcast$ and $adeliver$, processes have to agree on a common total order delivery of a set of messages.

Formally, we define (uniform) atomic broadcast by four properties [22]: (1) *Validity:* If a correct process p *abroadcasts* a message m, then it eventually $adelivers\ m$, (2) *Uniform agreement:* If a process $adelivers\ m$, then all correct processes eventually $adeliver\ m$, (3) *Uniform integrity:* For any message m, every process p $adelivers\ m$ at most once and only if m was previously $abroadcast$. and (4) *Uniform total order:* If some process, correct or faulty, $adelivers\ m$ before m', then every process $adelivers$ m' only after it has $adelivered\ m$.

B. Token based algorithms

In most traditional token based algorithm, processes are organized in a logical ring and, for token transmission, communicate only with their immediate predecessor and successor (except during changes in the composition of the ring). This definition is too restrictive for failure detector based algorithms. We define an algorithm to be *token based* if (1) processes are organized in a logical ring, (2) each process p_i has a failure detector module FD_i that provides information only about its immediate predecessor p_{i-1} and (3) each process only sends tokens to and receives tokens from its $f+1$ predecessors and successors, where f is the number of tolerated failures.

C. Failure detectors

We refer below to two failure detectors introduced in [9]: $\diamond\mathcal{P}$ and $\diamond\mathcal{S}$. The eventual perfect failure detector $\diamond\mathcal{P}$ is defined by the following properties: (i) *Strong completeness:* Eventually every process that crashes is permanently suspected by every correct process, and (ii) *Eventual strong accuracy:* there is a time after which correct processes are not suspected by any correct process. The $\diamond\mathcal{S}$ failure detector is defined by (i) *Strong completeness* and (ii) *Eventual weak accuracy:* there is a time after which some correct process is never suspected by any correct process.

D. Related work

As mentioned in the introduction, previous atomic broadcast protocols based on tokens need group membership or an equivalent mechanism. In Chang and Maxemchuk's Reliable Broadcast

Protocol [10], and its newer variant [26], an ad-hoc reformation mechanism is called whenever a host fails. Group membership is used explicitly in other atomic broadcast protocols such as Totem [3], the Reliable Multicast Protocol by Whetten et al. [37] (derived from [10]), and in [15].

These atomic broadcast protocols also have different approaches with respect to message broadcasting and delivery. In [10], [37], the *moving sequencer* approach is used: any process can broadcast a message at any time. The token holder then orders the messages that have been broadcast. Other protocols, such as Totem [3] or On-Demand [15] on the other hand use the *privilege based* approach, enabling only the token-holder to broadcast (and simultaneously order) messages. In these algorithms, the token is effectively broadcast to all processes, although the token ownership is passed along the processes on the ring. Both approaches can be used in the algorithm presented in this paper.

Finally, the different token based atomic broadcast protocols deliver messages in different ways. In [15], the token holder issues an "update dissemination message" which effectively contains messages and their global order. A host can deliver a message as soon as it knows that previously ordered messages have been delivered. "Agreed delivery" in the Totem protocol (which corresponds to *adeliver* in the protocol presented in this paper) is also done in a similar way. On the other hand, in the Chang-Maxemchuk atomic broadcast protocol [10], a message is only delivered once $f+1$ sites have received the message. Finally, the Train protocol presented in [14] transports the ordered messages in a token that is passed among all processes (and is in this respect related to the token based protocols presented in this paper).

Larrea *et al.* [24] also consider a logical ring of processes, with a different goal however. They use a ring for an efficient implementation of the failure detectors $\diamond\mathcal{W}$, $\diamond\mathcal{S}$ and $\diamond\mathcal{P}$ in a partially synchronous system.

Finally, the IEEE 802.4 Standard [1] defines a token-based access control protocol on top of a bus topology network, essentially implementing total order at the MAC layer.

III. Failure detector $\mathcal{R}$

For token based algorithms we define a new failure detector denoted $\mathcal{R}$ (stands for *Ring*). Given process p_i, the failure detector attached to p_i only gives information about the immediate predecessor p_{i-1}. For every process p_i, $\mathcal{R}$ ensures the following properties:

- *Completeness*: If p_{i-1} crashes and p_i is correct, then p_{i-1} is eventually permanently suspected by p_i, and
- *Accuracy*: If p_{i-1} and p_i are correct, there is a time t after which p_{i-1} is never suspected by p_i.

The *weaker/stronger* relationship between failure detectors has been defined in [9]. We show that (a) $\diamond\mathcal{P}$ is strictly stronger than $\mathcal{R}$ (denoted $\diamond\mathcal{P} \succ \mathcal{R}$) if $f > 1$, and (b) $\mathcal{R}$ is strictly stronger than $\diamond\mathcal{S}$ if $n \geq f(f+1)+1$ (denoted $\mathcal{R} \succ \diamond\mathcal{S}$).

Lemma 3.1: $\diamond\mathcal{P}$ is strictly stronger than $\mathcal{R}$ if $f > 1$.

The proof of Lemma 3.1 is presented in the appendix.

The relationship between $\mathcal{R}$ and $\diamond\mathcal{S}$ is more difficult to establish. We first introduce a new failure detector $\diamond\mathcal{S}2$ (Sect. III-A), then show that $\diamond\mathcal{S}2 \succ \diamond\mathcal{S}$ (Sect. III-B) and $\mathcal{R} \succeq \diamond\mathcal{S}2$ if $n \geq f(f+1)+1$ (Sect. III-C). By transitivity, we have $\mathcal{R} \succ \diamond\mathcal{S}$ if $n \geq f(f+1)+1$.

A. Failure detector $\diamond\mathcal{S}2$

For the purpose of establishing the relation between $\mathcal{R}$ and $\diamond\mathcal{S}$ we introduce the failure detector $\diamond\mathcal{S}2$ defined as follows:

- *Strong completeness*: Eventually every process that crashes is permanently suspected by every correct process and
- *Eventual "Double" Accuracy*: There is a time after which *two* correct processes are never suspected by any correct process.

B. $\diamond\mathcal{S}2$ strictly stronger than $\diamond\mathcal{S}$

$\diamond\mathcal{S}$ and $\diamond\mathcal{S}2$ differ in the accuracy property only: while $\diamond\mathcal{S}$ requires eventually *one* correct process to be no longer suspected by all correct processes, $\diamond\mathcal{S}2$ requires the same to hold for *two* correct processes. From the definition, it follows directly that $\diamond\mathcal{S}2 \succ \diamond\mathcal{S}$.

C. $\mathcal{R}$ stronger than $\diamond\mathcal{S}2$ if $n \geq f(f+1)+1$

We show that $\mathcal{R}$ is stronger than $\diamond\mathcal{S}2$ if $n \geq f(f+1)+1$ by giving a transformation of $\mathcal{R}$ into the failure detector $\diamond\mathcal{S}2$.

Transformation of $\mathcal{R}$ into $\diamond\mathcal{S}2$: Each process p_j maintains a set $correct_j$ of processes that p_j believes are correct.

(i) This set is updated as follows. Each time some process p_i changes its mind about p_{i-1} (based on $\mathcal{R}$), p_i broadcasts (using a FIFO reliable broadcast communication primitive [22]) the message $(p_{i-1}, faulty)$, respectively $(p_{i-1}, correct)$. Whenever p_j receives $(p_i, faulty)$, then p_j removes p_i from $correct_j$; whenever p_j receives $(p_i, correct)$, then p_j adds p_i to $correct_j$.

(iia) For process p_i, if $correct_i$ is equal to Π (no suspected process), the output of the transformation (the two non-suspected processes) is p_0 and p_1. All other processes are suspected.

(iib) For process p_i, if $correct_i$ is not equal to Π (at least one suspected process), the output of the transformation (the two non-suspected processes) is p_k and p_{k+1} such that k is the smallest index satisfying the following conditions: (a) p_{k-1} is not in $correct_i$, and (b) the f processes $p_k, \ldots, p_{k+f-1}$ are in $correct_i$. Apart from p_k and p_{k+1}, all other processes are suspected.

For example, for $n=7$, $f=2$, and $correct_i = \{p_0, p_2, p_3, p_5\}$, the non-suspected processes for p_i are p_2 and p_3. All other processes are suspected. If $correct_i = \{p_0, p_1, p_2, p_3, p_5\}$, the non-suspected processes for p_i are p_0 and p_1 (the predecessor of p_0 is p_6, not in $correct_i$). All other processes are suspected.

Lemma 3.2: Consider a system with $n \geq f(f+1)+1$ processes and the failure detector $\mathcal{R}$. The above transformation guarantees that eventually all correct processes do not suspect the same two correct processes.

Proof: **(i)** Consider t such that after t all faulty processes have crashed and each correct process p_i has accurate information about its predecessor p_{i-1}. It is easy to see that there is a time $t' > t$ such that after t' all correct processes agree on the same set $correct_i$. Let us denote this set by $correct(t')$.

(ii) The condition $n \geq f(f+1)+1$ guarantees that the set $correct(t')$ contains a sequence of f consecutive processes. Consider the following sequence of processes: 1 faulty, f correct, 1 faulty, f correct, etc. If we repeat the pattern f times, we have f faulty processes in a set of $f(f+1)$ processes. If we add one correct process to the set of $f(f+1)$ processes, there is necessarily a sequence of $f+1$ correct processes. With a sequence of $f+1$

correct processes, there is a sequence of f consecutive processes in $correct(t')$.

(iii) In the case $correct(t') = \Pi$, p_0 and p_1 are trivially correct.

(iv) In the case $correct(t') \neq \Pi$, we first of all show that p_k is correct. Consider the sequence of $f+1$ processes $p_k, \ldots, p_{k+f}$. Since there are at most f faulty processes, at least one process p_l in $p_k, \ldots, p_{k+f}$ is correct. If $p_l = p_k$, we are done. Otherwise, if p_l is correct, p_{l-1} is correct as well, since the failure detector of p_l is accurate after t' and does not suspect p_{l-1}. By the same argument, if p_{l-1} is correct, p_{l-2} is correct. By repeating the same argument at most $f-1$ times, we have that p_k is correct.

(v) In the case $correct(t') \neq \Pi$, we prove now that p_{k+1} is correct. Since p_k is correct and p_{k-1} is not in $correct(t')$ (by the selection rule of p_k and p_{k+1}), p_{k-1} is faulty. Thus, there are at most $f-1$ faulty processes in the sequence of f processes $p_{k+1},\ldots,p_{k+f}$. In the special case $f = 1$ ($\{p_{k+1}, \ldots, p_{k+f-1}\} = \emptyset$), all processes in $p_{k+1},\ldots,p_{k+f}$ are correct. In the case $f > 1$, there is a non-empty sequence $p_{k+1},\ldots,p_{k+f-1}$ in $correct(t')$. Furthermore, there are at most $f-1$ faulty processes among the f processes $p_{k+1},\ldots,p_{k+f}$. By the same argument used to show that p_k is correct, we can show that p_{k+1} is correct. ■

The transformation of $\mathcal{R}$ into $\diamond\mathcal{S}2$ ensures the *Eventual "double" accuracy* property if $n \geq f(f+1)+1$. Since all processes except two correct processes are suspected, the *Strong completeness* property also holds. Consequently, if $n \geq f(f+1)+1$ we have $\mathcal{R} \succeq \diamond\mathcal{S}2$.

D. Weaker version of $\mathcal{R}$

The following failure detector is slightly weaker than $\mathcal{R}$, but powerful enough to replace $\mathcal{R}$ in all the algorithms presented in this paper. It differs from $\mathcal{R}$ only in the *Accuracy* property:

- *Accuracy*: There is a sequence $\{p_x, \ldots, p_{x+f}\}$ of $f+1$ correct processes and a time t after which p_{i-1} is not suspected by p_i, for $i \in \{x+1, \ldots, x+f\}$.

This failure detector is not further developed in this paper, as it is more complex than $\mathcal{R}$, also requires $n \geq f(f+1)+1$ processes and provides only few additional benefits (it is, for example, still strictly stronger than $\diamond\mathcal{S}$).

IV. Token based consensus using unreliable failure detectors

The following section presents the token based consensus algorithm using failure detectors that is the basis for the atomic broadcast algorithm presented later. We first describe how the token is propagated (so that it is not lost in case of a process crash), then present the basic idea of the algorithm, before the algorithm itself.

A. Token circulation

Consensus is achieved by passing a token between the different processes. The token contains information regarding the current proposal (or the decision once it has been taken). The token is passed between the processes on the logical ring $p_0, p_1, \ldots, p_{n-1}$.

To avoid the loss of the token due to crashes, process p_i sends the token to its $f+1$ successors in the ring, $p_{i+1}, \ldots, p_{i+f+1}$.[2]

[2]The token should be seen as a *logical* token. Multiple backup copies circulate in the ring, but they are discarded by the algorithm if no suspicion occurs. Henceforth, the *logical* token will simply be referred to as "the token".

The algorithm is expressed as a sequence of rounds. In each round a single process sends its token: process p_i can only send a token in rounds $i + k \cdot n$ with $k \geq 0$. Since there are n processes, a complete revolution of the token requires n rounds. For example, in a system with 3 processes, process p_0 sends its first token in round 0. Processes p_1 and p_2 then send a token for the first time in rounds 1 and 2, respectively. The second revolution of the token starts when p_0 sends the token in round 3.

When awaiting the token for round r, process p_i first waits to get the token from p_{i-1} (sent in round $r-1$). If p_{i-1} crashes, p_i would wait forever. To avoid this, p_i accepts the token from any of its predecessors, if it suspects p_{i-1} (line 1 of Algorithm 1).

B. Token based consensus algorithm

1) Basic idea: Each token holder "votes" for the proposal in the token and then sends it to its neighbors. As soon as a sufficient number of token holders have voted for some proposal v, then v is decided. The decision is then propagated as the token circulates along the ring.

The algorithm can however not blindly increase the votes. We say that there is a *gap* in the token circulation if the token received by process p_i in round $round_i$ is not from p_{i-1} (*i.e.*, the token was not sent in round $round_i - 1$, but in some round before that). Now, upon receiving the token, a process does the following (see Algorithm 1): as soon as there is a gap in the token circulation, $votes$ is reset by the receiver p_i (line 3). After that, $votes$ is incremented (line 4). If at that point $votes$ is greater than the vote threshold $f+1$, p_i decides on the estimate of the token (lines 5 and 6). The decision is then propagated with the token (line 7, with some details omitted). Note that a decision is only taken if f successive processes receive the token from their predecessor (and therefore increment the votes without resetting them).

Algorithm 1 Basic token handling by p_i

```
1: upon receiving ⟨$votes, est, round$⟩ s.t.
         ($round = round_i - 1$) or
         ($p_{i-1} \in \mathcal{D}_{p_i}$ and
          $round_i - f - 1 \leq round \leq round_i - 2$) do
2:    if $round \neq round_i - 1$ then        {gap in the token circulation}
3:       $votes \leftarrow 0$                          {reset token}
4:    $votes \leftarrow votes + 1$
5:    if $votes \geq f + 1$ then
6:       $decide(est)$
7:    send ⟨$votes, est, round_i$⟩ to $\{p_{i+1}, \ldots, p_{i+f+1}\}$
```

2) Conditions for Agreement vs. Termination: In the above algorithm, where votes are reset as soon as a gap in the token circulation is detected, *Agreement* holds if the vote threshold is greater or equal to $f+1$. *Termination* additionally requires the failure detector $\mathcal{R}$ and that there be at least $n \geq (f+1)f+1$ processes in the system.

3) Detailed algorithm: The token contains the following fields: $round$ (round number), est (current estimate value), $votes$ (accumulated votes for the est value) and $decided$ (a boolean indicating if est is the decision).

Furthermore, each process p_i has two variables: $round_i$ that contains the next sending round of p_i and $decision_i$ that stores the value of the decision (or $\perp$ if p_i has not decided yet).

The initialization code is given by lines 1 to 8 in Algorithm 2. Lines 4-6 show p_0 sending the initial token (in round 0, with v_0

Algorithm 2 Token based consensus (code of p_i)

```
1:  upon $propose(v_i)$ do
2:      $decision_i \leftarrow \perp$;  $round_i \leftarrow i$
3:      constant $destSet_i \leftarrow \{p_{i+1}, \dots, p_{i+f+1}\}$
4:      if $p_i = p_0$ then {send token with $\langle round, est, votes, decided \rangle$ }
5:          send $\langle 0, v_0, 1, false \rangle$ to $\{p_1, \dots, p_{f+1}\}$
6:          $round_0 \leftarrow n$
7:      else if $p_i \in \{p_{n-f}, \dots, p_{n-1}\}$ then   {send "dummy" token}
8:          send $\langle -1, v_i, 0, false \rangle$ to $\{p_1, \dots, p_{i+f+1}\}$

    Token handling by $p_i$:
9:  upon receiving $\langle round, est, votes, decided \rangle$ s.t.
        $(round = round_i - 1)$ or
        $(p_{i-1} \in \mathcal{D}_{p_i}$ and
         $round_i - f - 1 \leq round \leq round_i - 2)$ do
10:     if $decision_i \neq \perp$ then                 {$p_i$ has already decided}
11:         send $\langle round_i, decision_i, 0, true \rangle$ to $destSet_i$
12:     else                                            {$p_i$ has not decided yet}
13:         if $(round \neq round_i - 1)$ then
14:             $votes \leftarrow 0$                     {reset votes if gap}
15:         $votes \leftarrow votes + 1$                 {add vote of $p_i$}
16:         if $(votes \geq f + 1)$ or $decided$ then
17:             $decision_i \leftarrow est$             {decide if enough votes}
18:             $decide(decision_i)$; $decided \leftarrow true$
19:         send $\langle round_i, est, votes, decided \rangle$ to $destSet_i$
20:     $round_i \leftarrow round_i + n$                 {set next sending round}
21: upon receiving $\langle round, est, -, decided \rangle$ s.t.
        $round < round_i - n$ do
22:     if $decided$ and $decision_i = \perp$ then       {decision in token}
23:         $decision_i \leftarrow est$
24:         $decide(decision_i)$
```

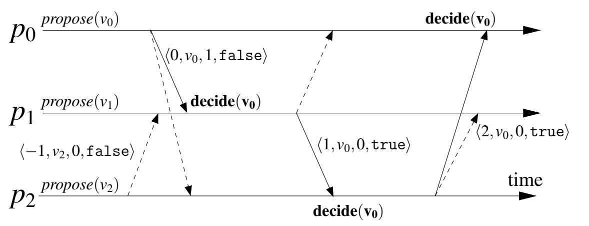
as estimate). Lines 7-8 show the *dummy* token sent to prevent blocking in case processes $p_0, \dots, p_{f-1}$ are initially crashed. A dummy token has $round = -1$ and $votes = 0$, and is sent only to processes $\{p_1, \dots, p_f\}$. The estimate est of this token is the proposed value v_i of the sender process p_i.

The token handling code is given by lines 9 to 24 in Algorithm 2. At line 9, process p_i starts by receiving the token (from p_{i-1}, or another process if p_{i-1} is suspected) for the expected $round_i$. If p_i has already decided in a previous round, then p_i directly sends a token with the decision (line 11).

If p_i has not yet decided, then p_i starts by detecting if there is a gap in the token circulation (line 13) and resets the votes if it is the case (line 14). The votes are then incremented (line 15).

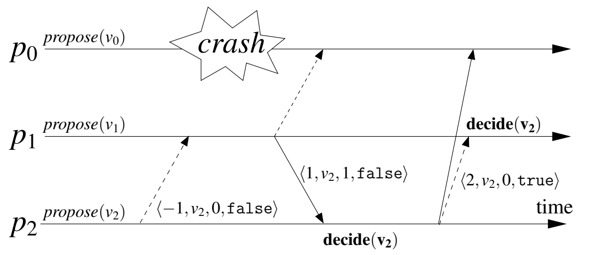
On line 16, p_i verifies if enough votes have been collected ($votes \geq f + 1$) or if the token already contains a decision ($decided = true$). If one of these conditions is true, p_i decides (lines 17-18). Process p_i then sends its token to the $f+1$ sucessors on line 19 and increments its next sending round $round_i$ by n.

Lines 9-20 ensure that at least one correct process eventually decides. However, if $f > 1$, this does not ensure that all correct processes eventually decide. Consider the following example: p_i is the first process to decide, p_{i+1} is faulty. In this case, p_{i+2} may always receive the token from p_{i-1}, a token that does not carry a decision; p_i might be the only process to ever decide. Lines 21-24 ensure that every correct process eventually decides. Note that the stopping of the algorithm is not discussed here. It can easily be added.



p_0
propose(v_0)
decide(v_0)
$\langle 0, v_0, 1, \texttt{false} \rangle$
p_1
propose(v_1)
decide(v_0)
$\langle -1, v_2, 0, \texttt{false} \rangle$
$\langle 1, v_0, 0, \texttt{true} \rangle$
$\langle 2, v_0, 0, \texttt{true} \rangle$
p_2
propose(v_2)
time
decide(v_0)

(a) No crash, no suspicion



p_0
propose(v_0)
crash
p_1
propose(v_1)
decide(v_2)
$\langle 1, v_2, 1, \texttt{false} \rangle$
$\langle 2, v_2, 0, \texttt{true} \rangle$
p_2
propose(v_2)
$\langle -1, v_2, 0, \texttt{false} \rangle$
time
decide(v_2)

(b) p_0 crashes

Fig. 1. Example execution of the consensus algorithm

4) Example run of the algorithm: Figure 1 presents an example execution of the consensus algorithm in a system with $n = 3$ processes. The dashed arrows correspond to "backup" tokens (that are used only when failures or suspicions occur) whereas the solid arrows show the main token (transmitted between process p_i and p_{i+1}). When no crashes nor suspicions occur (Figure 1(a)), process p_1 receives p_0's token and increments the votes for p_0's proposal v_0. With two votes, p_1 decides v_0. The token is then passed on to p_2 (with the *decided* flag set to `true`) that decides. Finally, p_0 decides after the last token transmission.

In the case of a crash of p_0, p_1 eventually suspects p_0 and thus accepts p_2's token (with p_2's proposal v_2). Since there is a gap in the token circulation, the votes are reset (no decision can be taken) and the token is sent from p_1 to p_2. Process p_2 receives the token, increments the votes and decides v_2. Process p_1 then decides one communication step later after receiving the backup token (with the decision flag).

5) Proof of the token based algorithm: The proofs of the *Uniform validity* and *Uniform integrity* properties are easy and omitted. We prove only the *Uniform agreement* and *Termination* properties.

Proof: (Uniform agreement) Let p_i be the first process to decide (say in round r), and let v be the decision value. By line 16 of Algorithm 2, we have $votes \geq f + 1$. Votes are reset for each gap. So, $votes \geq f + 1$ ensures that all processes $p_j \in \{p_{i-f}, \dots, p_{i-1}\}$, sent a token with $est = v$ in rounds $r - f, \dots, r - 1$ respectively.

Any process p_k, successor of p_i in the ring, receives the token from one of the processes $p_i, \dots, p_{i-f}$. Since all these processes sent a token with $est = v$, the token received by p_k necessarily carries the estimate v. So after round r, the only value carried by the token is v, *i.e.*, any process that decides will decide v. ∎

Proof: (Termination) Assume at most f faulty processes and the failure detector $\mathcal{R}$. We show that, if $n \geq f(f + 1) + 1$, then every correct process eventually decides.

First it is easy to see that the token circulation never stops: if p_i is a correct process that does not have the token at time t, then there exists some time $t' > t$ such that p_i receives the token at time t'. This follows from (1) the fact that the token is sent by a process to its $f+1$ successors, (2) the token reception procedure (line 9 in Algorithm 2), and (3) the completeness property of $\mathcal{R}$ (which ensures that if p_i waits for the token from p_{i-1} and p_{i-1} has crashed, then p_i eventually suspects p_{i-1} and accepts the token from any of its $f+1$ predecessors).

The second step is to show that at least one correct process eventually decides. Assume the failure detector $\mathcal{R}$, and let t be such that after t no correct process p_i is suspected by its immediate correct successor p_{i+1}. Since we have $n \geq f(f+1)+1$ there is a sequence of $f+1$ correct processes in the ring. Let $p_i, \ldots, p_{i+f}$ be this sequence. After t, processes $p_{i+1}, \ldots, p_{i+f}$ only accept the token from their immediate predecessor. Thus, after t, the token sent by p_i is received by p_{i+1}, the token sent by p_{i+1} is received by p_{i+2}, and so forth until the token sent by p_{i+f-1} is received by p_{i+f}. Once p_{i+f} has executed line 15 of Algorithm 2, we have $votes \geq f+1$. Consequently, p_{i+f} decides.

Finally, if one correct process p_k decides, and sends the token with the decision to its $f+1$ successors, the first correct successor of p_k, by line 21 or line 9, eventually receives the token with the decision and decides (if it has not yet done so). By a simple induction, every correct process eventually also decides. ■

V. Token based atomic broadcast using unreliable failure detectors

In this section we show how to transform the token based consensus algorithm into an atomic broadcast algorithm. Note that we could have presented the atomic broadcast algorithm directly. However, since the consensus algorithm is simpler than the atomic broadcast algorithm, we believe that a two-step presentation makes it easier to understand the atomic broadcast algorithm.

Note also that it is well known how to solve atomic broadcast by reduction to consensus [9]. However, the reduction, which transforms atomic broadcast into a sequence of consensus, yields an inefficient algorithm here. The reduction would lead to multiple instances of consensus, with one token per consensus instance. We want a single token to "glue" the various instances of consensus together. A variation of the algorithm that follows is presented in [19]. The algorithm presented here is easier to understand, with processes that send regular messages (at line 10) and tokens (versus only tokens in [19]).

To be correct, the atomic broadcast algorithm requires the failure detector $\mathcal{R}$, a number of processes $n \geq f(f+1)+1$, and a vote threshold at $f+1$ in order to decide, as was the case in the consensus algorithm above.

A. Overview

In the token based atomic broadcast algorithm, the token transports sets of messages. More precisely, the token carries the following information: $\langle round, ordering, votes, ordered \rangle$.

The $round$ and $votes$ fields are the same as in the consensus algorithm. The set of messages $ordering$ is the current proposal (these messages are delivered as soon as a sufficient number of consecutive "votes" have been collected). The field $ordered$ is the set of all ordered messages that the token is aware of (*i.e.*, a set of consensus decisions: pairs associating a sequence number to a set of messages).

Algorithm 3 Token based atomic broadcast (code of p_i)

```
1:  Initialization:
2:    $unordered_i \leftarrow \emptyset$; $ordered_i \leftarrow \emptyset$: $round_i \leftarrow 0$
3:    $nextCons_i \leftarrow 1$
4:    if $p_i = p_0$ then    {token: $\langle round, ordering, votes, ordered \rangle$ }
5:      send $\langle 0, unordered_0, 1, \emptyset \rangle$ to $\{p_1, \ldots, p_{f+1}\}$
6:      $round_0 \leftarrow n$
7:    else if $p_i \in \{p_{n-f}, \ldots, p_{n-1}\}$ then    {send "dummy" token}
8:      send $\langle -1, \emptyset, 0, \emptyset \rangle$ to $\{p_1, \ldots, p_{i+f+1}\}$

9:  To execute $abroadcast(m)$:
10:   send $m$ to all.
11: upon delivering $m$ do
12:   if $m \notin \{msgs \mid (msgs, -) \in ordered_i\}$ then
13:     $unordered_i \leftarrow unordered_i \cup \{m\}$
14: procedure delivery($seq$):
15:   while $\exists(nextCons_i, msgs) \in seq$ do
16:     $ordered_i \leftarrow ordered_i \cup \{(nextCons_i, msgs)\}$
17:     $unordered_i \leftarrow unordered_i \setminus msgs$
18:     $adeliver$ messages in $msgs$ in a deterministic order
19:     $nextCons_i \leftarrow nextCons_i + 1$

    Token handling by $p_i$:
20: upon receiving $\langle round, ordering, votes, ordered \rangle$ s.t.
          $(round = round_i - 1)$ or
          $(p_{i-1} \in \mathcal{D}_{p_i}$ and
           $round_i - f - 1 \leq round \leq round_i - 2)$ do
21:   if $|ordered| < |ordered_i|$ then    {"old" token}
22:     $ordering \leftarrow \emptyset$
23:   else    {token with new information}
24:     delivery($ordered$)
25:     if $(round \neq round_i - 1)$ or $(ordering = \emptyset)$ then
26:       $votes \leftarrow 0$    {reset votes}
27:     $votes \leftarrow votes + 1$
28:     if $votes \geq f + 1$ then
29:       delivery($\{(nextCons_i, ordering)\}$)
30:       $ordering \leftarrow \emptyset$
31:   if $ordering = \emptyset$ then    {new proposal}
32:     $ordering \leftarrow unordered_i$
33:     $votes = 1$
34:   token $\leftarrow \langle round_i, ordering, votes, ordered_i \rangle$
35:   send token to $\{p_{i+1}, \ldots, p_{i+f+1}\}$
36:   $round_i \leftarrow round_i + n$
37: upon receiving $\langle round, -, -, ordered \rangle$ s.t.
          $round < round_i - n$ do
38:   if $|ordered| > |ordered_i|$ then
39:     delivery($ordered$)
```

B. Details

Each process p_i manages the following data structures (see Algorithm 3): $round_i$ (the current round number), $unordered_i$ (the set of all messages that have been $abroadcast$ but not yet ordered), $ordered_i$ (the set of all ordered messages that p_i knows of, represented as a set of (consensus number, set of messages) pairs) and $nextCons_i$ (the next attributable sequence number).

Lines 1 to 19 of Algorithm 3 present the initialization of the atomic broadcast algorithm, as well as the $abroadcast$ and

$adelivery$ of messages. Lines 1 to 8 correspond to lines 1 to 8 of the consensus algorithm (Algorithm 2). The **delivery**(seq) procedure is called in the token handling part of Algorithm 3.

From line 20 and on, Algorithm 3 describes the token handling. Lines 21 to 30 of Algorithm 3 correspond to lines 10 to 18 of Algorithm 2. The procedure **delivery**($\ldots$) is called to $adeliver$ messages (line 29). When this happens, a new sequence of messages can be proposed for delivery. This is done at lines 31 to 33. Finally, lines 37 to 39 handle the reception of other tokens. This is needed for *Uniform agreement* when $f > 1$ (these lines play the same role as lines 22-24 in Algorithm 2).

The proof of the algorithm can be derived from the proof of the token based consensus algorithm.

C. Optimizations

The following paragraphs present the optimizations that were applied to the token based atomic broadcast algorithm presented in Algorithms 3. The performance figures presented later include these optimizations.

First of all, in Algorithm 3, the token carries entire messages, rather than only message identifiers. The algorithm can be optimized so that only the message identifiers are included in the token. This can be addressed by adapting techniques presented in other token based atomic broadcast algorithms, *e.g.*, [10], [26], and is thus not discussed further.

The optimization above reduces the size of the token but does not prevent it from growing indefinitely. This can be handled as follows. Consider a process p that receives the token with s_1 the highest sequence number in the $ordered$ set and later, in a different round, receives the token with a sequence number $s_2 > s_1$ in the same field. When p receives the token with the messages ordered at sequence number s_2, at least $f+1$ processes (and so at least one correct process) have received a token containing messages up to s_1. All pairs $(s, msgs)$ with $s \leq s_1$ can thus be removed from the $ordered$ set in the token. In good runs (no failures, no suspicions), this means that a process that $adelivers$ new messages in round i (thus increasing the size of the $ordered$ set in the token) then removes those messages from the token in round $i + n$.

The circulation of the token can also be optimized. If all processes are correct, each process actually only needs to send the token to its immediate successor. So, by default each process p_i only sends the token to p_{i+1}. This approach requires that if process p_i suspects its predecessor p_{i-1}, it must send a message to its predecessors p_{i-f} to p_{i-2}, requesting the token.[3] A process, upon receiving such a message, sends the token to p_i. If all processes are correct, this optimization requires only a single copy of the token to be sent by each token holder instead of $f + 1$ copies, thus reducing the network contention by a factor $f + 1$.

Furthermore, in Algorithm 3, the set of $adelivered$ messages are only transported in the token (in the $ordered$ field). As a consequence, the token has to perform a complete revolution for all processes to $adeliver$ a given message. This leads to high latencies (*i.e.*, the time between $abroadcast(m)$ and $adeliver(m)$), especially as the size of the system (and thus the token ring) increases. To achieve a lower latency, a process p_i that executes line 29 sends the pair $(nextCons_i, ordering)$ to all other processes. Upon receiving this message, the other processes can $adeliver$ the messages in the $ordering$ set without having to wait for the next token.

Finally, in Algorithm 3, a single proposal is contained in the token. The proposal contains a batch of messages to be ordered and a decision on a batch can be taken at the earliest f communication steps after the batch is proposed. As f increases, each decision requires more and more steps (and so messages are $adelivered$ slower and slower).

To achieve higher throughputs, it is thus essential to be able to have several proposals in a single token, *i.e.* propose a new batch of messages *before* the last one has been $adelivered$. To do this we allow the token to contain several proposals, with separate votes for each proposal (instead of a single proposal with a single votes variable in Algorithm 3). Moreover, the proposal also contains the consensus number in which it was proposed. This consensus number is used as a tie-breaker (to ensure *Uniform total order*) whenever several proposals reach $f + 1$ votes at the same time. The proposals with smallest consensus numbers are $adelivered$ first.

VI. Experimental performance evaluation

The performance of the token based atomic broadcast algorithm presented above has been evaluated in simulation in [19]. This simulation showed that the performance of the new algorithm is better than previous algorithms using failure detectors.

As with any simulation, the performance results strongly depend on the chosen model. To further confirm the good performance of the new algorithm, we *experimentally* evaluate the token based algorithm under varying conditions in *real* networks. We consider both local and wide area networks, with several different system sizes and network link characteristics. The new token based algorithm is once again compared to the same two failure detector based algorithms as in [19].

We focus on the case of a system without any process failures and examine the situations where (1) no suspicions occur and those where (2) wrong suspicions occur repeatedly. Situations (1) and (2) assess, respectively, the two desired properties of the new token based algorithm: high reachable throughputs in good runs (which are common) and a good performance in a system with frequent wrong failure suspicions.

A. Algorithms

We now present the atomic broadcast algorithm that is compared with the token based atomic broadcast algorithm presented in Section V.

The atomic broadcast algorithm proposed by Chandra and Toueg [9] reduces atomic broadcast to a sequence of consensus executions. The algorithm is shortly reminded below.

Whenever a message m is $abroadcast$, it is first reliably broadcast to all processes. The order of the $abroadcast$ messages that have not yet been $adelivered$ is then determined by consecutive consensus executions $1, 2, 3$, etc. Each consensus execution is performed on a set of undelivered messages. To $adeliver$ a message m that is $abroadcast$, the algorithm thus needs one reliable broadcast and one consensus execution. The cost (in terms of communication steps and sent messages) of $adelivering$ an application message depends on the choice of the underlying consensus and reliable broadcast algorithms.

[3]This request should only be sent once during each round, to avoid an explosion of request messages in the case of very frequent wrong suspicions.

In our performance study, we consider the reliable broadcast algorithm presented in [9], that requires one communication step and n^2 messages per reliable broadcast. Furthermore, we consider the two consensus implementations that were already used in the simulated performance study in [19]. Both algorithms use an unreliable failure detector $\diamond\mathcal{S}$ to solve consensus and require at least a majority of correct processes to reach a decision. The characteristics of the two consensus algorithms are shortly recalled in the following paragraphs.

1) Chandra-Toueg consensus [9]: The Chandra-Toueg algorithm solves consensus using a centralized communication scheme. A coordinator collects the estimates of all processes and proposes a value. All processes then acknowledge this proposal to the coordinator or refuse it if the coordinator is suspected. If the proposal is accepted, the coordinator reliably broadcasts the decision to all processes.

If neither failures nor suspicions occur, this algorithm requires $2n$ messages and one reliable broadcast to reach a decision. The decision is received after 3 communication steps by all processes (2 in the case of the coordinator).

2) Mostéfaoui-Raynal consensus [28]: The Mostéfaoui-Raynal algorithm solves consensus using a decentralized communication scheme. Again, a coordinator collects the estimates of all processes and proposes a value. This time, all processes retransmit this proposal to all other processes or send an invalid value ($\perp$) if the coordinator is suspected. Any process that receives a majority of acknowledgments decides and informs the other processes of its decision.

If neither failures nor suspicions occur, this algorithm requires $2n^2$ messages to reach a decision. The decision is received after 2 communication steps by all processes (or a single step in the case of non-coordinator processes if $n = 3$).

As in [35], all the algorithms are optimized for runs without failures and without suspicions, to minimize the latency when the load on the system is low (rather than minimizing the number of sent messages) and to tolerate high loads. For example, both consensus algorithms send the proposal of a new consensus at the same time as the decision of the previous one (to reduce the amortized latency, see also [20]). Different optimizations (choosing a different reliable broadcast protocol based on a failure detector to reduce the number of sent messages) could of course influence the performance results.

B. Elements of our performance study

The following paragraphs describe the benchmarks (*i.e.*, the performance metrics, the workloads and the faultloads) that were used to evaluate the performance of the three implementations of atomic broadcast (two atomic broadcast algorithms, one of which uses two different consensus algorithms). Similar benchmarks have been presented in [32], [35]. The three algorithms that are compared are noted *TokenFD* (the token based algorithm presented in Section V), *CT* (Chandra-Toueg's atomic broadcast with Chandra-Toueg's $\diamond\mathcal{S}$ consensus) and *MR* (Chandra-Toueg's atomic broadcast with Mostéfaoui-Raynal's $\diamond\mathcal{S}$ consensus). The algorithms are implemented in Java, using the Neko [33] framework.

1) Performance metrics and workloads: The performance metric that is used to evaluate the algorithms is the latency of atomic broadcast. For a single atomic broadcast, the latency L is defined as follows. Let t_a be the time at which the $abroadcast(m)$ event occurred and let t_i be the time at which $adeliver(m)$ occurred on process p_i, with $i \in 0, \ldots, n-1$. The latency L is then defined as $L \overset{def}{=} (\frac{1}{n}\sum_{i=0}^{n-1} t_i) - t_a$. In our performance evaluation, the mean for L is computed over many messages and for several executions. 95% confidence intervals are shown for all the results.

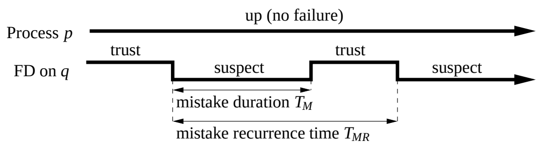


Fig. 2. Quality of service model of a failure detector in the *suspicion-steady* faultload. Process q monitors process p.

The latency L is measured for a certain workload, which specifies how the *abroadcast* events are generated. We chose a simple symmetric workload where all processes send atomic broadcast messages (without any payload) at the same constant rate and the *abroadcast* events come from a Poisson stochastic process. The global rate of atomic broadcasts is called the *throughput* T, which is expressed in messages per second (or $msgs/s$).

Furthermore, we only consider the system in a stationary state, when the rate of *abroadcast* messages is equal to the rate of *adelivered* messages. The clocks of the processes are synchronized to sub-1 ms precision at the beginning of each run of an experiment.

2) Faultloads: The faultload specifies the events related to process failures that occur during the performance evaluation [23], [32]. In our experiments, the faultload focuses on the process crashes and the behavior of the unreliable failure detectors. We evaluate the atomic broadcast algorithms in the *normal-steady* and *suspicion-steady* faultloads [32] which are presented below.

a) Normal-steady: In the *normal-steady* faultload, only runs without process failures and without wrong suspicions are considered. The parameters that influence the latency are n (the number of processes), the algorithm (*TokenFD*, *CT* or *MR*) and the throughput.

b) Suspicion-steady: In the *suspicion-steady* faultload, no processes fail, but wrong suspicions occur. This faultload is implemented by using simulated failure detectors, whose quality of service is modeled as in [11].

The two quality of service metrics presented in [11] that apply to the (failure free) *suspicion-steady* faultload are presented in Figure 2 and detailed below:

- The *mistake recurrence time* T_{MR} is the time between two consecutive mistakes (the failure detector module on process q wrongly suspects process p).
- The *mistake duration* T_M is the time needed to correct the mistake of the failure detector (the time needed for q to trust p again).

To keep the model as simple as possible, we consider that the two random variables T_M and T_{MR} associated with each failure detector are independent and identically distributed and follow an exponential distribution with a (different) constant parameter.

Finally, this simulated failure detector model does not put any load on the network, since no messages are exchanged between the failure detector modules. However, since in a real failure detector implementation, a good quality of service can often be achieved without sending messages frequently, this trade-off is acceptable.

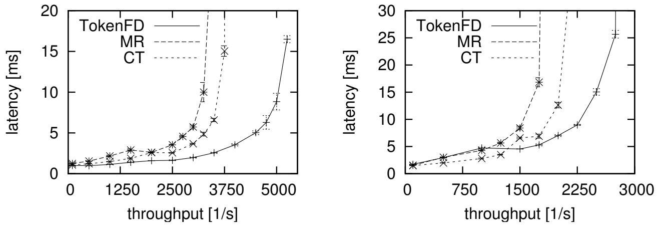
TABLE I
MAXIMUM THROUGHPUT IN A *normal-steady* FAULTLOAD

(a) $n = 3$: one supported failure

Algorithm	*Throughput*
TokenFD	5250 $msgs/s$
CT	4000 $msgs/s$
MR	3500 $msgs/s$

(b) Two supported failures

Algorithm	*Throughput*
TokenFD ($n = 7$)	3000 $msgs/s$
CT ($n = 5$)	2125 $msgs/s$
MR ($n = 5$)	1875 $msgs/s$



(a) $n = 3$: one supported failure (b) $n = 5$ (*CT*, *MR*), $n = 7$ (*TokenFD*): two supported failures

Fig. 3. Latency vs. throughput with a *normal-steady* faultload

C. Related work

Most performance evaluations of the algorithms mentioned above consider a local area network or a simulated model in which all processes and network links are symmetrical [13], [19], [21], [31], [32], [34], [35]. All processes have the same processing power and have identical peer-to-peer round-trip times. Furthermore, the evaluations only consider low round-trip times between processes (and thus comparatively high message processing costs): a setting which is favorable to algorithms that limit the number of sent messages, at the expense of additional communication steps.

The performance of atomic broadcast algorithms in wide area networks has been evaluated in hybrid models that combine a local area network with emulated network delays [36]. The effect of message loss [4] and the performance of other distributed algorithms [5] (that are however not representative of failure detector based algorithms) have also been evaluated in wide area networks.

VII. RESULTS IN A LOCAL AREA NETWORK

A. Evaluation environment

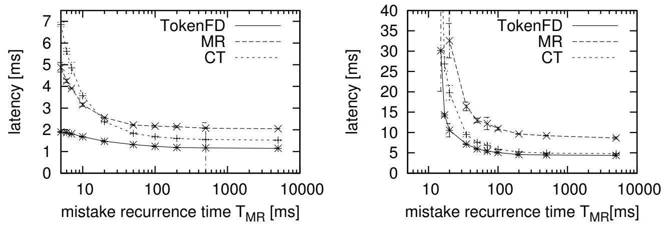
The experiments were executed on a local area network of nodes with Pentium 4 processors at 3 GHz, with 2 MB of L2 cache and 1 GB of RAM. The nodes are interconnected by a single Gigabit Ethernet switch. The round-trip time between two nodes is approximately 0.1 ms. All nodes run Linux (with a 2.6.11 kernel) and Sun's Java 1.5.0_05 virtual machine.

B. Normal-steady faultload

The performance of the three algorithms in a system without failures nor suspicions is presented in Figure 3. The horizontal axis shows the throughput (in messages per second) that is applied, whereas the latency of the algorithms (in ms) for a given throughput is shown vertically. Table I summarizes the maximum throughput reached by the three algorithms (while remaining in a stationary state) in the experiments.

In a system with three processes (Figure 3(a)), in which all three algorithms support one failure, *CT* achieves slightly lower latencies than *MR*, while *TokenFD* reaches the highest throughput and lowest latencies of the three algorithms. *CT* and *MR* need to send more messages than *TokenFD* in this setting: to $abroadcast$ m both algorithms reliably broadcast m (with a cost of 6 messages), then propose and acknowledge m (4 messages for CT, 6 for MR). The decision of both algorithms is propagated with the proposal of the next consensus and no separate message is needed for this. CT thus sends 10 messages to adeliver m, whereas MR sends 12 messages. TokenFD broadcasts m (2 messages), sends 2 messages for the token circulation and broadcasts the decision (2 messages), for a total of 6 messages. The additional messages sent by *CT* and *MR* add a load on the network and the CPUs that explains the difference in performance with respect to *TokenFD*.

The situation is similar in a system where two failures are tolerated (Figure 3(b)), except that *TokenFD* has a higher latency than *CT* and *MR* when the throughput is low. The explanation is the following: in *TokenFD*, the number of communication steps needed to $adeliver$ a message is equal to $f+2$ (with f the tolerated failures) and thus, as f increases, the latency of $adeliver$ also increases. In *MR* and *CT* however, the number of communication steps does not depend on f and the latency of the algorithms is less affected by the increase of the system size.



(a) $n = 3$: one supported failure (b) $n = 5$ (*CT*, *MR*), $n = 7$ (*TokenFD*): two supported failures

Fig. 4. Latency vs. mistake recurrence time T_{MR} with a *suspicion-steady* faultload, a throughput of 1500 $msgs/s$ and a mistake duration $T_M = 5ms$.

C. Suspicion-steady faultload

The performance of the *TokenFD*, *CT* and *MR* algorithms in a system with wrong suspicions (but without process failures) is discussed in the following paragraphs. We consider the case where the frequency of these wrong suspicions varies (but the duration of a wrong suspicion is fixed). In [17], several other parameters are examined for the suspicion-steady faultload. The parameters presented here (throughput of 1500 $msgs/s$ and a duration of wrong suspicions of 5 ms) were chosen because they are representative of the entire parameter space.

Figure 4 illustrates the performance of the three algorithms in systems supporting one or two failures. We present the latency of the three algorithms for a throughput of 1500 $msgs/s$ and wrong suspicions that last on average $T_M = 5ms$. As mentioned above, different throughputs and durations of wrong suspicions are analyzed in [17]. The horizontal axis of each graph represents the recurrence time of wrong suspicions (wrong suspicions occur frequently on the left side of the graph and rarely on the right side) and the vertical axis again represents the latency of atomic broadcast.

In the case of a system with three processes supporting one failure (Figure 4(a)), the *TokenFD* algorithm achieves lower latencies than *CT* and *MR*, both in the case of rare wrong suspicions, shown on the right hand side of the graph (as T_{MR}, the mistake recurrence time grows, the *suspicion-steady* faultload approaches the *normal-steady* faultload presented previously) and when wrong suspicions occur extremely frequently (shown on the left hand side of the graph). *TokenFD* achieves lower latencies than *CT* and *MR* in a system with three processes for two reasons: first of all, *TokenFD* can order messages as soon as there exists one process that is not suspected by its successor, whereas in *CT* and *MR*, a *single* process that suspects the coordinator can delay a consensus decision. Secondly, a wrong suspicion is more costly in *CT* and *MR*: if consensus cannot be reached in a given round, the consensus algorithm starts a new round and needs to send at least an additional $4n = 12$ or $n + n^2 = 12$ messages in 4 and 2 additional communication steps respectively.

In the case of *TokenFD*, a wrong suspicion incurs a cost of at least an additional $f + 1 = 2$ messages and one communication step. The cost of a wrong suspicion also explains why the latency of *MR* is lower than that of *CT* when suspicions are frequent, whereas *CT* outperforms *MR* when (almost) no wrong suspicions occur.

When a system that supports two failures is considered (Figure 4(b)), the results are slightly different. Indeed, when the interval between wrong suspicions is low enough — around 15 ms — the algorithms cannot $adeliver$ messages at the offered load and the latency increases sharply. In the case of *CT* and *MR* in a system with $n = 5$ processes, 4 processes can potentially suspect the coordinator and send a *nack* (*CT*) or $\perp$ (*MR*) that can prevent a decision in the current round. Only 2 processes could suspect the coordinator in the case of $n = 3$. The increased fault tolerance also affects *TokenFD*: indeed, in a system supporting two failures (*i.e.*, $n = 7$), a batch of messages can only be ordered if two consecutive processes do not suspect their respective predecessors.

D. Summary

In a system with $n = 3$ processes without crashes nor wrong suspicions, the latency of atomic broadcast is lower when using *TokenFD* than *CT* or *MR*. Furthermore, *TokenFD* allows a higher rate of $abroadcasts$ while maintaining the system in a stationary state. When two failures are tolerated (requiring 5 processes with *CT* and *MR*, 7 processes with *TokenFD*), *CT* and *MR* achieve lower latencies than *TokenFD* when the system load is low. As soon as the load reaches about 1500 $msgs/s$, *TokenFD* again outperforms both other algorithms. These results confirm that performance-wise *TokenFD* behaves better than other failure detector based algorithms in small systems. Furthermore, the latency of *TokenFD* also remains low with respect to the two other algorithms, when the load on the system increases in good runs.

Concerning the second desired property — handling wrong suspicions well — the results above show that in small systems, the performance of the *TokenFD* algorithm is better than both *CT* (which is shown to handle wrong suspicions better than a group membership based algorithm in [35]) and *MR* when the interval between wrong suspicions is short. These results confirm that the second desired property of the *TokenFD* algorithm holds: wrong failure suspicions do not drastically reduce the performance of the algorithm. This in turn allows an implementation of the failure detector with aggressive timeouts, which consequently allows actual failures to be detected fast. If the failure detector implementation commits a mistake and wrongly suspects a process that is still alive, this mistake does not cost much in terms of performance.

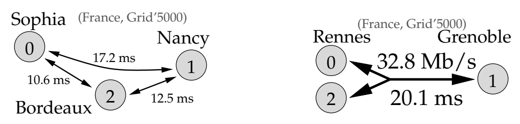


Fig. 5. WAN evaluation environments (WAN Three Locations, WAN 20.1)

VIII. Results in a Wide Area Network

The performance of the algorithms that we consider is affected by a trade-off between the number of communication steps and the number of messages needed to reach a decision. Some algorithms reach decisions in few communication steps but require more messages to do so. Others save messages at the expense of additional communication steps (to diffuse the decision to all processes in the system for example). This trade-off is heavily influenced by the message transmission and processing times. When deploying an atomic broadcast algorithm, the user must take these factors into account in order to choose the algorithm that is best adapted to the given network environment.

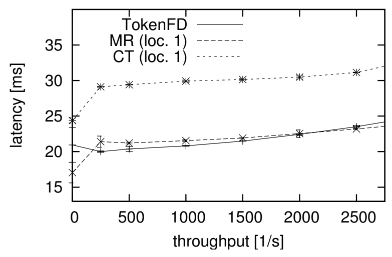
Furthermore, evaluating the performance of atomic broadcast on wide area networks is not only of theoretical interest. As [25] shows, it is feasible to use atomic broadcast as a service to provide consistent data replication on wide area networks. We initially focus on the case of a system with three processes — *i.e.*, supporting one failure — where either (i) all three processes are on different locations and (ii) the three processes are on two locations only (and thus one of the locations hosts two processes). The system with three processes is interesting as it has no single point of failure and represents the case in which the group communication algorithms reach their best performance. Furthermore, atomic broadcast provides strong consistency guarantees (that can be used to implement active replication for example [30]) and is limited to relatively small degrees of replication. Google, for example, uses the Paxos consensus algorithm for Chubby, its distributed lock service, in systems with $n = 5$ processes (supporting up to $f = 2$ process failures) [8]. If a large degree of replication is needed, then alternatives that provide weaker consistency should be considered [2].

A. Evaluation environments

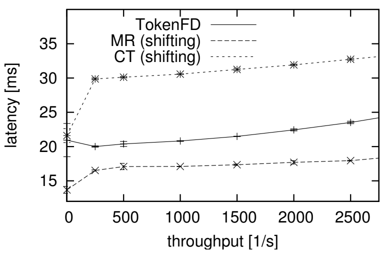
The algorithms are evaluated with the *normal-steady* faultload and with a large variation in link latency (*e.g.*, round-trip times ranging from 4 to 300 ms).

Four wide area network environments are used to evaluate the performance of the three atomic broadcast and consensus algorithms (Figure 5 presents two of the environments). All machines run Linux (2.6.8 to 2.6.12 kernels) and Sun's Java 1.5.0 virtual machine. The following paragraphs describe the different wide area network environments in which the atomic broadcast algorithms are evaluated.

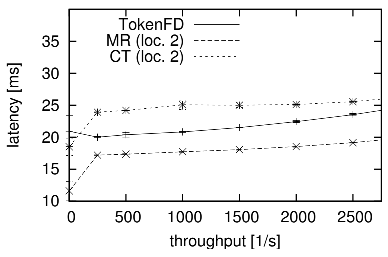
1) Three-location wide area network: The first evaluation environment (noted WAN Three Locations, Figure 5, left) is a system with three locations on Grid'5000 [7], a French grid of interconnected clusters designed for the experimental evaluation of distributed systems. The round-trip times of the links between the three processes are respectively 17.2 ms, 12.5 ms and 10.6 ms.



(a) Initial coordinator on location 1.



(b) Shifting initial coordinator



(c) Initial coordinator on location 2

Fig. 6. Average latency of the three algorithms as a function of the throughput in the WAN Three Locations setting.

The observed bandwidth of the links are 30.1 Mbits/s, 41.4 Mbits/s and 48.7 Mbits/s.

2) Two-location wide area networks: Three processes are distributed on two different locations in a wide area network. The location with two processes is denoted the *local* location, whereas the location with a single process is the *distant* location. Three different two-location environments are considered:

– WAN 295: The first two-location environment consists of a local location in Switzerland and a distant one in Japan. The round-trip time between the locations is 295 ms and the bandwidth of the connecting link is 1.74 Mb/s.

– WAN 20.1 and WAN 3.9: The two following environments are systems with both locations on Grid'5000. The WAN 20.1 system (Figure 5, right) features a round-trip time between locations of 20.1 ms and a link bandwidth of 32.8 Mb/s. The WAN 3.9 system features a round-trip time between locations of 3.9 ms and a link bandwidth of 152 Mb/s. The performance characteristics of the three algorithms are similar in WAN 20.1 and WAN 3.9.

Due to a lack of space, we present the results in the WAN Three Locations and WAN 20.1 settings here and refer the reader to [18] for the WAN 295 and WAN 3.9 settings.

B. The issue of the initial coordinator location

Both *CT* and *MR* are coordinator based algorithms: a coordinator process proposes the value that is later decided upon. Upon starting a consensus execution in one of these algorithms, one process is selected deterministically as coordinator (this is the *initial* coordinator). Later on in the consensus execution, the coordinator process may change in case of suspicions.

The choice of the initial coordinator process in a local area network is not an issue, since all processes are symmetrical (same hardware and same network latency between processes). In a wide area network, however, the performance of the *CT* and *MR* algorithms heavily depends on the choice of the initial coordinator. In the following performance evaluation, we therefore consider runs with an initial coordinator on each one of the wide area network locations (this initial coordinator is fixed for all consensus executions), as well as runs where the initial coordinator, for each consensus execution, is on a different location (*i.e.*, a *shifting* initial coordinator).

C. Comparing the performance of the three algorithms

1) WAN Three Locations*:* The average latency of the three algorithms in the WAN Three Locations environment is presented in Figure 6. *TokenFD* and *MR* outperform *CT* for all locations of the initial coordinator and for all throughputs, due to the additional communication step that is needed by the *CT* algorithm. *TokenFD* and *MR* perform similarly when the initial *MR* coordinator is on site 1 (which is the worst-case scenario for *MR*), whereas *MR* achieves slightly better latencies than *TokenFD* for both other initial coordinator locations.

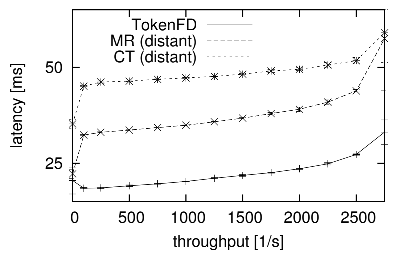
Surprisingly enough, the result of using a shifting initial coordinator in the *CT* and *MR* algorithms are opposite: in the case of *MR*, the latency is lower using a shifting initial coordinator than a fixed initial coordinator on any location, whereas in *CT* it is higher. The explanation is the following: *MR* and *CT* both start a new consensus execution after two communication steps if the coordinator is on a fixed location. If the coordinator shifts, a new execution can start as soon as the next non-coordinator process decides. This is done after *one* communication step in *MR* (if $n = 3$), but after *three* steps in *CT*.

2) WAN 295*,* WAN 20.1 *and* WAN 3.9*:* The average latency of the three atomic broadcast and consensus algorithms in the WAN 20.1 environment is presented in Figure 7 (the WAN 295 and WAN 3.9 environements are presented in [18]). *TokenFD* has lower latencies than *CT* and *MR* when they use a distant initial coordinator (Figure 7(a)), whereas the situation is reversed when the coordinator is initially on a local location (Figure 7(c)). When the initial coordinator shifts at each new consensus execution, *MR* and *TokenFD* have similar latencies while *CT* is slightly slower.

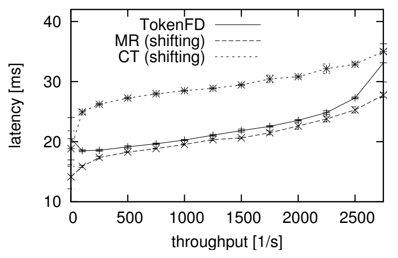
Finally, when the *CT* or *MR* algorithm is used with an initial coordinator on the local location, the system never reaches a stationary state given a sufficiently high throughput (2000 $msgs/s$ in Figure 7(c)). The processes on the local location reach consensus decisions very fast without needing any input from the distant location. The updates that are then sent to the distant location saturate the link between both locations (its bandwidth is only 32.8 Mbits/s in WAN 20.1). The process on the distant location thus takes decisions slower than the two local processes and prevents the average latency of atomic broadcast from stabilizing. This problem does not affect the settings with a distant or shifting initial coordinator, since the distant location periodically acts as a consensus coordinator, providing a natural flow control. We see that setup issues, such as the choice of the initial coordinator, affect the maximum achievable throughput of the algorithms.

D. Summary

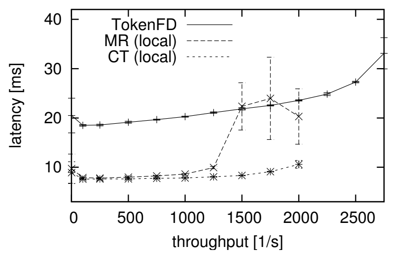
As expected, the performance results presented above show that *communication steps have the largest impact on performance in wide area networks, whereas the number of sent messages is a key to the performance in a local area network* (as illustrated in Section VII). As the network latency decreases, the impact of the



(a) Distant coordinator



(b) Shifting initial coordinator



(c) Local coordinator

Fig. 7. Average latency of the three algorithms as a function of the throughput in the WAN 20.1 setting.

additional messages that need to be sent and processed increases. In the case of a network with a 20.1 ms (or 3.9 ms [18]) round-trip time, this impact is clearly observable. However, for a given set of parameters, the algorithm with the best performance is generally the same (whether a wide area network with a 3.9 ms round-trip time is considered or one with a 20.1 ms or 295 ms round-trip time).

Finally, we also saw that choosing a *CT* and *MR* coordinator on the local location yields low latencies, but is not necessarily the best solution in terms of throughput, since the system cannot reach a stationary state as the total throughput increases. An additional ad-hoc flow control mechanism is needed. Shifting the initial coordinator between locations at each new consensus execution or choosing the *TokenFD* algorithm results in a natural flow control which enables the system to remain in a stationary state even for high throughputs (at the expense of a higher *adelivery* latency).

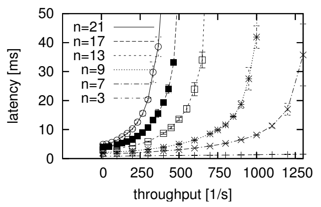
IX. Performance of the algorithms in large systems

The experimental performance evaluation of the three atomic broadcast algorithms in Sections VII and VIII was limited to relatively small systems and the algorithms could be criticized for not scaling well as the number of processes in the system increases. This scalability problem is due to different factors for each of the considered algorithms. The *MR* algorithm, for example, is affected by the $O(n^2)$ messages that need to be transmitted and processed to solve consensus. In the case of *CT*, a fan-in problem arises: one process needs to receive and handle replies from all other processes in order to solve consensus and thus becomes a bottleneck for the performance of the system. Finally, in the *TokenFD* algorithm described in Section V, the number of communication steps needed to solve atomic broadcast increases with the size of the system. We evaluate the algorithms in systems with up to 23 processes. In larger systems, fault tolerance is less of an issue and group communication services with weaker consistency guarantees might be preferred.

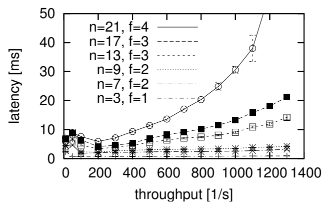
A. Evaluation environments

The performance of the algorithms is measured in a large local area network and then in a wide area networks of interconnected clusters. The characteristics of these environments are the following.

1) Local area network: The first set of experiments were executed on the local area network cluster previously presented in Section VII.



(a) *CT* ($f = \lfloor \frac{n}{2} \rfloor$)



(b) *TokenFD*

Fig. 8. Latency vs. throughput of the *CT* and *TokenFD* algorithms in a local area network with 3 to 21 processes.

2) Wide area network – Grid'5000: The second set of experiments were executed on Grid'5000, which was mentioned in Section VIII. Grid'5000 is composed of 14 sites (in 9 geographical locations), among which seven were used for our measurements.

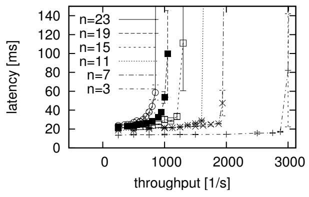
All the sites (Bordeaux, Rennes, Toulouse, Lyon, Nancy, Orsay, Sophia) feature nodes with AMD Opteron 246 or 248 dual processors at 2 GHz or 2.2 GHz respectively, with 1MB of L2 cache and 2GB of RAM. The machines on the different sites all run Linux (with a 2.6.8 or a 2.6.12 kernel). The nodes on a given site are interconnected by Gigabit Ethernet and run Sun's Java 1.5.0_06 AMD64 Server Virtual Machine. The geographical location and round trip times between sites are presented in [17].

The coordinator process of the *CT* consensus algorithms always runs on a node of the Orsay site. The *TokenFD* passes the token among sites in the following order: Orsay – Bordeaux – Nancy – Lyon – Toulouse – Sophia – Rennes.

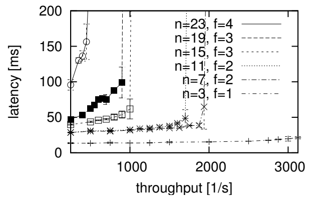
B. Comparing the performance of the three algorithms

1) Local Area Network: Figure 8 shows the latency versus the throughput of the Chandra-Toueg (Figure 8(a)) and *TokenFD* (Figure 8(b)) atomic broadcast algorithms when all processes in the system participate in the algorithms. The figure shows results for system sizes ranging from 3 to 21 processes, supporting from 1 to 10 failures (*CT*) or 1 to 4 failures (*TokenFD*). The horizontal axis shows the load on the system (in messages per seconds), and the average latency (ms) is shown on the vertical axis.

In the case of a small system with 3 processes, the latency remains almost constant as the throughput increases, for all three algorithms. In the largest system with 21 processes, however, the latency increases extremely fast with the throughput, especially for the *CT* algorithm, where $O(n^2)$ messages are needed to solve atomic broadcast. In the case of *TokenFD*, the scalability problem is less severe (since only $O(n)$ messages are sent by the algorithm). The fault tolerance is however lower and the latency



(a) *CT* ($f = \lfloor \frac{n}{2} \rfloor$)



(b) *TokenFD*

Fig. 9. Latency vs. throughput of the *CT* and *TokenFD* algorithms in the Grid'5000 wide area network with 3 to 23 processes.

still increases with the system size ($O(\sqrt{n})$ communication steps are needed to $adeliver$ messages).

2) Wide area network – Grid'5000: We now evaluate the performance of the algorithms in a wide area network, to examine how high latency channels affect the performance of the atomic broadcast algorithms.

Figure 9 shows the latency versus the throughput of the three atomic broadcast algorithms running on all processes, on the Grid'5000 wide area network. The figure shows results for system sizes ranging from three to 23 processes, supporting from 1 to 11 failures (*CT*) or 1 to 4 failures (*TokenFD*).

For a given system size and a given algorithm, the latency of the algorithms remains relatively stable until a threshold is reached. Above that threshold, the latency quickly increases. This is especially the case for the smaller systems ($n = 3$ to $n = 15$).

Furthermore, in the wide area network, the additional communication steps (which are costly) needed by the *TokenFD* algorithm as the system size increases strongly affect the performance of the algorithm. Figure 9(b) shows that if a single failure is supported ($n = 3$), then the maximum throughput of the algorithm is above 3250 msgs/s. This threshold drops to approximately 2000 msgs/s when $n = 7$ and $f = 2$, and does not exceed 500 msgs/s when $n = 23$ and $f = 4$. Each additional supported failure requires an additional 12 ms communication step.

C. Summary

In the local area network, the performance of *CT* depends strongly on the number of messages that are sent. As the size of the system increases, its performance drops due to the $O(n^2)$ messages that are sent. The *TokenFD* algorithm is less affected by the size of the system, since the algorithm only sends $O(n)$ messages. The $\sqrt{n}$ communication steps needed by *TokenFD* to $adeliver$ a messages only have a minor effect on the performance of the algorithm in the local area network.

In the wide area network, on the other hand, the performance *TokenFD* is strongly affected by the additional (expensive) communication steps between sites. The performance of *CT* shows similar trends in the wide and local area networks. The *CT* algorithm needs a fixed number of communication steps to $adeliver$ a message and is thus not more affected by a large wide area system than by a large local area system.

X. Conclusion

According to various authors, token based atomic broadcast algorithms are more efficient in terms of throughput than other atomic broadcast algorithms; the token can be used to reduce network contention. However, all published token based algorithms rely on a group membership service; none of them use unreliable failure detectors directly. The first part of this paper presented the first token based atomic broadcast algorithms that solely relies on a failure detector, namely the new failure detector called $\mathcal{R}$. Such an algorithm has the advantage of tolerating failures *directly* (*i.e.*, it also tolerates wrong failure suspicions), instead of relying on a membership service to exclude crashed processes (which, as a side-effect, also excludes incorrectly suspected processes). Thus, failure detector based algorithms have advantages over group membership based algorithms, in case of wrong failure suspicions, and possibly also in the case of real crashes.

The local area network performance evaluation in the second part of this paper showed that the token based algorithm *TokenFD* surpasses the Chandra-Toueg atomic broadcast algorithm (using the Chandra-Toueg or Mostéfaoui-Raynal consensus algorithm) for systems that support up to two process failures, both in runs without faulty processes and in the case of wrong suspicions. However, *TokenFD* requires a system size n that is quadratic in the number of failures f. In systems that need to handle a high fault-tolerance degree (*i.e.*, when f becomes large), *TokenFD*'s relative performance degrades compared to *CT* and *MR*.

Furthermore, although token based atomic broadcast algorithms are usually considered to be efficient only in terms of throughput, our experimental performance evaluation showed that for small values of n, our algorithm outperforms the two other algorithms in terms of latency as well, at all but the lowest loads.

Finally, this paper also presented the performance of the token based algorithm in wide area networks and systems with a large number of processes. This evaluation showed that the token based algorithm provides a natural flow control in wide area networks, which both other algorithms do not. In large wide area networks, however, the latency of the token based algorithm is affected by the additional costly communication steps that are needed.

References

[1] Token Passing Bus Access Method, ANSI/IEEE standard 802.4. 1985.

[2] L. Alvisi and K. Marzullo. Waft: Support for fault-tolerance in wide-area object oriented systems. In *Proc. 2nd Information Survivability Workshop – ISW '98*, pages 5–10, Los Alamitos, CA, USA, October 1998. IEEE Computer Society Press.

[3] Y. Amir, L. Moser, P. Melliar-Smith, D. Agarwal, and P.Ciarfella. The Totem Single-Ring Ordering and Membership Protocol. *ACM Trans. on Computer Systems*, 13(4):311–342, November 1995.

[4] T. Anker, D. Dolev, G. Greenman, and I. Shnayderman. Evaluating total order algorithms in WAN. In *Proc. International Workshop on Large-Scale Group Communication*, Florence, Italy, October 2003.

[5] O. Bakr and I. Keidar. Evaluating the running time of a communication round over the internet. In *PODC '02: Proc. twenty-first annual symposium on Principles of distributed computing*, pages 243–252, Monterey, California, USA, 2002. ACM Press.

[6] K. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Trans. on Computer Systems*, 9(3):272–314, August 1991.

[7] F. Cappello, E. Caron, M. Dayde, F. Desprez, E. Jeannot, Y. Jegou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, and O. Richard. Grid'5000: a large scale, reconfigurable, controlable and monitorable Grid platform. In *Grid'2005 Workshop*, Seattle, USA, November 13-14 2005. IEEE/ACM.

[8] T. Chandra, R. Griesemer, and J. Redstone. Paxos made live: An engineering perspective. In *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407, New York, NY, USA, 2007. ACM.

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

[10] J. M. Chang and N. Maxemchuck. Reliable Broadcast Protocols. *ACM Trans. on Computer Systems*, 2(3):251–273, August 1984.

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

[12] G. V. Chockler, I. Keidar, and R. Vitenberg. Group Communication Specifications: A Comprehensive Study. *ACM Computing Surveys*, 4(33):1–43, December 2001.

[13] 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 Conf. on Dependable Systems and Networks (DSN 2002)*, pages 551–560, Washington, DC, USA, June 2002.

[14] F. Cristian. Reaching Agreement on Processor Group Membership in Synchronous Distributed Systems. *Distributed Computing*, 4(4):175–187, April 1991.

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

[16] X. Défago, A. Schiper, and P. Urbán. Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey. *ACM Computing Surveys*, 36(2):372–421, December 2004.

[17] R. Ekwall. *Atomic Broadcast: a Fault-Tolerant Token based Algorithm and Performance Evaluations*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, May 2007. Number 3811.

[18] R. Ekwall and A. Schiper. Comparing Atomic Broadcast Algorithms in High Latency Networks. Technical Report LSR-Report-2006-003, École Polytechnique Fédérale de Lausanne, Switzerland, July 2006.

[19] R. Ekwall, A. Schiper, and P. Urbán. Token-based Atomic Broadcast using Unreliable Failure Detectors. In *Proceedings of 23rd IEEE Symposium on Reliable Distributed Systems (SRDS-23)*, Florianopolis, Brazil, Oct. 2004.

[20] R. Friedman and R. van Renesse. Packing messages as a tool for boosting the performance of total ordering protocols. In *6th IEEE Symposium on High Performance Distributed Computing*, pages 233–242, Portland, OR, USA, Aug. 1997.

[21] R. Guerraoui, R. R. Levy, B. Pochon, and V. Quéma. High Throughput Total Order Broadcast for Cluster Environments. In *IEEE International Conference on Dependable Systems and Networks (DSN 2006)*, Philadelphia, PA, USA, June 2006.

[22] V. Hadzilacos and S. Toueg. Fault-Tolerant Broadcasts and Related Problems. Technical Report 94-1425, Department of Computer Science, Cornell University, May 1994.

[23] K. Kanoun, H. Madeira, and J. Arlat. A Framework for Dependability Benchmarking. In *Workshop on Dependability Benchmarking (jointly organized with DSN-2002)*, pages F7–F8, Bethesda, Maryland, USA, June 2002.

[24] M. Larrea, S. Arevalo, and A. Fernandez. Efficient algorithms to implement unreliable failure detectors in partially synchronous systems. In *Int. Symposium on Distributed Computing*, pages 34–48, 1999.

[25] Y. Lin, B. Kemme, M. Patiño-Martínez, and R. Jiménez-Peris. Consistent data replication: Is it feasible in WANs?. In *Proc. 11th Euro-Par Conference*, pages 633–643, Lisbon, Portugal, September 2005.

[26] N. F. Maxemchuk and D. H. Shur. An Internet multicast system for the stock market. *ACM Trans. on Computer Systems*, 19(3):384–412, August 2001.

[27] S. Mena, A. Schiper, and P. Wojciechowski. A Step Towards a New Generation of Group Communication Systems. In *Proc. Int. Middleware Conference*, pages 414–432. Springer Verlag, LNCS 2672, June 2003.

[28] A. Mostefaoui and M. Raynal. Solving Consensus using Chandra-Toueg's Unreliable Failure Detectors: A Synthetic Approach. In *13th. Intl. Symposium on Distributed Computing (DISC'99)*. Springer Verlag, LNCS 1693, September 1999.

[29] A. Schiper and S. Toueg. From Set Membership to Group Membership: A Separation of Concerns. *IEEE Transactions on Dependable and Secure Computing*, 3(1):2–12, 2006.

[30] F. B. Schneider. Replication Management using the State-Machine Approach. In S. Mullender, editor, *Distributed Systems*, pages 169–197. ACM Press, 1993.

[31] A. Sousa, J. Pereira, F. Moura, and R. Oliveira. Optimistic Total Order in Wide Area Networks. In *21st IEEE Symp. on Reliable Distributed Systems (SRDS-21)*, pages 190–199, Osaka, Japan, October 2002.

[32] 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.

[33] 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.

[34] P. Urbán, N. Hayashibara, A. Schiper, and T. Katayama. Performance comparison of a rotating coordinator and a leader based consensus algorithm. In *Proc. 23rd IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 4–17, Florianópolis, Brazil, October 2004.

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

[36] P. Vicente and L. Rodrigues. An Indulgent Total Order Algorithm with Optimistic Delivery. In *21st IEEE Symp. on Reliable Distributed Systems (SRDS-21)*, pages 92–101, Osaka, Japan, October 2002.

[37] B. Whetten, T. Montgomery, and S. M. Kaplan. A high performance totally ordered multicast protocol. In *Dagstuhl Seminar on Distributed Systems*, pages 33–57, 1994.

APPENDIX

PROOF OF LEMMA 3.1

From the definition it follows directly that $\Diamond\mathcal{P}$ is stronger or equivalent to $\mathcal{R}$, denoted by $\Diamond\mathcal{P} \succeq \mathcal{R}$. We prove that $\Diamond\mathcal{P} \succ \mathcal{R}$.

We prove that $\Diamond\mathcal{P} \succ \mathcal{R}$ in a system with $f > 1$ by contraction. We denote by $\mathcal{D}_i^t$ the output of $\mathcal{R}$ at time t on p_i (which gives information about p_{i-1}). $\mathcal{D}_i^t$ is the set of processes that p_i suspects at time t and can either be the empty set $\emptyset$ or $\{p_{i-1}\}$. From the definition of $\mathcal{R}$, $\forall j \neq i : \forall t : p_{i-1} \notin \mathcal{D}_j^t$

For a contradiction assume $\Diamond\mathcal{P} \equiv \mathcal{R}$ in a system with $f > 1$. Let algorithm A allow us to construct $\Diamond\mathcal{P}$ from $\mathcal{R}$. Also, assume that process p_i is faulty. With p_i faulty, there is a time t_0 after which the output $\mathcal{D}_i^t$ of $\mathcal{R}$ on p_i does not change (whether p_{i-1} is correct or faulty, or crashes before or after t_0): $\forall t > t_0 : \mathcal{D}_i^t = \mathcal{D}_i^{t_0}$.

Consider first a run R_0 of A in which p_{i-1} is correct. By accuracy, let t_{ns} be the time at which p_{i-1} is no more suspected by correct processes. We define $t_1 = \max(t_0, t_{ns})$.

Then, construct run R_1 of A as follows. Run R_1 is identical to R_0 except that p_{i-1} crashes at time t_1 (with $f > 1$, there can be two faulty processes in the system). For all processes $\Pi - \{p_{i-1}\}$, run R_1 is indistinguishable from run R_0 up to t_1. Therefore, no correct process suspects p_{i-1} at time t_1. By completeness, in run R_1, there must exist a time $t_2 > t_1$, such that all correct processes suspect p_{i-1} forever after t_2.

Construct run R_2 of A as follows. Run R_2 is identical to R_1 except that p_{i-1} is correct in R_2 and delay all messages sent by p_{i-1} after t_1 such that they are received after t_2. Since $\forall t > t_0 : \mathcal{D}_i^t = \mathcal{D}_i^{t_0}$, run R_2 is indistinguishable from R_1 until t_2. Therefore, p_{i-1} is suspected by all correct processes at t_2.

To summarize, starting from run R_0 of A in which p_{i-1} is correct, we have constructed run R_2 of A in which p_{i-1} is correct but suspected one more time. Repeating the same construction shows that p_{i-1} can be suspected an unbounded number of times. $\square$