Revisiting Token-Based Atomic Broadcast Algorithms

Richard Ekwall and André Schiper
École Polytechnique Fédérale de Lausanne (EPFL)
1015 Lausanne, Switzerland

{nilsrichard.ekwall,andre.schiper}@epfl.ch

Technical Report IC/2003/39

***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, i.e., none of them uses failure detectors. The paper presents the first token-based atomic broadcast algorithm that uses an unreliable failure detector – the new failure detector denoted by $\mathcal{R}$ – instead of a group membership service. The failure detector $\mathcal{R}$ is compared with $\Diamond\mathcal{P}$ and $\Diamond\mathcal{S}$. In order to make it easier to understand the atomic broadcast algorithm, the paper derives the atomic broadcast algorithm from a token-based consensus algorithm that also uses the failure detector $\mathcal{R}$.**

I. Introduction

A. Context

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 [1]. Many atomic broadcast algorithms have been published in the last twenty years. These algorithms can be classified according to the mechanism used for message ordering [2]. *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 exclude synchronous systems and consider only crash failures, the two main mechanisms to tolerate failures in the context of atomic broadcast algorithms are (i) *unreliable failure detectors* [3] and (ii) *group membership* [4]. For example, the atomic broadcast algorithm in [3] (together with a consensus algorithm using the failure detector $\Diamond\mathcal{S}$ [3]) falls into the first category; the atomic broadcast algorithm in [5] falls into the second category.

Research supported by OFES under contract number 01.0537-1 as part of the IST REMUNE project (number 2001-65002).

B. Group membership mechanism $vs.$ failure detector mechanism.

A group membership service provides a consistent membership information to all the members of a group [4]. Its main feature is to *remove* processes that are suspected to have crashed.[1] In contrast, an unreliable failure detector, e.g., $\Diamond\mathcal{S}$, does not provide consistent information about the failure status of processes. For example, it can tell to process p that r has crashed, while telling at the same time to process q that r is alive.

Both mechanisms can make mistakes, e.g., by incorrectly suspecting correct processes. However, the cost of a false failure suspicion is higher when using a group membership service than when using failure detectors. This is because the group membership service removes suspected processes from the group

[1]The comment applies to the so-called *primary-partition* membership [4].

(a costly operation); there is no removal of suspected processes with a failure detector. Moreover, with a group membership service, the removal of a process is usually followed by the addition of another (or the same) process, in order to keep the same replication degree. So, with a group membership service, a false suspicion leads to two costly membership operations: *removal* of a process followed by the *addition* of another process.

In an environment where false failure suspicions are frequent,[2] algorithms based on failure detectors thus have advantages over algorithms based on a group membership service. The cost difference has been experimentally evaluated in [6] in the context of one specific (not token-based) atomic broadcast algorithm.

C. Why token-based algorithms?

According to [7], [8], [9], token-based atomic broadcast algorithms are extremely efficient in terms of throughput, i.e., 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, and/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.[3] 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.

Note that we do not claim that token-based atomic broadcast algorithms are efficient in terms of latency. As just said, token-based algorithms are interesting from another point of view: they are efficient in terms of throughput. For many applications, high throughput may be more important than low latency.

[2]This typically happens if the timeouts used to suspect processes have been set to small values (i.e., in the order of the average message transmission delay), in order to reduce the time needed to detect the crash of processes.

[3]The group membership mechanism does not necessarily appear explicitly in the algorithm, e.g., in [9]. It can be implemented in an ad-hoc way.

D. Contribution of the paper

The paper gives the first token-based atomic broadcast algorithm that uses unreliable failure detectors instead of group membership. This result is obtained in several steps. The paper first gives a new and more general definition for token-based algorithms (Sect. II) and introduces 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 $\mathcal{R}$ is stronger than necessary to solve consensus or atomic broadcast (since $\Diamond\mathcal{S}$ is strong enough), $\mathcal{R}$ is needed for token-based algorithm: $\Diamond\mathcal{S}$ is too weak to be used for token-based algorithms. Moreover, $\mathcal{R}$ 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.

Section IV concentrates on the consensus problem. First we define two classes of token-based algorithms: *token-accumulation* algorithms and *token-coordinated* algorithms. Then we give a token-accumulation consensus algorithm based on the failure detector $\mathcal{R}$.

Atomic broadcast is solved in Section V: a token-accumulation algorithm is presented. The algorithm is inspired from the token-accumulation consensus algorithm of Section IV. Note that a standard solution consists in solving atomic broadcast by reduction to consensus [3]. However, this solution is not adequate here. The atomic broadcast algorithm that we propose is "derived" from the token-based consensus algorithm, but the algorithm cannot be expressed in terms of "reduction" of atomic broadcast to consensus.[4] Actually, we could have presented only the token-based atomic broadcast algorithm. However, the token-based consensus algorithm is simpler than the token-based atomic broadcast algorithm. The detour through the consensus algorithm makes it easier to understand the atomic broadcast algorithm.

Related work is presented in Section VI and Section VII concludes the paper.

[4]The solution obtained by reduction of atomic broadcast to consensus would be very inefficient.

II. System model and definitions

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, from now on, simply 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 (no Byzantine failures). A process that never crashes is said to be *correct*, otherwise it is *faulty*. At most f processes are *faulty*. The system is augmented with unreliable failure detectors [3] (see below).

A. The Consensus problem

As in [3], we specify the (uniform) consensus problem by four 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.

B. The Atomic Broadcast problem

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 Atomic Broadcast by four properties [1]: (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.

C. Token-based algorithm

In a 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 the reformation phase). 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, and (2) each process p_i has a failure detector module FD_i that provides information only about its immediate predecessor p_{i-1}.

D. Failure detectors

We refer below to two failure detectors introduced in [3]: $\Diamond\mathcal{P}$ 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.

III. Failure detector $\mathcal{R}$

For token-based algorithms we define a new failure detector denoted by $\mathcal{R}$ (stands for *Ring*). Given process p_i, the failure detector attached to p_i only gives information about the immediate predecessor p_{i-1}.[5] For every process p_i, $\mathcal{R}$ ensures the following properties:
(i) *Completeness*: If p_{i-1} crashes and p_i is correct, then p_{i-1} is eventually permanently suspected by p_i, and
(ii) *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 relation *weaker/stronger* between failure detectors has been defined in [3]. We show that (a) $\Diamond\mathcal{P}$ is strictly stronger than $\mathcal{R}$ (denoted $\Diamond\mathcal{P} \succ \mathcal{R}$), and (b) $\mathcal{R}$ is strictly stronger than $\Diamond\mathcal{S}$ if $n \geq f(f+1)+1$ ($\mathcal{R} \succ \Diamond\mathcal{S}$).

Lemma 1: $\Diamond\mathcal{P}$ is strictly stronger than $\mathcal{R}$.

Proof: This result is easy to establish. From the definition it follows directly that $\Diamond\mathcal{P}$ is stronger or equivalent to $\mathcal{R}$, denoted by $\Diamond\mathcal{P} \succeq \mathcal{R}$. Moreover, when p_i is faulty, then $\mathcal{R}$ provides no information about p_{i-1}:[6] so $\Diamond\mathcal{P} \not\cong \mathcal{R}$ ($\Diamond\mathcal{P}$ not equivalent to

[5]Remember the meaning of the notation p_{i-k} or p_{i+k} introduced at the beginning of Section II.

[6]In the special case of $f = 1$, the information about p_{i-1} can be obtained indirectly, i.e., if $f = 1$, the relation between $\Diamond\mathcal{P}$ and $\mathcal{R}$ is not strict: $\Diamond\mathcal{P} \succeq \mathcal{R}$.

$\mathcal{R}$). Together with $\Diamond\mathcal{P} \succeq \mathcal{R}$ we have that $\Diamond\mathcal{P} \succ \mathcal{R}$. $\Box$

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:
(i) *Strong Completeness*: Eventually every process that crashes is permanently suspected by every correct process and
(ii) *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: 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 [1]) 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 $f-1$ immediate successors $p_{k+1},\ldots,p_{k+f-1}$ 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 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 $correct(t') \neq \Pi$, 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_{i-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. $\square$

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 $\mathcal{R} \succeq \Diamond\mathcal{S}2$.

IV. Token-based consensus

A. Two classes of algorithms

We identify two classes of token-based consensus algorithms: *token-accumulation* algorithms and *token-coordinated* algorithms. In the *token-accumulation* algorithms, each token holder votes for the proposal transported in the token. Votes are accumulated as the token circulates and once enough votes have been collected, the token holder can decide. In this class of algorithms, the only communication is related to the circulation of the token. This is not the case of *token-coordinated* algorithms. In these algorithms the token holds a proposal, but, in order to decide, the token holder can communicate with all other processes. Algorithms based on the *rotating-coordinator paradigm* (such as the Chandra-Toueg $\Diamond\mathcal{S}$ consensus algorithm [3]) can easily be adapted to this class. Token-accumulation algorithms are more genuine token-based algorithms, and the paper concentrates on this class of algorithms.

B. Token circulation

The token circulation, which can be handled in the same way for token-accumulation and token-coordinated algorithms, is as follows. To avoid the loss of the token due to crashes, process p_i sends the token to its $f+1$ successors in the ring, i.e., to $p_{i+1}, \ldots, p_{i+f+1}$.[7] Furthermore, when awaiting the token, process p_i waits to get the token from p_{i-1}, unless it suspects p_{i-1}. If p_i suspects p_{i-1}, it accepts the token from any of its predecessors (see Procedure 1).

Procedure 1 Receive token (code of process p_i)

```
1: wait until received token from $p_{i-1}$ or suspected($p_{i-1}$)
2: **if** token not received **then**            {*accept from anyone*}
3:   wait until received token from $p \in \{p_{i-f-1}, ..., p_{i-1}\}$
4: **end if**
```

C. Token-accumulation consensus algorithm

1) Basic idea: 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 a logical ring p_0, $p_1,\ldots,p_{n-1}$. 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.

2) Naive algorithm: We start by presenting a naive algorithm that illustrates both the basic idea behind our algorithm and its difficulty. Let the token carry an $estimate$ value (denoted by $token.estimate$) and the number of votes for this estimate (denoted $token.votes$). Let each process p_i, upon receiving the token, blindly add its vote to the proposal (see Procedure 2). Obviously, this naive algorithm does not work: it would solve consensus in an asynchronous system, in contradiction with the FLP impossibility result [10].

3) Overview of the token-accumulation consensus algorithm: As just shown, a token-accumulation algorithm cannot blindly increase the votes accumulated. We slightly change the above behavior. The token carries one additional information: $token.gap$

[7]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”.

Procedure 2 Token handling by p_i (option 1)

```
$p_i.estimate \leftarrow token.estimate$
$token.votes \leftarrow token.votes + 1$
**if** $token.votes \geq voteThreshold$ **then**
    decide($token.estimate$)
**end if**
send token to $p_{i+1}, \ldots, p_{i+f+1}$
```

which is the accumulated *gap* in the circulation of the token, defined as follows: when a process p_i receives the token from process $sender{\equiv}p_j$, the *gap* is $i{-}j{-}1$, denoted by $gap(sender{\rightarrow}p_i)$. We have $gap(sender{\rightarrow}p_i){=}0$ only if the token is received from the immediate predecessor. Upon receiving the token, a process does the following (see Procedure 3):

As long as the number of gaps in the token circulation is less than $gapThreshold$, and the token has not performed a full rotation of the ring ($token.gap+ token.votes < n$), $token.votes$ is incremented by the receiver p_i. If at that point $token.votes$ is greater than $voteThreshold$, p_i decides on the estimate of the token. The decision is then propagated with the token.

Procedure 3 Token handling by p_i (option 2)

```
$token.gap \leftarrow token.gap + gap(sender \rightarrow p_i)$
**if** ($token.gap > gapThreshold$) **or** ($token.gap + token.votes \geq n$) **then**
    $token.gap \leftarrow 0$;    $token.votes \leftarrow 0$ {*reset token*}
**end if**
$p_i.estimate \leftarrow token.estimate$
$token.votes \leftarrow token.votes + 1$
**if** $token.votes \geq voteThreshold$ **then**
    decide($token.estimate$)
**end if**
send token to $p_{i+1}, \ldots, p_{i+f+1}$
```

4) Conditions for agreement vs. termination: It is interesting to distinguish the conditions required for agreement and thoses required for termination. Agreement holds if

$$voteThreshold \geq (gapThreshold + 1)f + 1.$$

Termination holds with the failure detector $\mathcal{R}$, $gapThreshold = 0$, $voteThreshold = f + 1$ and $n \geq (f + 1)f + 1$.

5) Details of the algorithm: The token contains the following fields: $round$ (round number), $estimate$, $votes$ (accumulated votes for the $estimate$ value), gap (sum of all gaps in the token circulation) and $decision$ (a boolean indicating if $estimate$ is the decision).

Procedure 4 Consensus: Initialisation

```
1: $\forall p_i,\ i \in [0, n-1]$ :
2:     $estimate_i \leftarrow v_i$; $decided_i \leftarrow false$; $round_i \leftarrow 0$

3: $p_0$ :          {*send token*}
4:     send(0,$v_0$,1,0,false) to $\{p_1,\ldots,p_{f+1}\}$

5: $\forall p_i,\ i \in [n-f, n-1]$:          {*send "dummy" token*}
6:     send(-1,$\perp$,0,0,false) to $\{p_1, \ldots, p_{i+f+1}\}$
```

The initialization code is given by Procedure 4. Lines 5-6 show the *dummy* token sent to prevent blocking in the case processes $p_0, \ldots, p_{f-1}$ are initially crashed. A dummy token has $round = -1$, $estimate = \perp$ and $votes = 0$, and is sent only to processes $p_1, \ldots, p_f$.

The token handling code is given by Procedure 5. At line 2, process p_i starts by receiving the token (see Procedure 1) for the expected $round_i$.[8] If no value is transported by the token ($dummy$ initialization token), p_i replaces $token.estimate$ by its own estimate (lines 3-5). If p_i has not yet decided, then p_i starts by updating its estimate (line 7). At line 8 the gap is updated. If $gap_i \leq gapThreshold$ and the token has not performed a full circulation on the ring (line 9)[9], then the votes are incremented (line 10). Otherwise, the votes are reset to 1 and the gap to 0 (line 12), which starts a new sequence of vote accumulation. At line 14, process p_i checks whether there are enough votes for a decision to be taken. If so, p_i decides (line 15). Finally, the token with the updated fields is sent to the $f + 1$ successors (line 19), and process p_i increments $round_i$ (line 20).

Lines 1-21 ensure that at least one correct process eventually decides. However, if $f > 1$, this does not

[8]To avoid complicated notation, we implicitly assume that, for process p_i, waiting a token for $round_i$ means either (1) waiting a token from p_j, $j < i$, with $token.round = round_i$, or (2) waiting a token from p_j, $j > i$, with $token.round = round_i{-}1$.

[9]The condition on line 9 can be reduced to $gap_i \leq gapThreshold$. The algorithm would however need some minor modifications, which would imply a slightly more complex proof of Agreement.

Procedure 5 Token-accumulation consensus: token handling by p_i

```
1: loop
2:   token ← receive-token($round_i$)   {see Proc. 1}
3:   if $token.estimate = \perp$ then   {use initial value}
4:     $token.estimate \leftarrow estimate_i$
5:   end if
6:   if not $decided_i$ then
7:     $estimate_i \leftarrow token.estimate$
8:     $gap_i \leftarrow token.gap + gap(sender \rightarrow p_i)$
9:     if ($gap_i \leq gapThreshold$) and ($token.gap + token.votes < n$) then
10:      $votes_i \leftarrow token.votes + 1$   {add vote}
11:    else
12:      $votes_i \leftarrow 1$; $gap_i \leftarrow 0$   {reset votes/gap}
13:    end if
14:    if ($votes_i \geq voteThreshold$) or $token.decision$ then
15:      decide($estimate_i$); $decided_i \leftarrow true$
16:    end if
17:  end if
18:  token ← ($round_i, estimate_i, votes_i, gap_i, decided_i$)
19:  send token to $\{p_{i+1},\ldots,p_{i+f+1}\}$
20:  $round_i \leftarrow round_i + 1$
21: end loop

22: upon reception of token s.t. $token.round < round_i$ do
23:   if $token.decision$ and (not $decided_i$) then
24:     $estimate_i \leftarrow token.estimate$
25:     $decide(estimate_i)$; $decided_i \leftarrow true$
26:   end if
27: end upon
```

ensure that all correct processes eventually decide. Consider $gapThreshold = 0$ and 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 22-27 ensure that every correct process eventually decides. The token received at line 2, for $round_i$, follows Procedure 1. Other tokens are received at line 22: if the token carries a decision, process p_i decides. Note that stopping of the algorithm is not discussed here. It can easily be added.

6) Proof of the token-accumulation algorithm: The proofs of the uniform validity and uniform integrity properties are easy and omitted. The proof of the uniform agreement and termination properties are in the appendix.

V. Token-based atomic broadcast algorithms

In this section we show how to transform the token-accumulation 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 [3]. The reduction, which transforms atomic broadcast into a sequence of consensus, is however not adequate 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.

For simplification, we express the atomic broadcast algorithm for the case $gapThreshold = 0$ (the votes for a proposal are reset as soon as a gap is detected in the token circulation). To be correct, the atomic broadcast algorithm requires the failure detector $\mathcal{R}$, and the conditions $n \geq f(f+1)+1$, $voteThreshold = f+1$ (and $gapThreshold = 0$, which is hard-wired in the algorithm).

A. Overview

In the token-accumulation atomic broadcast algorithm, the token transports (i) sets of messages and (ii) sequences of messages. More precisely, the token carries the following information: *(round, proposalSeq, votes, adeliv, nextSet)*. Messages in the sequence *proposalSeq* are delivered as soon as a sufficient number of consecutive "votes" have been collected. The field *adeliv* is the sequence of all messages adelivered that the token is aware of (in the delivery order). When a process receives the token, it can therefore, if needed, catch up with the message deliveries performed by other processes.

Finally, while the token accumulates votes for *proposalSeq*, it simultaneously collects in *nextSet* the messages m such that $abroadcast(m)$ has been executed. The set *nextSet* grows as the token circulates. Whenever messages in *proposalSeq* can be delivered, *nextSet* is used as the "proposals" for the next decision.

Procedure 6 Atomic Broadcast: Initialisation

```
1: $\forall p_i,\ i \in [0, n-1]$ :
2:     $abroadcast_i \leftarrow \emptyset$; $adeliv_i \leftarrow \epsilon$: $round_i \leftarrow 0$

3: $p_0$ : {*send token*}
4:     send(0, $abroadcast_0$, 1, $\epsilon$, $abroadcast_0$) to $\{p_1,..,p_{f+1}\}$
5:
6: $\forall p_i,\ i \in [n-f, n-1]$: {*send "dummy" token*}
7:     $send(-1, \emptyset, 0, \epsilon, \emptyset)$ to $\{p_1,\ldots,p_{i+f+1}\}$
```

Procedure 7 Atomic Broacast: $abroadcast$ and $adeliver$ (code of p_i)

```
1: To execute $abroadcast(m)$:
2:     $abroadcast_i \leftarrow abroadcast_i \cup \{m\}$

3: To execute $delivery(seq)$:
4:     adeliver messages in $seq$ not in $adeliv_i$
5:     $adeliv_i \leftarrow adeliv_i \oplus seq$
6:     $abroadcast_i \leftarrow abroadcast_i \setminus adeliv_i$
```

B. Details

Each process p_i manages the following data structures (see Procedure 6): $round_i$ (the current round number), $abroadcast_i$ (the set of all messages that have been abroadcast by p_i or another process, and not yet ordered), and $adeliv_i$ (the sequence of messages adelivered by p_i). The algorithm is decomposed into several procedures.

Procedure 6 is the initialization procedure (ϵ denotes the empty sequence).

Procedure 7 describes the *abroadcast* and *adelivery* of messages: $delivery(seq)$ is called by Procedure 8. The operator $\oplus$ at line 5 of Procedure 7 is the sequence concatenation operator ($seq_1 \oplus seq_2$ is the sequence of elements in seq_1 concatenated with the sequence of elements in seq_2 that are not in seq_1).

Procedure 8 Atomic broadcast: token handling by p_i

```
1: loop
2:     token $\leftarrow$ receive-token($round_i$) {*see Procedure 1*}

3:     $abroadcast_i \leftarrow abroadcast_i \cup token.proposalSeq \cup token.nextSet$

4:     if $|token.adeliv| < |adeliv_i|$ then {*$p_i$ more up to date than the token*}
5:         $token.proposalSeq \leftarrow \emptyset$
6:     else {$|adeliv_i| \leq |token.adeliv|$}
7:         $delivery(token.adeliv)$
8:         if (token received from $p_{i-1}$) and ($token.proposalSeq \neq \emptyset$) then
9:             $votes_i \leftarrow token.votes + 1$
10:        else
11:            $votes_i \leftarrow 1$
12:        end if

13:        if ($votes_i \geq f+1$) then
14:            $delivery(token.proposalSeq)$
15:            $token.proposalSeq \leftarrow \emptyset$
16:        end if
17:    end if

18:    if $token.proposalSeq = \emptyset$ then {*new proposal can be made...*}
19:        $token.proposalSeq \leftarrow abroadcast_i$ {*add new "proposals"*}
20:        $votes_i = 1$
21:    end if
22:    token $\leftarrow (round_i, token.proposalSeq,$
                   $votes_i, adeliv_i, abroadcast_i)$
23:    send token to $\{p_{i+1}..p_{i+f+1}\}$
24:    $round_i \leftarrow round_i + 1$
25: end loop

26: upon reception of token s.t.
            $token.round < round_i$ do
27:     if $|token.adeliv| > |adeliv_i|$ then{*the token has "new" information*}
28:         $delivery(token.adeliv)$
29:     end if

30:     $abroadcast_i \leftarrow abroadcast_i \cup token.nextSet$
31: end upon
```

Procedure 8 describes the token-handling. Lines 4 to 17 of Procedure 8 correspond to lines 6-17 of the consensus algorithm (Procedure 5). Procedure $delivery()$ is called to deliver messages (line 14). When this happens, a new sequence of messages can be proposed for delivery. This is done at lines 18 to 21. Finally, lines 26-31 handle reception of other tokens. This is needed for Uniform Agreement and Validity when $f > 1$. Lines 27-29 are for Uniform Agreement (they play the same role as lines 23-25 of Procedure 5). Line 30 is for Validity (consider $f = 2$, p_i correct and p_{i+1} faulty; without line 30, process p_{i+2} might, in all rounds, receive the token only from p_{i-1}; if this happens, messages *abroadcast* by p_i would never be *adelivered*).

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

C. Optimization

In our algorithm, the token carries whole messages, rather than only message identifiers. This solution is certainly inefficient. The algorithm can be optimized so that only the message identifiers are included in the token. However, this solution requires messages to be reliably broadcast by each host. This approach leads to a slightly more complex algorithm not given here as this issue is orthogonal to the contribution of the paper.

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 $f + 1$ predecessors,[10] requesting the token. 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 due to the token circulation by a factor $f + 1$.

[10]Actually, the message does not need to be sent by p_i to p_{i-1}.

VI. Related work

As was mentioned in Section I, previous atomic broadcast protocols based on tokens need group membership or an equivalent mechanism. In the Chang and Maxemchuk's Reliable Broadcast Protocol [11], and its newer variant [9] an ad-hoc reformation mechanism is called whenever a host fails. Group membership is used explicitly in other atomic broadcast protocols such as Totem [8], the Reliable Multicast Protocol by Whetten et al. [7] (derived from [11]), and in [12].

These atomic broadcast protocols also have different approaches with respect to message broadcasting and delivery. In [11], [7], 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 [8] or On-Demand [12] on the other hand only enable the token-holder to broadcast (and simultaneously order) messages.

Finally, the different token-based atomic broadcast protocols deliver messages in different ways. In [12], 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 [11], a message is only delivered once $f + 1$ sites have received the message.

Larrea *et al.* [13] also consider a logical ring of processes, however with a different goal. 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.

VII. Conclusion

According to various authors, token-based atomic broadcast algorithms are more efficient in terms of throughput than other atomic broadcast algorithms. The reason is that the token can be used to reduce network contention. However, all published token-based algorithms rely on a group membership service, i.e., none of them can use a failure detection mechanism. The paper has given 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 to tolerate failures (i.e., it also tolerates false failure suspicions). Algorithms that do not tolerate failures, need to rely on a membership service to exclude crashed processes. As a side-effect, these algorithms also exclude correct processes that have been incorrectly suspected. Thus, failure detector based algorithms have advantages over group membership based algorithms, in case of false failure suspicions, and possibly also in case of real crashes.

In the future we plan to compare the performance of these two classes of token-based atomic broadcast algorithms (*failure-detector* based and *membership* based) in a similar way as done in [6], for a different class of atomic broadcast algorithms. Note that these experiments may require to address practical issues not addressed here, such as reducing the size of the information carried by the token, carrying in the token message identifiers rather then whole messages, and also adapting the algorithm to fair-lossy channels, as in [11], [9].

Acknowledgements. We would like to thank Bernadette Charron-Bost for discussions related to failure detectors and Péter Urbán for useful comments on an earlier version of the paper.

REFERENCES

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

[2] Xavier Défago, André Schiper, and Péter Urbán, “Totally ordered broadcast and multicast algorithms: A comprehensive survey,” Tech. Rep. DSC/2000/036, École Polytechnique Fédérale de Lausanne, Switzerland, Sept. 2000.

[3] T. D. Chandra and S. Toueg, “Unreliable failure detectors for reliable distributed systems,” *Journal of ACM*, vol. 43, no. 2, pp. 225–267, 1996.

[4] G.V. Chockler, I. Keidar, and R. Vitenberg, “Group Communication Specifications: A Comprehensive Study,” *ACM Computing Surveys*, vol. 4, no. 33, pp. 1–43, December 2001.

[5] K. Birman, A. Schiper, and P. Stephenson, “Lightweight causal and atomic group multicast,” *ACM Trans. on Computer Systems*, vol. 9, no. 3, pp. 272–314, Aug. 1991.

[6] Péter Urbán, Ilya Shnayderman, and André 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)*, June 2003, pp. 645–654.

[7] B. Whetten, T. Montgomery, and S. Kaplan, “A high performance totally ordered multicast protocol,” in *Theory and Practice in Distributed Systems*, Springer-Verlag, Ed., Dagstuhl Castle, Germany, Sept. 1994, number 938 in Lecture Notes in Computer Science, pp. 33–57.

[8] Y. Amir, L.E. Moser, P.M. Melliar-Smith, D.A. Agarwal, and P.Ciarfella, “The Totem single-ring ordering and membership protocol,” *ACM Trans. on Computer Systems*, vol. 13, no. 4, pp. 311–342, November 1995.

[9] N. F. Maxemchuk and D. H. Shur, “An Internet multicast system for the stock market,” *ACM Trans. on Computer Systems*, vol. 19, no. 3, pp. 384–412, August 2001.

[10] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson, “Impossibility of distributed consensus with one faulty process,” *Journal of ACM*, vol. 32, no. 2, pp. 374–382, Apr. 1985.

[11] J. Chang and N. F. Maxemchuk, “Reliable broadcast protocols,” *ACM Trans. on Computer Systems*, vol. 2, no. 3, pp. 251–273, Aug. 1984.

[12] F. Cristian, S. Mishra, and G. Alvarez, “High-performance asynchronous atomic broadcast,” *Distributed System Engineering Journal*, vol. 4, no. 2, pp. 109–128, June 1997.

[13] Mikel Larrea, Sergio Arevalo, and Antonio Fernandez, “Efficient algorithms to implement unreliable failure detectors in partially synchronous systems,” in *International Symposium on Distributed Computing*, 1999, pp. 34–48.

APPENDIX

A. Correctness of the token-accumulation consensus algorithm

a) (Uniform) Agreement (sketch).: We prove that $voteThreshold \geq (gapThreshold+1)f+1$ ensures *uniform* agreement.[11] The proof is by induction on $gapThreshold$.

i) Base case: $gapThreshold = 0$. Let p_i be the first process to decide (say at time t), and let v be the decision value. By line 14 of Procedure 5, we have $votes_i \geq voteThreshold \geq (gapThreshold+1)f+1 \geq f+1$. Since $gapThreshold = 0$, the votes are reset for each gap. So, $votes_i \geq f+1$ ensures that at time t, all processes $p_j \in \{p_{i-1}, \ldots, p_{i-f}\}$, have $p_j.estimate = v$. Any process p_k, successor of p_i in the ring, receives the token from one of the processes $p_i, \ldots, p_{i-f}$. Since all these processes have their estimate equal to v, the token received by p_k necessarily carries the estimate v. So after t, the only value carried by the token is v, i.e., any process that decides will decide v.

ii) Induction step. We proove that if uniform agreeement holds for $gapThreshold = l$, it holds for $gapThreshold = l+1$. We introduce the following notation: $votes_{x,y}$ is the number of votes collected by the token between (and including) processes p_x and p_y, and $gap_{x,y}$ is the number of gaps experienced by the token when moving from p_x to p_y. Let p_i be the first process to decide (say at time t), and let v be the decision value. Let $votes_i$ and gap_i be the number of votes accumulated and the gap experienced by the token when p_i decides. The most recent reset of the votes/gap occurred at process $p_k \equiv p_{i-gap_i-(votes_i-1)}$ (by line 9, all processes $\{p_k \ldots p_i\}$ are different). So, $gap_i = gap_{k,i}$ and $votes_i = votes_{k,i}$. We consider two cases (see Figure 1): (1) the token has circulated without gap from p_k to p_{k+f}, (2) the token has circulated with a gap of at least 1 from p_k to p_{k+f}.

In case (1), when p_k receives the token, all $f+1$ processes p_k to p_{k+f} have set their estimate to v. Therefore, the token received by any process after p_{k+f}, including the processes after p_i, only carries v. Thus, after t, the only value carried by the token is v, i.e., any process that decides will decide v.

[11]Two processes, correct or faulty, cannot decide differently.

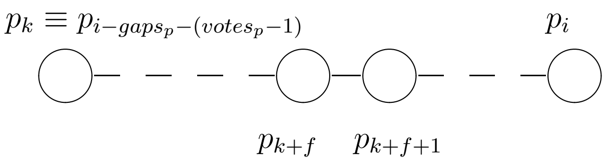

Fig. 1. Proof by induction for Agreement

In case (2) we have $gap_{k,k+f} \geq 1$, and so necessarily $votes_{k,k+f} \leq f$. When p_i decides, we have $gap_i = gap_{k,i} \leq l+1$ (*) and $votes_i = votes_{k,i} \geq voteThreshold \geq (gapThreshold + 1)f+1 = (l+2)f+1$ (**). Since $gap_{k,k+f} \geq 1$, we have by (*): $gap_{k+f+1,i} = gap_{k,i} - gap_{k,k+f} \leq gap_{k,i} - 1 \leq l$ (***). Since $votes_{k,k+f} \leq f$, we have by (**): $votes_{k+f+1,i} = votes_{k,i} - votes_{k,k+f} \geq votes_{k,i} - f \geq (l+2)f+1-f = (l+1)f+1$ (****). By (***) and (****), when p_i decides, the token circulation between p_{k+f+1} and p_i satisfies the induction hypothesis. When p_i decides, uniform agreement is therefore ensured. $\square$

b) Termination (sketch).: Assume at most f faulty processes and the failure detector $\mathcal{R}$. We show that, if $n \geq f(f+1)+1$, $gapThreshold = 0$,[12] and $voteThreshold = (gapThreshold+1)f+1 = f+1$, 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 *receive token* procedure (Procedure 1), 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 processes eventually decides. Assume the failure detector $\mathcal{R}$, $gapThreshold = 0$, 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 (see section III-C). Let $p_i \ldots p_{i+f}$ be this sequence. After t, processes

[12]Agreement does not require $gapThreshold = 0$. However, with the failure detector $\mathcal{R}$, $gapThreshold > 0$ does not ensure termination.

$p_{i+1} \dots p_{i+f}$ 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 10 of Procedure 5, we have $votes_i \geq f + 1 = voteThreshold$. 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 22, 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. $\square$