Fault-Tolerant Termination Detection with Safra's Algorithm

Wan Fokkink
Department of Computer Science
Vrije Universiteit Amsterdam
Amsterdam, the Netherlands
w.j.fokkink@vu.nl

Per Fuchs
Department of Computer Science
Technische Universität München
München, Germany
per.fuchs@cs.tum.edu

Georgios Karlos
Department of Computer Science
Vrije Universiteit Amsterdam
Amsterdam, the Netherlands
g.karlos@vu.nl

***Abstract*—We adapt Safra's distributed termination detection algorithm to make it fault-tolerant in the presence of node failures. The algorithm is based on an implicit token ring structure within the network; only passive nodes forward the token, and a counter in the token keeps track of the number of sent minus the number of received messages. We split this single counter into counters per node, so that counts from crashed nodes can be discarded. If a node crashes, the token ring structure is restored locally and a backup token is sent. Nodes inform each other of detected crashes via the token. Our algorithm imposes no additional message overhead, copes with crashes in a decentralized fashion, can tolerate $N-1$ crashes, and is robust against simultaneous crashes. We report on experiments with an emulator as well as an implementation of our algorithm on top of two fault-tolerant distributed algorithms from the literature.**

I. Introduction

Termination detection is a fundamental problem in distributed systems, which was introduced independently in [9], [12]. Termination can be announced when all nodes in the network have become passive and no messages are in transit. Many (mostly failure-sensitive) termination detection algorithms have been presented in the literature; a taxonomy is given in [16]. In Safra's algorithm [7], [10] a token repeatedly visits all the nodes in the network via a predetermined ring structure starting at the initiator node; a node passes on the token when it is passive. Each node keeps track of the number of outgoing minus incoming messages, and these counts are accumulated in the token. Nodes that receive a message are colored black, as the message count in the token is unreliable if the message overtakes the token in the ring; the black color is transferred to the token at its next visit. If the token returns to the initiator without a black color and with counter 0, the initiator can announce termination. In [6] an optimized version of Safra's algorithm was proposed, that does not always color receiving nodes black and detects termination within a single round trip of the token after actual termination has occurred.

We present a novel fault-tolerant algorithm that can cope with permanent node crashes. It builds upon the improved version of Safra's algorithm from [6], employing a failure detector. A node crash is handled locally at its predecessor, whereby the token ring structure is restored, and a new token is sent since the old token may have been lost during the crash. A numbering scheme in the token makes sure that only a single token is being passed on; if the old token was not lost in the crash, the new token will be dismissed. Only the message exchange between alive nodes is being counted at the nodes and in the token. For this purpose the counter at the nodes and in the token is split into N counters, with N the number of nodes in the network. Nodes inform each other of node crashes, through the token, so that they uniformly count the message exchange with the same set of nodes. A node reporting a new crashed node makes sure the token completes another round trip, to avoid that the token accumulates inconsistent message counts.

Following e.g. [21], we call the distributed algorithm for which termination is checked basic and the termination detection algorithm the control algorithm. A basic or control message is invoked by the basic or control algorithm, respectively. Safra's algorithm has two main advantages in comparison to other termination detection algorithms. First, if some node in the system remains active over a long period of time, then Safra's algorithm imposes only little control message overhead, unlike termination detection algorithms for which every basic message needs to be acknowledged (e.g. [9]). Second, Safra's algorithm does not run into issues with floating point arithmetic precision or so-called underflow, as is the case with the weight-throwing scheme from [18]. Our fault-tolerant version of Safra's algorithm has some additional strong points, compared to existing fault-tolerant termination detection algorithms, which are discussed in Section III. Our algorithm is fully decentralized, except for the initialization of the token. If the node that should send out the first token crashes before doing so, this role is automatically taken over by its predecessor in the ring. Thus our algorithm can cope with any number of node crashes, is robust against multiple simultaneous node crashes, and does not require a leader election scheme. It requires only one additional message for each crash. The price to pay is that, in the absence of stable storage, the bit complexity of a message is $\Theta(N)$, compared to $\Theta(1)$ for the failure-sensitive version of Safra's algorithm. Considering current network technologies, with Gbits/second throughput and microseconds latency, this token size incurs a tolerable overhead in network load, especially since the token is only forwarded by idle nodes.

We implemented the algorithm, developed a dedicated

multi-threaded emulation environment to test its functional correctness, and performed experiments on two fault-tolerant distributed algorithms from the literature. Developing our algorithm was a delicate matter, we made several flawed attempts before coming up with the design presented here. Still, owing to a formal correctness proof in combination with extensive test runs with an actual implementation, we can confidently claim that our algorithm correctly detects termination, even when $N-1$ nodes crash. We compared it with an implementation of the failure-sensitive version of Safra's algorithm. Our implementation exhibits a satisfactory performance, in the sense that it imposes no additional message overhead. Of course it does impose an overhead compared to the failure-sensitive version, because it adds extra concurrency at each node and requires additional synchronization. However, experiments show that even with a large number of failures, the dominant factor for the overall performance is the execution time of the basic algorithm.

II. Background

A node in a distributed computer network is either *active* or *passive*. An active node can send or receive basic messages, perform internal events, or become passive when it *terminates* locally. A passive node cannot send basic messages or perform internal events. An active node may activate a passive node by sending a basic message to it. In a fully asynchronous setting, inevitably termination may be announced while a basic message sent by a crashed node is still in transit. It is therefore required that passive nodes never become active by the receipt of a basic message from a node they know has crashed. An execution of the basic algorithm has terminated if all alive nodes are passive and for all basic messages in transit, the destination node either has crashed or knows that the sender has crashed. The termination detection problem consists of two parts: *Liveness*: if the system has terminated, this is eventually detected by an alive node; and *Safety*: when termination is detected, the system terminated at some point in the past.

III. Related work

We discuss some existing fault-tolerant termination detection algorithms, mainly from a functional point of view. We note that [15] is the only one to report performance results based on an actual implementation. Generally a complete network topology and a perfect failure detector are required.

Lai and Wu [14] presented a fault-tolerant variant of the tree-based approach of Dijkstra and Scholten [9] for centralized basic algorithms. All active nodes are in the tree, rooted in the initiator, which announces termination when the tree has disappeared. In the Lai-Wu algorithm, in the event of a crash, every alive node communicates with a designated root node, causing a sequential bottleneck. Lifflander et al. [15] proposed a series of fault-tolerant termination detection algorithms based on the Dijkstra-Scholten algorithm. In case of a crash the tree is reconstructed locally. If a pair of nodes fails concurrently, the algorithm may not be able to recover. The algorithm then detects that this is the case and issues an Irrecoverable Detection message. The algorithms from [14], [15] incur additional control messages even in crash-free executions. In [15] performance results are reported based on an experimental setup, consisting of three mock-up parallel algorithm implementations (N queens and methods from computational chemistry and tensor calculus). These experiments show most crashes are handled successfully and the performance overhead in crash-free executions is acceptable.

Tseng [22] developed a fault-tolerant variant of the weight-throwing scheme from [18] for centralized basic algorithms, meaning that they must have a single initiator. Node donate part of their weight to the basic messages they send. Upon receipt of such a message, the receiving node adds this weight to its own weight. A node that becomes passive returns its weight to the leader. The leader announces termination once it has reclaimed its original weight. Since the leader may crash, an election scheme is needed. A global snapshot is taken when a new crashed node is detected or a new leader is elected.

In Venkatesan's algorithm [23] a leader node is in charge of announcing termination, and a backup leader is employed in case the leader crashes. If the leader crashes, an election is held. The local stacks at the nodes must be continuously duplicated to the leader and its backup, so that upon learning of a crashed node, a leader can simulate the state of every node in the system to determine whether it has terminated.

Hursey and Graham [13] developed a termination detection scheme for their fault-tolerant ring-based MPI application. In case of a crash, the ring structure is recovered locally. If the root of the network crashes, a new leader is elected.

Dash and Hansdah [5] employed roughly synchronized local clocks for fault-tolerant termination detection, using timestamped control messages that are circulated through the network via a predetermined ring structure. Recovery of a crashed node by means of stable storage, which can still be read after a crash, is in some circumstances essential.

Mittal et al. [19] introduced a nice general framework for transforming any failure-sensitive termination detection algorithm into a fault-tolerant, wait-free variant that tolerates any number of node crashes. The basic idea is to restart termination detection after each node crash. When applied to existing failure-sensitive algorithms, the resulting fault-tolerant algorithms can give a significant overhead in control messages.

A termination detection algorithm for mobile ad hoc networks was developed by Matocha [17], based on the Dijkstra-Scholten algorithm, and by Roman and Payton [20], based on a mix of the Dijkstra-Scholten algorithm and weight throwing.

IV. Safra's algorithm

The N nodes carry a unique id. Messages are exchanged between nodes asynchronously via directed channels. Messages may arrive in any order. Message delays are unbounded.

A. Informal description of Safra's algorithm

Safra's (failure-sensitive) termination detection algorithm generalizes the algorithm by Dijkstra, Feijen and van Gasteren

[8] from synchronous to asynchronous message passing networks. A detailed description of Safra's algorithm and its adaptations from [6] is required in order to properly explain our fault-tolerant version later on. Assuming (for the moment) a crash-free network, nodes are organized in a directed ring structure within the network topology. Node identities $0, \ldots, N-1$ are assigned in a strictly increasing order in the ring: for every node i, node $i+1$ is its successor and node $i-1$ its predecessor, modulo N. Safra's algorithm is centralized: node 0 is supposed to be the initiator of the algorithm, while all other nodes are noninitiators. The basic algorithm however is allowed to be decentralized, meaning that there can be multiple concurrent initiators of the algorithm.

Local variables at a node i are written with subscript i and variables in the token with subscript t. Each node i records in $count_i$ the number of basic messages it sent minus the number of basic messages it received, since the last time it forwarded the token. A token t circulates the ring to take a snapshot of the system, starting at the initiator of the control algorithm when it becomes passive for the first time, and being forwarded by the other nodes once they are passive. The field $count_t$ in t represents the number of basic messages in transit during the snapshot. Each time t is forwarded by a node i, $count_i$ is added to $count_t$, and $count_i$ is reset to 0. Upon return of t to the initiator, after it has become passive and added $count_0$ to $count_t$, termination is detected if the latter value is 0.

Consistency of a snapshot is violated if t overtakes a basic message m, meaning that an active node sends m, then becomes passive, and forwards t, while t reaches the destination of m before m. In that case the send of m is recorded in $count_t$ but not its receipt. This leads to a positive value of $count_t$, so that no termination can be detected. Consistency of the snapshot is also violated if a basic message m overtakes t: a node that forwarded t becomes active (because it receives a basic message) and sends m, which reaches its destination before t. To recognize such situations, the algorithm introduces colors black and white. The Boolean field $black_i$ is *true* if and only if node i is considered black; else i is considered white. Likewise $black_t$ represents the color of the token. Initially all nodes are white, and when the initiator sends out a fresh token it carries the color white. Whenever a node i receives a basic message m, there is the possibility that the send of m was not recorded in $count_t$. Therefore upon receipt of m, i marks itself black. Once t is forwarded by a black noninitiator, t becomes black and the forwarding node white; from then on t remains black for the rest of the round trip. When the initiator has received back t, has become passive, and has added the value of $count_0$ to $count_t$, it decides whether termination can be detected. If t or the initiator is black or $count_t$ is not 0, the initiator becomes white and sends out a fresh white token again. Else it can safely announce that the execution run of the basic algorithm has terminated.

B. Two improvements for Safra's algorithm

Demirbas and Arora [6] proposed two enhancements for Safra's algorithm, with the aim to detect termination with fewer token steps after actual termination has occurred.

One inefficiency of Safra's algorithm is that a basic message always blackens the receiving node. Actually only a basic message that overtakes the token in the ring causes an inconsistent snapshot, that is, a basic message that is sent by a node after it has been visited by the token and received by a node that has not yet been visited by this token. Therefore only such messages should blacken the receiving node. The first enhancement introduces a sequence number seq_i for every node i, which initially is 0. At the moment a node forwards the token to its successor, it increases its sequence number by 1, so that nodes in the visited region have a different sequence number from those in the unvisited region. A node piggybacks its sequence number as well as its identity to every basic message m it sends; the sequence number in m is called seq_m. Based on this information the receiver can determine whether or not m has overtaken the token (see below), and only blackens itself if this is indeed the case.

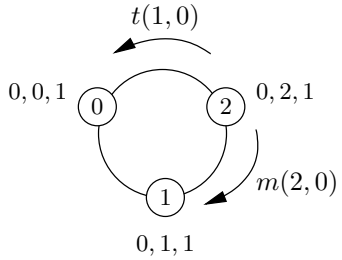
Another inefficiency of Safra's algorithm is that termination can only be detected at the initiator. The second enhancement allows to detect termination at any node. This means that in case a basic message m overtakes the token, not only the receiver i but also all subsequent nodes in the ring up to (but not including) the sender of m, denoted by $sender_m$, should be turned black. Namely, at all these nodes the token represents an inconsistent snapshot, because it takes into account the receipt but not the sending of m. So none of these nodes should be able to detect termination. If m has overtaken the token, then one of the following two cases holds: either $sender_m < i$ and $seq_m = seq_i + 1$, or $sender_m > i$ and $seq_m = seq_i$.

The field $black_t$ in the token becomes a node index, expressing that all nodes the token visits from now up to (but not including) $black_t$ are black. When the token is sent by the initiator for the first time, $count_t$ is set to the value of $count_0$. Moreover, $black_t = N-1$, so that all nodes from 1 up to $N-1$ are initially considered black. Hence termination can only be detected after the token has visited all nodes. Likewise, $black_i$ at a node i represents that all nodes that the token visits from i up to $black_i$ are black. Initially $black_i = i$ at all nodes i, meaning that i considers all nodes white. If a node i receives a basic message m that has overtaken the token, then $black_i$ is set to the furthest node from i among $black_i$ and $sender_m$. The function $furthest_i(j, k)$ computes whether node j or k is furthest away from i in the ring. It is defined by: k if $i \leq j \leq k$ or $k < i \leq j$ or $j \leq k < i$; and j otherwise.

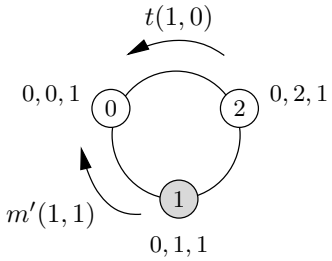
When the token reaches a node i, it must wait until i has become passive. Then i adds the value of $count_i$ to the value of $count_t$. If i is considered white, meaning that $black_t = black_i = i$, then it can determine termination in the same way the initiator does in Safra's algorithm: check whether the value of $count_t$ is 0. If i is black or detects no termination, it forwards the token to its successor. Before doing so, it sets $black_t$ to $furthest_i(black_t, black_i)$ if this value is not i, or else to $(i+1) \bmod N$. The latter means the successor of i in the ring will consider the token white. Finally, i sets $count_i$ to 0 and $black_i$ to i and increases seq_i by 1.

The pseudocode of the four procedures available at each node regarding the improved version of Safra's algorithm are presented in Procedures 1-4: initialization (INIT), sending and receiving a basic message (SBM and RBM), and handling a received token (RT). The subscript i at each procedure name represents the node where the procedure is performed. The action *send*(m, j) denotes that message m is sent to node j, and the Boolean field *passive*$_i$ is *true* only while node i is passive. The procedures are supposed to be executed without interruption. The only exception is that while waiting to become passive, in line 2 of procedure RT, a node is allowed to perform SBM and RBM calls. By *update*(t) we denote that a node updates its local token variables.

Example 1. We consider one possible run of the improved version of Safra's algorithm on a ring of three nodes, with *count*$_i$ = *seq*$_i$ = 0 and *black*$_i$ = i at all nodes i. Initially only node 2 is active; it sends a basic message m to node 1, with its identity 2 and its *seq*$_2$ value 0 attached, sets *count*$_2$ to 1, and becomes passive. The initiator 0 sends out a token with *count*$_t$ = 0 and *black*$_t$ = 2, which travels via node 1 to node 2, where *count*$_t$ is set to 1. On the way, *seq*$_0$, *seq*$_1$ and *seq*$_2$ are set to 1 and *count*$_2$ is set to 0. At nodes 1 and 2 no termination is detected because *black*$_t$ = 2 and *count*$_t$ = 1, respectively. Node 2 sends the token to the initiator with *count*$_t$ = 1 and *black*$_t$ = 0, leading to the following picture. Node identities are placed within the nodes; white nodes are passive and gray nodes (in the picture further below) are active; the triple at each node represents the values of *count*, *black* and *seq*, respectively, at this node.



Next m reaches node 1, making it active and setting *count*$_1$ to -1; the value of *black*$_1$ remains 1 because *sender*$_m$ = 2 > 1 and *seq*$_m$ = 0 ≠ *seq*$_1$. Node 1 sends a basic message m' to the initiator, with its identity 1 and its *seq*$_1$ value 1 attached, and sets *count*$_1$ to 0. This leads to the picture below.



m' reaches the initiator before the token, making it active and setting *count*$_0$ to -1. Moreover, *black*$_0$ is set to 1, since *sender*$_{m'}$ = 1 > 0 and *seq*$_{m'}$ = 1 = *seq*$_0$. Now the initiator becomes passive again. The token reaches the initiator, where *count*$_t$ and *count*$_0$ are both set to 0. No termination is detected because *black*$_0$ = 1. The token travels on to node 1 with *black*$_t$ = 1. Next this node becomes passive and announces termination because *count*$_t$ = 0 and *black*$_1$ = *black*$_t$ = 1.

At some points we deviated from the algorithm in [6]. There the second disjunct in line 2 of the pseudocode of procedure RBM was forgotten. Their approach is limited to centralized basic algorithms; in case of a decentralized algorithm, premature termination detection would be possible before the token has completed one round trip, because initially all nodes are white. They interpret sequence numbers modulo 2, so that a receiving node is spuriously blackened in situations where a basic message is overtaken by the token. Last but not least, we condensed N Boolean fields into a single node index *black* at the nodes and in the token, and N integer fields in the token into a single counter.

We argue that the improved version of Safra's algorithm correctly detects termination, because we modified the algorithm in [6] and the proof sketch is a corner stone for the correctness argument of our fault-tolerant algorithm later on.

Procedure 1: Initialization$_i$

```
1 count_i ← 0;   black_i ← i;   seq_i ← 0;
2 if i = 0 then
3     wait(passive_0);
4     count_t ← count_0;   black_t ← N − 1;
5     send(t, 1);   count_0 ← 0;   seq_0 ← 1;
```

Procedure 2: SendBasicMessage$_i(m, j)$

```
1 seq_m ← seq_i;   sender_m ← i;
2 send(m, j);   count_i ← count_i + 1;
```

Procedure 3: ReceivedBasicMessage$_i$

```
1 m ← dequeue(MessageQueue_i);
2 if (sender_m < i ∧ seq_m = seq_i + 1) ∨
     (sender_m > i ∧ seq_m = seq_i) then
3     black_i ← furthest_i(black_i, sender_m);
4 count_i ← count_i − 1;
```

Procedure 4: ReceivedToken$_i$

```
1 update(t);   wait(passive_i);
2 count_t ← count_t + count_i;
3 black_i ← furthest_i(black_i, black_t);
4 if count_t = 0 ∧ black_i = i then
5     Announce;
6 black_t ← furthest_i(black_i, (i + 1) mod N);
7 send(t, (i + 1) mod N);
8 count_i ← 0;   black_i ← i;   seq_i ← seq_i + 1;
```

Theorem 1. The improved Safra's algorithm is a correct termination detection algorithm.

Proof: First we argue *Liveness*: if the basic algorithm has terminated, then eventually termination will be announced. Since all nodes are passive, they all proceed beyond *wait*(*passive*$_i$)

in line 1 of RT and pass on t. The counter of t accumulates all changes in the counters at the nodes (by line 2 and $count_i \leftarrow 0$ in line 8 of RT). Since the sender of a basic message increases its counter by 1 (in line 2 of SBM) and the receiver decreases its counter by 1 (line 4 of RBM), and there are no basic messages in the channels, $count_t$ eventually (and permanently) becomes 0. Furthermore, since RBM will not be called anymore at any node, $black_i$ and $black_t$ are only mutated in lines 3,8 and line 6 of RT_i, respectively. This implies that eventually $black_i$ permanently becomes i, while t always reaches a node i with $black_t = i$. Hence eventually t reaches a node i with $count_t = 0$ and $black_i = i$, so that Announce is called (lines 4-5 of RT_i).

Now we argue *Safety*: if termination is announced, then the basic algorithm has terminated. Let some node i reach line 4 of RT_i with $black_i = i$. We argue that then $count_t \geq 0$. Namely, the only way this value could be negative at this point is if some basic message m was accounted for in $count_t$ at its receiver k, but not at its sender j, meaning that (1) m overtook t, and (2) j is not between k and i, i.e., $furthest_k(i, j) \neq i$. By (1), when k received m, the condition in line 2 of RBM_k would be true, so by line 3 of RBM_k together with (2), $black_k$ would be pushed beyond i. So at the last visit of t to k, $black_t$ was pushed beyond i. Then $black_i$ would have been pushed beyond i at line 3 of RT_i, contradicting the assumption that $black_i = i$ at line 4 of RT_i. Suppose now that at line 4 of RT_i the basic algorithm has not yet terminated. We argue that then $black_i = i$ implies $count_t > 0$. There are two cases to consider. The first case is that some node is still active. Since nodes must be passive to forward t and i is passive (by $wait(passive_i)$ in line 1 of RT), some node $k \neq i$ was made active after it forwarded t for the last time, by a basic message m from a node j with $furthest_k(i, j) = i$, while t was traveling from k to j. Since sending m by j is accounted for in $count_t$ but not the reception of m at k, by the time t reaches i, $count_t > 0$. The second case is that some basic message is traveling from a node j to a node k. Either sending this message was accounted for in $count_t$, which implies that its value at i is positive. Or $j \neq i$ became active after it forwarded t for the last time. Then it follows as in the first case that $count_t > 0$. □

V. Fault-tolerant version

From now on we consider the fail-stop model. Nodes may spontaneously and permanently crash – nodes work correctly until such a crash occurs. It is customary to assume for fault-tolerant distributed algorithms that there is a bidirectional channel between each pair of distinct nodes (see e.g. [11], [21]), because else a node failure may result in disconnected subnetworks. Actually, for our algorithm it suffices to require that at any time a channel can be established between any two alive nodes. We moreover assume that there is a failure detector for which no spurious crash detection can happen and each node crash is eventually detected at all (alive) nodes. In fact, without a so-called perfect failure detector it is impossible to solve the termination detection problem in a faulty network [19]. Such a mechanism can be easily implemented in case of a known upper bound on network latency, by letting nodes send heartbeat messages at regular time intervals. Actually, in our token ring algorithm it suffices if a node sends these heartbeat messages only to its predecessor and its successor in the ring, as identities of crashed nodes are announced through the token. Each node i stores the identities of crashed nodes in one of the two sets CRASHED_i and REPORT_i. The latter set contains the identities that i has not yet reported to the other alive nodes by means of the token; this is explained below.

Since message counts related to crashed nodes need to be discarded, the token contains N counters, one per node; moreover, each node needs to count its message exchange with each other node separately and from the start of the execution run (instead of since the last token visit). So we split the field $count_i$ for each node i into a sequence $[count_i^0, \ldots, count_i^{N-1}]$. For each node j, the field $count_i^j$ stores the number of basic messages i has sent to j minus the number of basic messages i has received from j. (The fields $count_i^i$ are redundant as they always carry the value 0.) If (the failure detector of) i detects that a node j has crashed, then i permanently disregards the value of $count_i^j$. Likewise, to separately keep track of the counters at the different nodes in the token, the field $count_t$ is split into a sequence $[count_t^0, \ldots, count_t^{N-1}]$. If these counters were lumped together into a single counter $count_t$, and say a node i sent a basic message to a node j which then crashed, there may be no way of telling whether or not j received this message and updated $count_j^i$ and $count_t$.

If a node i learns from its failure detector that some other node j crashed, it must share this information with the other alive nodes via the token. Else there would be the risk that although i from now on disregards $count_i^j$, some other alive node k may still take into account $count_k^j$, which could lead to a premature termination detection at k. For this purpose the token contains a set CRASHED_t. When i forwards the token with $j \in \text{CRASHED}_t$, it makes sure that $j \in \text{CRASHED}_i$, to avoid that it announces the same crashed node multiple times.

Another adaptation is that each node i must keep track of its successor $next_i$ in the ring. Initially its value is $(i+1) \bmod N$. Each time i detects $next_i$ has crashed, the value of this field is changed into i's nearest alive successor in the ring. We must ensure the token is not lost; this could happen if the token was traveling to or being handled by $next_i$ at the moment it crashed. Therefore, after having determined its new successor $next_i'$ in the ring, i forwards the token once again, but this time to $next_i'$. For this purpose i stores the content of the last token it forwarded. These local variables are updated as soon another token (with a higher sequence number) arrives. By abuse of notation we denote a variable in the token and its local copies at nodes with the same name, subscripted with t.

In case $next_i$ forwarded the token before crashing, $next_i'$ will receive the same token twice. Therefore the token is endowed with a sequence number seq_t, which is increased by 1 at each consecutive round trip of the token. In the first round this sequence number is 1. Each node i keeps track of the highest sequence number it has passed on so far in seq_i;

initially this value is 0. It ignores incoming tokens that do not have a higher sequence number than seq_i. The last node in the ring, so initially node $N-1$, must increase the sequence number every time it forwards the token. If the last node in the ring crashes, this task is taken over by its predecessor i. This can be determined by means of a simple check: $next_i < i$.

As in the failure-sensitive version of Safra's algorithm, $black_t$ and $black_i$ express which nodes should be considered black. Again, when the token is sent by the initiator for the first time, $black_t = N-1$, to guarantee that it visits all nodes. As before, if a node receives a basic message, it colors all nodes in the ring from itself up to the sender black. Furthermore, if the failure detector of a node i reports a crashed node and at the next token visit i does not detect termination, then i colors all other nodes black, as they must all take into account this crash before termination can be safely detected. A fine point is that if $black_i \neq i$ and $next_i$ lies beyond $black_i$, then $black_i$ is assigned the value of $next_i$. Namely, then i considers only itself black, as all other nodes it considered black have crashed.

Procedure 5: Initialization$_i$

```
1 for j = 0 to N − 1 do
2     count_i^j ← 0;    count_t^j ← 0;
3 black_i ← i;    seq_i ← 0;
4 CRASHED_i ← ∅;    CRASHED_t ← ∅;    REPORT_i ← ∅;
5 next_i ← (i + 1) mod N;
6 if i = 0 then
7     black_t ← N − 1;    seq_t ← 1;    HandleToken_0;
8 else
9     black_t ← i;
```

Procedure 6: SendBasicMessage$_i(m, j)$

```
1 if j ∉ CRASHED_i ∪ REPORT_i ∪ CRASHED_t then
2     seq_m ← seq_i;    sender_m ← i;
3     send(m, j);    count_i^j ← count_i^j + 1;
```

Procedure 7: ReceivedBasicMessage$_i$

```
1 m ← dequeue(MESSAGEQUEUE_i);
2 if sender_m ∉ CRASHED_i then
3     if (sender_m < i ∧ seq_m = seq_i + 1) ∨
        (sender_m > i ∧ seq_m = seq_i) then
4         black_i ← furthest_i(black_i, sender_m);
5     count_i^{sender_m} ← count_i^{sender_m} − 1;
```

The pseudocode of the procedures available at each node is given in Procedures 5-11. Again they are supposed to be executed without interruption, except that while waiting to become passive, in line 1 of procedure HandleToken, a node is allowed to perform SendBasicMessage, ReceiveBasicMessage and FailureDetector calls. We note that in the initialization phase all nodes initialize local token variables.

The initialization phase is similar in spirit as in the failure-sensitive case. Sending or receiving a basic message is also as before: the sender or receiver i of the message updates its counter associated with the receiver or sender, respectively. Only, this receiver or sender should be alive according to i. Basic messages received from a crashed node in REPORT$_i$ may still be accounted for by i in the control algorithm, to allow for termination detection at the next token visit to i.

Procedure ReceivedToken$_i$ (RT$_i$) is executed when a token arrives at node i. It only proceeds if i did not receive a token from a later round before (line 1). Then i updates CRASHED$_t$, which is denoted by *updateCrash*(t).* Next procedure HandleToken$_i$ (HT$_i$) is called. There i must wait until it becomes passive (*wait*(*passive*$_i$) in line 1), because in the meantime the values of $count_i^j$, $black_i$ and REPORT$_i$ may still change. Once passive, i updates its local token variables (*update*(t) in line 1). Then, the set CRASHED$_t$ is relieved of the nodes that i reported through the token before (line 2). The remaining nodes in CRASHED$_t$ are copied to CRASHED$_i$ (line 3), because they will be reported when i forwards t. $black_i$ is set to the furthest of $black_i$ and $black_t$ (line 4), and REPORT$_i$ is relieved of nodes in CRASHED$_t$ (line 5). The sum of the values $count_i^j$ for nodes $j \notin$ CRASHED$_i \cup \{i\}$ is assigned to $count_t^i$ (lines 7-9); but only if i is white or REPORT$_i$ is empty (line 6), because then it may be employed in termination detection at i or at other nodes, respectively. If i is white (line 10), the values $count_t^j$ for nodes $j \notin$ CRASHED$_i$ are added up in an auxiliary field sum_i (lines 11-13); if this sum is 0, i announces termination (lines 14-15). If no termination is detected, then i checks whether its successor is among the newly reported crashed nodes in t; if so, NewSuccessor$_i$ is called to select another successor (lines 16-17). Next i checks whether it is the last node in the ring, and if so increases the sequence number of t by 1 (lines 18-19). If REPORT$_i$ is nonempty (line 20), then it is added to CRASHED$_t$ (line 21), so that t will report these crashed nodes to all alive nodes; next all nodes in REPORT$_i$ are moved to CRASHED$_i$ (lines 22-23); and $black_t$ is set to i (line 23), to ensure that the token completes another round trip before termination can be detected, as all alive nodes must first achieve a consistent view on the set of crashed nodes. Else, if $black_t = i$ then i changes the value of this field to $next_i$ (lines 24-25). Finally i forwards t to $next_i$, colors itself white, and increases seq_i by 1 (line 26).

Procedure 8: ReceivedToken$_i$

```
1 if seq_t = seq_i + 1 then
2     updateCrash(t);    HandleToken_i;
```

The procedure FailureDetector$_i$ (FD$_i$) is invoked if the failure detector reports a crashed node j, represented by the action *crashed*(j) (line 1). If i was not yet aware of this crash (line 2), then j is added to REPORT$_i$ (line 3), so that this crash will be reported to the other nodes via the token. If j happens to be the successor of i in the ring, then the procedure NewSuccessor$_i$ is invoked to compute the new successor of i (lines 4-5). Next a backup token is sent to this new successor (line 11), if i received the token at least once (first disjunct in line 6); the second disjunct in line 6 makes sure a backup token is also sent in case the initiator crashes before ever becoming

*For developers: it is important to update only CRASHED$_t$ and no other variables related to crashed nodes. That would lead to subtle bugs if a node receives a backup token while handling the token.

Procedure 9: HandleToken $_i$

```
1  wait(passive_i);      update(t);
2  CRASHED_t ← CRASHED_t \ CRASHED_i;
3  CRASHED_i ← CRASHED_i ∪ CRASHED_t;
4  black_i ← furthest_i(black_i, black_t);
5  REPORT_i ← REPORT_i \ CRASHED_t;
6  if black_i = i ∨ REPORT_i = ∅ then
7      count_t^i ← 0;
8      for all j ∈ {0,...,N−1} \ (CRASHED_i ∪ {i}) do
9          count_t^i ← count_t^i + count_i^j;
10 if black_i = i then
11     sum_i ← 0;
12     for all j ∈ {0,...,N−1} \ CRASHED_i do
13         sum_i ← sum_i + count_t^j;
14     if sum_i = 0 then
15         Announce;
16 if next_i ∈ CRASHED_t then
17     NewSuccessor_i;
18 if next_i < i then
19     seq_t ← seq_t + 1;
20 if REPORT_i ≠ ∅ then
21     CRASHED_t ← CRASHED_t ∪ REPORT_i;
22     CRASHED_i ← CRASHED_i ∪ REPORT_i;
23     REPORT_i ← ∅;      black_t ← i;
24 else
25     black_t ← furthest_i(black_i, next_i);
26 send(t, next_i);      black_i ← i;      seq_i ← seq_i + 1;
```

Procedure 10: FailureDetector $_i$

```
1  crashed(j);
2  if j ∉ CRASHED_i ∪ REPORT_i then
3      REPORT_i ← REPORT_i ∪ {j};
4      if j = next_i then
5          NewSuccessor_i;
6          if seq_i > 0 ∨ next_i < i then
7              CRASHED_t ← CRASHED_t ∪ REPORT_i;
8              black_t ← i;
9              if next_i < i then
10                 seq_t ← seq_i + 1;
11             send(t, next_i);
```

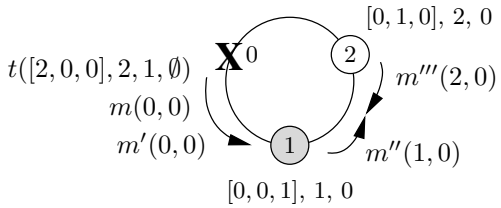
Procedure 11: NewSuccessor $_i$

```
1  next_i ← (next_i + 1) mod N;
2  while next_i ∈ CRASHED_i ∪ REPORT_i do
3      next_i ← (next_i + 1) mod N;
4  if next_i = i then
5      wait(passive_i);      Announce;
6  if black_i ≠ i then
7      black_i ← furthest_i(black_i, next_i);
```

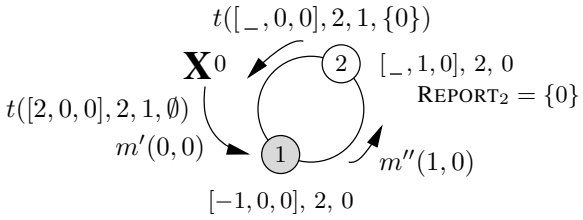
passive. REPORT_i is added to CRASHED_t (line 7); the nodes in REPORT_i are not transposed to CRASHED_i yet, because the backup token may be discarded in favor of the original token. By assigning $black_t$ the value i (in line 8), it is guaranteed that the backup token must complete an entire round trip before it can lead to termination detection, as first all alive nodes must take into account that node j has crashed. If no alive node in the ring has an identity greater than i, then the sequence number of the token is increased by 1 (lines 9-10).

The procedure NewSuccessor_i (NS_i) computes the new successor of i after $next_i$ has crashed. For a start, $next_i$ is changed into $(next_i + 1) \bmod N$ (line 1). Repeatedly it is checked whether the new value of $next_i$ is a crashed node (line 2), and if so its value is increased by 1, modulo N (line 3). After the value of $next_i$ has stabilized, i checks whether it is the only remaining alive node in the network (line 4), and if so waits until it has become passive to announce termination (lines 5-6). Else it is made sure that $black_i$ is not the identity of a (crashed) node between i and $next_i$ (lines 7-8).

Example 2. We consider one possible run of the fault-tolerant version of Safra's algorithm on a ring of three nodes. Initially the three nodes are active, all counters carry the value 0, and $black_i = i$ and $seq_i = 0$ for $i = 0, 1, 2$. Node 0 sends basic messages m and m' to node 1, node 1 sends basic message m'' to node 2, and node 2 sends basic message m''' to node 1 (all with their node identity and sequence number 0 attached); $count_0^1$ is set to 2, and $count_1^2$ and $count_2^1$ are set to 1. Nodes 0 and 2 now become passive. Node 0 sends the token to node 1 (with $count_t^0 = 2$, $count_t^1 = count_t^2 = 0$, $black_t = 2$, $seq_t = 1$ and $\text{CRASHED}_t = \emptyset$), and crashes. This leads to the following picture, where the cross at node 0 represents that it has crashed, the sequences of *count* values at alive nodes are placed between square brackets, and empty CRASHED and REPORT sets at nodes have been omitted.

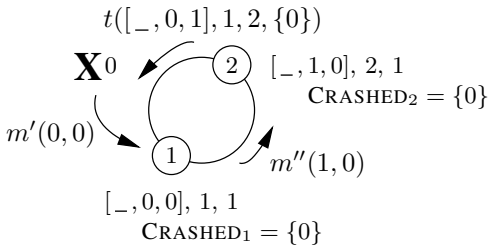


Node 2 detects node 0 crashed and sets REPORT_2 to $\{0\}$; from now on node 2 ignores $count_2^0$. Since $next_2 = 0$, node 2 makes node 1 its new successor. Since $1 < 2$, node 2 sends a backup token to node 1 (with $count_t^1 = count_t^2 = 0$, $black_t = 2$, $seq_t = 1$ and $\text{CRASHED}_t = \{0\}$). Node 1 receives m from node 0 and sets $count_1^0$ to -1; since $sender_m = 0 < 1$ and $seq_m = 0 = seq_1$, $black_1$ remains unchanged; moreover, node 1 receives m''' from node 2 and sets $count_1^2$ to 0; since $sender_{m'''} = 2 > 1$ and $seq_{m'''} = 0 = seq_1$, $black_1$ is set to 2.



Node 1 receives the backup token from node 2 and sets CRASHED_1 to $\{0\}$. It becomes passive, passes on the token to node 2 with $count_t^1$ set to $count_1^2 = 0$, and sets both $black_1$ and seq_1 to 1. Node 1 does not detect termination since $black_t = 2$. Next node 1 receives the original token from node 0, which is dismissed. Node 2 receives the token and sets CRASHED_2 to

$\{0\}$ and REPORT_2 to $\emptyset$. It does not detect termination because it sets $count_t^2$ to $count_2^1 = 1$, and $count_t^1 + count_t^2 = 0+1 > 0$. It passes on the token to node 1 with $black_t = 1$ and $seq_t = 2$, and sets $black_2$ to 2 and seq_2 to 1.



When the token arrives, node 1 sets CRASHED_t to $\emptyset$, computes $count_t^1 = 1$, passes on the token to node 2 with $black_t = 2$, and sets $black_1$ to 1 and seq_1 to 2. In the meantime node 2 receives m'' from node 1 and sets $count_2^1$ to 0; since $sender_{m''} = 1 < 2$ and $seq_{m''} = 0 < seq_2$, $black_2$ remains unchanged. Node 2 becomes passive again. When the token arrives, node 2 computes $count_t^1 + count_t^2 = 0 + 0 = 0$. Since also $black_t = 2$, it announces termination. Finally node 1 ignores message m' from node 0, because $0 \in \text{CRASHED}_1$.

Theorem 2. The fault-tolerant version of Safra's termination detection algorithm is correct.

Proof: We argue that the following two properties hold. *Liveness*: If the basic algorithm has terminated and not all nodes in the network crash, then eventually termination will be announced by some node. *Safety*: If termination is announced, then (1) all alive nodes are passive, and (2) for all basic messages in the channels, the receiver either has crashed or knows that the sender has crashed. We explain how the line of reasoning in the proof of Theorem 1 needs to be extended and adapted for the fault-tolerant version of Safra's algorithm.

Since failure detectors are perfect, a node only ends up in a CRASHED set if it has really crashed. Counting basic messages between alive nodes must be performed in a consistent way at all alive nodes, so they must have a uniform view on their CRASHED sets. Therefore, when node i reports newly detected crashed nodes in the token (line 21 of HT_i), it makes sure the token must complete a round trip before termination can be detected (by $black_i \leftarrow i$ in line 23 of HT_i and line 4 of HT_j at other nodes j). Furthermore, if i issues a backup token (line 11 of FD_i) after its successor crashed, then it makes sure the token will complete a round trip (by line 8 of FD_i), so that this crash is accounted for at all alive nodes.

With regard to *Liveness* we observe that the token continuously visits all consecutive alive nodes in the ring, if there are at least two of those, since NS_i is called when the successor of node i in the ring is found to have crashed (lines 16-17 of HT_i and 4-5 of FD_i). Moreover, owing to the backup mechanism (lines 6-11 of FD_i) the token is not lost at a crash; and by managing seq_t and seq_i (in lines 18-19 and 26 of HT_i and 9-10 of FD_i) we make sure that only a single token is being forwarded (by line 1 of RT_i). This means that the argumentation in Section IV-B why the improved version of Safra's algorithm leads to a call to Announce when the basic algorithm has terminated basically also applies here. The only real difference is that this argumentation is restricted to the nodes that are still alive, and that Announce is only called if at least one node stays alive.

We proceed with *Safety*. There are two places where a node i can call Announce. In case of line 5 of NS_i we have $next_i = i$ (by line 4 of NS_i), which implies that all other nodes have crashed. So when i becomes passive (in line 5 of NS_i), it can safely call Announce. In case of line 15 of HT_i we have $sum_i = 0$ (by line 14 of HT_i), meaning that the counters of the active nodes add up to 0 (by lines 7-9 and 11-13 of HT_i). With a similar line of reasoning as for the improved version of Safra's algorithm (at the end of Section IV-B) it can be argued that this implies the basic algorithm has terminated. □

It can be argued that when an execution terminates, this is always detected within N token steps.

VI. Implementation and experimental results

We evaluated our fault-tolerant algorithm (SafraFT), and compared it with its fault-sensitive counterpart (SafraFS), both on an emulator and by applying them to fault-tolerant versions of two well-known distributed algorithms. The experiments were conducted on the Vrije Universiteit Amsterdam cluster of DAS-4 [3]. The nodes are equipped with 2x Intel E5620 CPUs and a Linux CentOS built with kernel version 3.10.0-693.17.1.el7.x86_6. For communication 1Gbit/s Lan is used.

A. Emulation environment

We emulated the activity of a basic algorithm and the network behavior. Nodes are allocated a thread each to run uninterrupted. Active nodes choose up to three activities to perform. Activities emulate internal or send events of the basic algorithm. Internal events are emulated by causing the node-thread to sleep for some random amount of time. To send a basic message, a random node is chosen to receive it after some random bounded delay, emulating network latency. Forwarding the token corresponds to an invocation of the RT procedure of the receiver also after some random bounded delay. All message sending is handled by a separate thread. Another thread decides beforehand which nodes to crash and crashes them throughout the run with some probability.

We ran the emulator experiments on a single compute node of DAS-4. We used networks of 16, 48 and 144 nodes and two probability distributions (uniform and Gaussian) for the randomized choices. We emulated a decentralized basic algorithm, with half of the nodes initially active. For each version, network size and probability distribution we performed a test with no nodes crashing and a test for each 20% interval ($[1, 20]$, $[21, 40]$, etc.) of crashing nodes (SafraFT only). We repeated each test 1000 times, for a total of 42,000 runs. In all runs, termination was detected correctly. We moreover used the emulation results to reason about the performance of SafraFT and SafraFS. In practice workloads do not always follow a smooth probability distribution and the thread-scheduling policies are likely to introduce biases. (Using a

specific basic algorithm and hardware platform of course also comes with a certain bias). Still the synthetic results give an indication of potential overheads. Our implementation in Java can be found at https://github.com/gkarlos/FTSEmu.

B. Application to two distributed algorithms

We applied SafraFT to detect the termination of a fault-tolerant version of the Chandy-Misra routing algorithm [4] (CM) and the Afek-Kutten-Yung self-stabilizing spanning tree algorithm [1] (AKY). The implementation is in Java, on top of the Ibis distributed programming platform [2]. Multiple network nodes were run on each DAS-4 compute node to achieve decently sized networks (up to 2000 nodes). Before each run a certain percentage of nodes, of up to 90%, was randomly selected to crash after performing a certain number of send and receive events. The results presented here are a representative excerpt. For the full version as well as the implementation see https://github.com/PerFuchs/safra-termination-detection-fault-tolerant.

As an aside, the experiment unveiled a delicate implementation issue. It is vital that updates of variables in the ReceivedToken procedure are atomic with regard to the HandleToken procedure. Else incorrect behavior may occur if a node receives a backup token while handling the token.

C. Results

Emulation results in Figure 1 (left), show that SafraFT imposes no additional control message overhead, in the absence of crashes, compared to SafraFS. Both variants require the same number of tokens to detect termination after it has occurred (T_{post}), incurring, on average, less than one extra round of token-forwarding (R_{post}). Additionally, both variants require the same number of tokens before termination (T_{pre}). This is to be expected since in the absence of crashes SafraFT's operation is almost identical to that of SafraFS. The small variations in the plot are attributed to the randomized workloads of the emulator and we expect deterministic workloads to produce identical results. These results are stable across the two probability distributions used in the emulator.

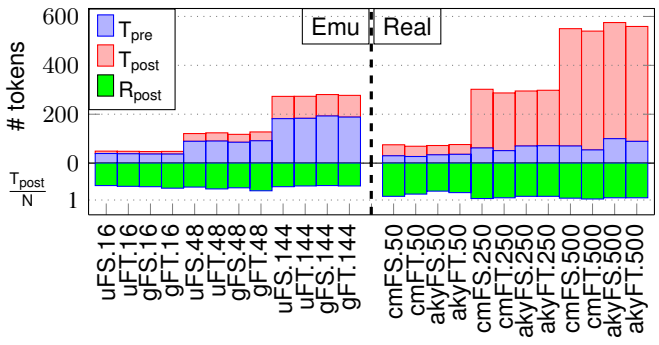


Fig. 1. **Top**: Tokens sent on crash-free networks. **Bottom**: Detection delay.

The CM/AKY results in Figure 1 (right) similarly show that SafraFT incurs no extra no control message overhead on fault-free runs compared to SafraFS. Compared to the emulation results we observe a (relative) increase in T_{post}, yet termination is still detected within one token round. Moreover there is a decrease in T_{pre}. We attribute this to the generally higher workload of the emulator. For instance, in CM, the initiator does not take part in the computation after the initial broadcast, and thus is mostly passive. Moreover nodes send estimate messages to their neighbors but not their parent. This leaves more nodes passive compared to the randomized activities of the emulator. A similar argument applies to AKY. This can also be seen by the fact that for CM/AKY, we observe $T_{pre} \ll N$ whereas for the emulation $T_{pre} \simeq 1.5 * N$.

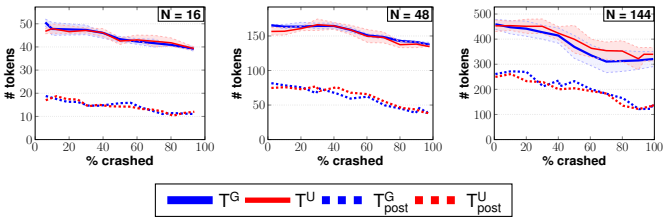


Fig. 2. Tokens sent by SafraFT on faulty ($\leq N-1$) networks in the emulator.

Figure 2 shows the effect of crashes on the emulation runs. We observe a decrease in the total number of tokens sent (T) as failures increase. Moreover the decrease in both T and T_{post} is roughly linear and the difference between 1 and $N-1$ nodes crashing tends to N. In a real-world scenario the basic algorithm will usually be aware of the crash and respond to it. This may cause (active) nodes to activate other nodes that in turn may render the token inconsistent and extend its trip. Moreover, the basic algorithm may re-assign the workload of the crashed node to some other alive node(s), causing higher activity overall. Finally, crashed nodes may have already forwarded the token some number of times before crashing. These reasons indicate an increase in the number of tokens as failures increase. Such an increase is observed for CM/AKY in Figure 3, although experiments with more diverse workloads are needed to confirm this hypothesis. Unlike CM/AKY, the emulated basic algorithm does not respond to crashes. T_{pre} remains roughly the same for small networks and actually decreases on the largest one. We attribute the latter to the fact that the overall activity produced by the emulator on 16 and 48 nodes is relatively small compared to that of 144 nodes.

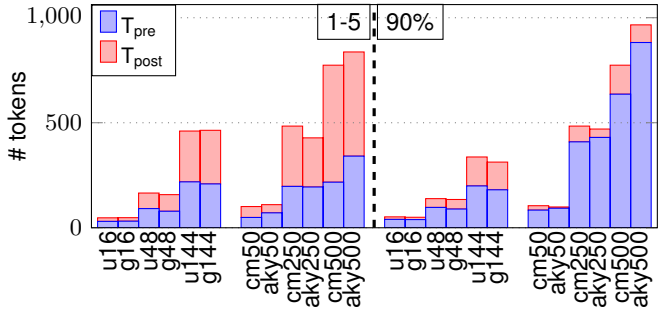


Fig. 3. Tokens sent by SafraFT with 1-5 (left) and 90% (right) crashed nodes.

Table I shows the processing time (pt), detection overhead (ov), and time to detect termination after it happened (tt), for SafraFS and SafraFT. For $N \leq 1000$, pt-FS and pt-FT grow roughly linearly to N. An analysis of processing times shows

TABLE I
CM/AKY TIMING RESULTS OF SAFRAFS/FT ON CRASH-FREE NETWORKS AND SAFRAFT ON FAULTY NETWORKS. TIMES ARE IN SECONDS

	Crash-Free Chandy-Misra			Crash-Free Afek-Kutten-Yung			Faulty Chandy-Misra				Faulty Afek-Kutten-Yung			
N	**pt-FS/FT**	**ov-FS/FT**	**tt-FS/FT**	**pt-FS/FT**	**ov-FS/FT**	**tt-FS/FT**	**pt-1/5**	**pt-90**	**tt-1/5**	**tt-90**	**pt-1/5**	**pt-90**	**tt-1/5**	**tt-90**
50	0.02/**0.03**	4.8/7.4%	0.01/**0.01**	0.04/**0.05**	4.1/ 5.7%	0.01/**0.01**	0.06	0.11	0.02	0.01	0.07	0.11	0.01	0.002
250	0.12/**0.24**	3.7/7.1%	0.03/**0.10**	0.15/**0.26**	6.1/10.5%	0.02/**0.08**	0.29	0.52	0.10	0.04	0.29	0.60	0.07	0.01
500	0.26/**0.64**	2.4/5.7%	0.06/**0.34**	0.30/**0.52**	6.4/11.1%	0.05/**0.18**	0.71	1.18	0.30	0.08	0.65	1.48	0.19	0.03
1000	0.59/**1.15**	2.2/4.3%	0.12/**0.54**	0.68/**1.46**	4.9/10.6%	0.11/**0.70**	1.59	2.43	0.70	0.11	1.91	4.40	0.42	0.10
2000	1.43/**2.66**	1.2/2.2%	0.23/**1.05**	1.63/**6.03**	3.4/12.7%	0.30/**3.85**	5.11	6.88	2.22	0.31	4.25	9.93	1.70	0.17

that the main factor in the higher time consumption of SafraFT, compared to SafraFS, is the growth in token size when N increases. The outlier $N = 2000$, for Safra-FS but especially SafraFT, turned out to be caused by our experimental setup. Each physical compute node may simultaneously host up to 100 network nodes, depending on N. Each of these instances uses multiple threads. Altogether there are at least four times as many threads as cores on each machine. This leads to threads being preempted by the operating system, which happens more often for threads that try to send large messages (and in SafraFT, the token size grows linearly with N).

A relatively large part of the overall processing time is spent on detecting termination. This is because Chandy-Misra and Afek-Kutten-Yung complete their tasks quickly. For basic algorithms that take multiple hours to complete, the seconds taken for termination detection are negligible. The processing time overhead of termination detection (between 2.2% and 12.7% in all runs for SafraFT) would reduce significantly for long-running jobs, owing to the fact that Safra's algorithm imposes only little control message overhead, unlike termination detection algorithms for which every basic message needs to be acknowledged (e.g. [9]). Remarkably, for Chandy-Misra the overhead of SafraFT decreases when N grows, while for Afek-Kutten-Yung it increases. The reason is that the number of times nodes become passive grows significantly slower, in terms of N, for Chandy-Misra than for Afek-Kutten-Yung.

Table I (right) shows the processing time (pt) and time to detect termination after it happened (tt), in seconds, when 1 to 5 nodes (1/5) and when 90% of the nodes (90%) crash. The increase in processing time when more nodes crash is due to the fact that more backup tokens are sent. On the other hand, the time to detect termination decreases when more nodes crash, because there are fewer alive nodes.

VII. CONCLUSION

We presented a fault-tolerant algorithm for distributed termination detection based on an improved version of Safra's token ring algorithm. In Safra's failure-sensitive version, a counter in the token contains the number of basic messages sent minus the number of basic messages received. This counter is updated at every node visit. In our fault-tolerant variant this counter is split into counters per node, so that counts from crashed nodes can be discarded. If a node crashes, the ring structure is restored locally and a backup token is sent.

Experiments indicate that our algorithm imposes no significant control message overhead compared to its failure-sensitive counterpart. Despite the $O(N)$ bit complexity of the token, the available throughput and low latency of current network technologies, as well as the low message complexity of our algorithm, may render our approach feasible for large networks. This needs to be validated in experiments with real-life distributed networks under realistic and diverse workloads. Future work may investigate the use of stable storage. In that case the memory overhead of splitting the counter in the token can be avoided, but at the price of communication overhead as nodes will have to update or consult the stable storage.

Acknowledgment Ceriel Jacobs provided useful feedback on the design, implementation and analysis of our algorithm.

REFERENCES

[1] Y. Afek, S. Kutten and M. Yung, *The local detection paradigm and its applications to self-stabilization*, Theor. Comput. Sci. 186(1-2), 1997
[2] H.E. Bal et al., *Real-world distributed computer with Ibis*, IEEE Computer 43(8):54–62, 2010
[3] H.E. Bal et al., *A medium-scale distributed system for computer science research: Infrastructure for the long term*, IEEE Computer 49(5), 2016
[4] K.M. Chandy and J. Misra, *Distributed computation on graphs: Shortest path algorithms*, Communications of the ACM 25(11):833–837, 1982
[5] P.K. Dash and R.C. Hansdah, *A fault-tolerant distributed algorithm for termination detection using roughly synchronized clocks*, in Proc. ICPADS, pp. 736-743, IEEE, 1997
[6] M. Demirbas and A. Arora, *An optimal termination detection algorithm for rings*, Technical report, The Ohio State University, 2000
[7] E.W. Dijkstra, *Shmuel Safra's version of termination detection*. EWD Manuscript 998, The University of Texas at Austin, 1987
[8] E.W. Dijkstra, W.H.J. Feijen and A.J.M. van Gasteren, *Derivation of a termination detection algorithm for distributed computations*, Information Processing Letters 16:217-219, 1983
[9] E.W. Dijkstra and C.S. Scholten, *Termination detection for diffusing computations*, Information Processing Letters 11(1):1-4, 1980
[10] W.H.J. Feijen and A.J.M. van Gasteren, *Shmuel Safra's termination detection algorithm*, On a Method of Multiprogramming, Monographs in Computer Science, pp. 313-332, Springer, 1999
[11] W.J. Fokkink, *Distributed Algorithms: An Intuitive Approach* (2nd edition), MIT Press, 2018
[12] N. Francez, *Distributed termination*, ACM Transactions on Programming Languages and Systems 2:42-55, 1980
[13] J. Hursey and R.L. Graham, *Building a fault tolerant MPI application: A ring communication example*, in Proc. IPDPS Workshop on High Performance Computing, pp. 1549-1556, IEEE, 2011
[14] T.-H. Lai and L.-F. Wu, *An (N-1)-resilient algorithm for distributed termination detection*, IEEE Trans. Parallel Distrib. Syst. 6(1), 1995
[15] J. Lifflander, P. Miller and L. Kale, *Adoption protocols for fanout-optimal fault-tolerant termination detection*, in Proc. PPoPP, ACM, 2013
[16] J. Matocha and T. Camp, *A taxonomy of distributed termination detection algorithms*, Journal of Systems and Software 43(3):207-221, 1998
[17] J. Matocha, *Distributed termination detection in a mobile wireless network*, in Proc. Southeast Regional Conference, ACM, 1998
[18] F. Mattern, *Global quiescence detection based on credit distribution and recovery*, Information Processing Letters 30(4):195-200, 1989
[19] N. Mittal, F. Freiling, S. Venkatesan and L. Penso, *On termination detection in crash-prone distributed systems with failure detectors*, Journal of Parallel and Distributed Computing 68(6):855-875, 2008
[20] G.-C. Roman and J. Payton, *A termination detection protocol for use in mobile ad hoc networks*, Autom. Softw. Eng. 12(1):81–99, 2005
[21] G. Tel, *Introduction to Distributed Algorithms* (2nd ed.), CUP, 2000
[22] T.C. Tseng, *Detecting termination by weight-throwing in a faulty distributed system*, J. Parallel Distr. Com. 25(1):7-15, 1995
[23] S. Venkatesan, *Reliable protocols for distributed termination detection*, IEEE Transactions on Reliability 38(1):103-110, 1989