Fault-Tolerant Termination Detection with Safra's Algorithm

Karlos, Georgios; Fokkink, Wan; Fuchs, Per

published in
Networked Systems
2021

DOI (link to publisher)
10.1007/978-3-030-91014-3_5

document version
Publisher's PDF, also known as Version of record

Fault-Tolerant Termination Detection with Safra's Algorithm

Georgios Karlos[1(✉)], Wan Fokkink[1], and Per Fuchs[2]

[1] Vrije Universiteit Amsterdam, Amsterdam, The Netherlands
{g.karlos,w.j.fokkink}@vu.nl
[2] Technische Universität München, München, Germany
per.fuchs@cs.tum.edu

Abstract. Safra's distributed termination detection algorithm employs a logical token ring structure within a distributed 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 adapt this classic algorithm to make it fault-tolerant. The counter is split into counters per node, to discard counts from crashed nodes. If a node crashes, the token ring 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, tolerates any number of crashes as well as simultaneous crashes, and copes with crashes in a decentralized fashion. Experiments with an implementation of our algorithm were performed on top of two fault-tolerant distributed algorithms.

1 Introduction

Termination detection is a fundamental problem in distributed systems which was introduced independently in [9] and [12]. Termination can be announced when all nodes in the network have become passive and no messages are in transit. Distributed termination detection is applied in e.g. workpools, routing, diffusing computations, self-stabilization, and checking stable system properties such as deadlock and garbage in memory. Many (mostly failure-sensitive) termination detection algorithms have been proposed in the literature, see [17,18].

In Safra's algorithm [7,10] a token repeatedly visits all nodes in the network via a predetermined logical ring structure; 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 count in the token may be unreliable, if the message overtook 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.

Safra's algorithm imposes only little message overhead when nodes remain active over a long period of time, unlike termination detection algorithms for which every message needs to be acknowledged (e.g. [9]). Additionally, it does not require idle messages to be sent out when nodes become passive and does not

K. Echihabi and R. Meyer (Eds.): NETYS 2021, LNCS 12754, pp. 71–87, 2021.
https://doi.org/10.1007/978-3-030-91014-3_5

run into underflow issues, as opposed to weight-throwing schemes ([19,22]). 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 propose a fault-tolerant algorithm based on an improved version of Safra's algorithm [6]. A node crash is handled locally by its predecessor in the ring; a new token is issued, as the old token may have been lost in the crash. A numbering scheme in the token makes sure 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 counted. 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 have a failure detector and inform each other of crashes through the token, so that they uniformly count the message exchange with the same set of nodes. A node reporting a new crash makes sure the token completes another round trip, to avoid inconsistent message counts in the token.

Next to the aforementioned strong points of Safra's algorithm, our fault-tolerant variant has some additional advantages compared to existing fault-tolerant termination detection algorithms, which will be discussed in Sect. 2. (Following [11,21], we call the distributed algorithm for which termination is checked basic and the termination detection algorithm the control algorithm.) First, the basic algorithm can be decentralized, meaning there can be multiple initiator nodes. Second, if the initiator of our termination detection algorithm crashes before sending out the first token, this role is automatically taken over by its predecessor. Thus our algorithm can cope with any number of node crashes, and it is robust against simultaneous node crashes. Third, only one additional message is required for each crash, and no relatively expensive schemes like leader election or taking a global snapshot are employed. 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 tested our algorithm in a multi-threaded emulation environment and performed experiments on two fault-tolerant distributed algorithms from the literature. Compared with the failure-sensitive version of Safra's algorithm, our algorithm exhibits a satisfactory performance, in the sense that it imposes no additional message overhead. Of course it does impose some overhead, by adding extra concurrency at each node and additional synchronization. However, even with a large number of failures, profiling of our experiments shows that the execution time of the basic algorithm remains the dominant factor for the overall performance. The two basic distributed algorithms employed in our experiments are quite different in nature, which suggests this conclusion holds more generally, admittedly only up to the network sizes we analyzed.

Developing our algorithm was a delicate matter. Still, owing to a correctness proof (omitted here) in combination with many test runs with an implementation, we can confidently claim that our algorithm correctly detects termination.

2 Related Work

We discuss some existing fault-tolerant termination detection algorithms, mainly from a functional point of view. Only [16] reports on performance results based on an actual implementation. Generally a complete network topology and a perfect failure detector are required, as such assumptions are essential for developing a fault-tolerant termination detection algorithm, see [20].

Lai and Wu [15] presented a fault-tolerant variant of the Dijkstra-Scholten algorithm [9] for centralized basic algorithms, meaning there is a single initiator node. Active nodes are in the tree, rooted in the initiator, which announces termination when the tree has disappeared. In the event of a crash, all alive nodes communicate with the designated root node, causing a sequential bottleneck.

Lifflander et al. [16] proposed a series of algorithms based on [9] that avoid the bottleneck of [15]. These algorithms are resistant to single-node failures but are only probabilistically tolerant to multi-node failures and incur additional control messages even in crash-free executions. In case of a crash the tree is reconstructed locally. If two nodes fail concurrently, the algorithms may not be able to recover. The algorithm then detects that this is the case. Failure of the root node cannot be handled. Performance results are reported based on an experimental setup, consisting of three mock-up parallel algorithm implementations. Results show acceptable processing time overhead. Message overhead results are not reported.

Tseng [22] developed a fault-tolerant variant of weight-throwing [19] for centralized basic algorithms. Nodes donate part of their weight to the basic messages they send. The receiver claims that weight on receipt. A node that becomes passive returns its weight to the leader, who announces termination when it is passive and reclaimed its original weight. The number of control messages increases linearly with the number of basic messages. The algorithm is vulnerable to underflow of weight values and control messages require space to represent floats at high precision. A global snapshot is taken when a new crashed node is detected. When the leader crashes, an election scheme is employed.

In Venkatesan's algorithm [23] a leader node is in charge of announcing termination. If the leader crashes, an election is held. The local stacks at the nodes must be continuously replicated by the leader and its backup Upon learning of a crashed node, the leader simulates the state of every node in the system to determine whether it has terminated.

Hursey and Graham [14] developed a termination detection scheme for their fault-tolerant ring-based MPI application. Their algorithm relies on a leader election scheme and fault-tolerant primitives provided by MPI.

Mittal et al. [20] introduced a general framework for transforming any failure-sensitive termination detection algorithm into a fault-tolerant variant that can cope with 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 have a significant overhead in control messages, even when no nodes become passive or crash.

3 System Model

We assume a fully asynchronous message-passing system with no shared memory or global clock. Messages may arrive in any order and delays are unbounded but finite. The N nodes are *logically* organized in a ring and are assigned unique, totally-ordered IDs. Failures are permanent; once a node crashes, it halts and never recovers. Like most fault-tolerant distributed algorithms in the literature, we require that the underlying physical network provides reliable bidirectional communication channels between each pair of nodes.

Nodes are either *active* or *passive*. An active node can send/receive basic messages, perform internal events, or become passive when it *terminates locally*. A passive node cannot send basic messages or perform internal events and only becomes active upon receipt of a basic message. Termination may be announced while basic messages from crashed nodes are in transit. It is therefore required that passive nodes never become active by the receipt of a basic message sent by 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.

A *perfect* failure detector [4] is required to solve termination detection in the presence of failures [13,20]. Such a failure detector, which never falsely suspects that a node crashed, and eventually detects each node crash, can be built if there is a known upper bound on network latency.

4 Safra's Algorithm

Safra's (failure-sensitive) termination detection algorithm [7,10] generalizes the Dijkstra-Feijen-van Gasteren algorithm [8] from synchronous to asynchronous message passing networks. We give a detailed description of Safra's algorithm, including improvements from [6]. This will serve as a basis for the description of our fault-tolerant version later on.

Safra's algorithm is centralized, with node 0 as initiator. The basic algorithm however is allowed to be decentralized and does not need to be ring-based. A token t circulates the ring, 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 round trip of 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. Each time t is received by a node i, $count_i$ is added to $count_t$, and $count_i$ is reset to zero. Upon return of t to the initiator, after it has become passive, termination is detected if $count_t$ is zero.

The token can during its round trip underestimate the number of basic messages in transit, if the receipt of a message is accounted for in the token before

the send of this message. To recognize this, colors black and white are used. Initially all nodes are white, and when the initiator sends out a fresh token, the token is white. When a node i receives a basic message m, it may be that the send of m was not yet recorded in $count_t$. Therefore upon receipt of m, i marks itself black. When t visits a black node, t becomes black and the 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 is black or $count_t$ is not zero, the initiator sends out a fresh white token again. Otherwise, it can safely announce that the execution of the basic algorithm has terminated.

Two enhancements of Safra's algorithm to reduce detection delay were given in [6]. One inefficiency is that a basic message always blackens its receiver. Actually an inconsistent snapshot can only exist when the receipt of a message is recorded before its send. This can happen when a basic message overtakes the token, meaning that it is sent after the sender was visited by the token, but reaches the receiver before it is visited by the token. A second inefficiency of the original algorithm is that termination is only detected at the initiator. Another enhancement allows detection to occur at any node. When multiple nodes can detect termination, there is a second situation in which an inconsistent snapshot can occur. When both the sender and the receiver of a message are ahead of the token, but the receiver will be visited by the token before the sender, it is possible for the receiver to detect termination before the sender is visited. (This case is omitted in [6], which may result in erroneous detection.)

To deal with the aforementioned scenarios, a sequence number seq_i is introduced at every node i, starting at zero. When a node forwards the token, it increases its sequence number by one, so that nodes in the visited region have a higher sequence number from those in the unvisited region. A node piggybacks its sequence number seq_m to every basic message m it sends. Using the sequence number, an offending message can be detected if it has a higher sequence number than the receiver, or they both have the same sequence number but the sender has a higher ID than the receiver. Since multiple nodes can detect termination, an offending message should not only blacken the receiver but also all subsequent nodes in the ring up to (but not including) the sender. At all these nodes, the token represents an inconsistent snapshot. So none should detect termination.

The field $black_t$ in the token is now a node ID, 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, $black_t = N - 1$, 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 at least once. 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 of which the send may not have been accounted for in the token, then $black_i$ is set to the furthest node from i among $black_i$ and the sender of 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.

If the token reaches a node i, it must wait until i is passive. Then i adds the value of $count_i$ to the value of $count_t$. If i is white, meaning that $black_t = black_i = i$, it can determine termination in the same way the initiator does in Safra's algorithm: check whether the value of $count_t$ is zero. 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 zero and $black_i$ to i and increases seq_i by one.

Algorithms 1–4 present the pseudocode of the four procedures available at each node i for the improved version of Safra's algorithm: initialization, sending/receiving a basic message m to/from a node j (SBM/RBM) , and receiving a token (RT). Subscript i of a procedure name represents the node where the procedure is performed. Action $send(m, j)$ denotes that message m is sent to node j, and the Boolean field $passive_i$ is *true* only when node i is passive. Procedures are executed without interruption, except that while waiting to become passive, in line 1 of RT, a node is allowed to perform SBM and RBM calls.

Algorithm 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;   send(t, 1);   count_0 ← 0;   seq_0 ← 1;
```

Algorithm 2: SendBasicMessage $_i\,(m, j)$

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

Algorithm 3: ReceiveBasicMessage $_i\,(m, j)$

```
1 if seq_m = seq_i + 1 ∨ (j > i ∧ seq_m = seq_i) then
2     black_i ← furthest_i(black_i, j);
3 count_i ← count_i − 1;
```

Algorithm 4: ReceiveToken $_i$

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

5 Fault-Tolerant Version

From now on we assume nodes may spontaneously and permanently crash. 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. [21]), because else a node failure may result in disconnected subnetworks. Actually, it suffices if at any time a channel can be established between any two alive nodes.

Mittal et al. [20] showed that a perfect failure detector is required to solve termination detection in the presence of failures. In a fully asynchronous setting, such a detector cannot be built. A practical compromise is to assume an upper bound on the network latency. Each node sends out heartbeat messages at regular time intervals. When a node i has not received a heartbeat from another node j within some time interval, then i permanently considers j as crashed.

Each node i stores the identities of crashed nodes in one of the sets $\textsc{Crashed}_i$ and $\textsc{Report}_i$. The latter contains the identities that i has not yet reported to the other alive nodes by means of the token; this will be explained below.

Since counts of messages to and from 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 zero.) 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 might 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 $\textsc{Crashed}_t$. When i forwards the token with $j \in \textsc{Crashed}_t$, it adds j to $\textsc{Crashed}_i$, to avoid that it announces the same crashed node multiple times.

Each node i keeps track of its successor $next_i$ in the ring; initially $(i+1)$ mod N. Each time i detects $next_i$ has crashed, the value of this field is changed into i's nearest alive successor. We must ensure that 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 $next_i'$, i forwards the token once again, to $next_i'$. For this purpose i stores the last token it forwarded. These local variables are updated as soon another token (with a higher sequence number) arrives.

In case $next_i$ forwarded the token before crashing, $next_i'$ will receive the same token twice. Therefore the token has a sequence number seq_t, which is increased

by one at each consecutive round trip of the token. In the first round $seq_t = 1$. Each node i keeps track of the highest sequence number it has passed on so far in seq_i (initially $seq_i = 0$), and ignores incoming tokens with $seq_t \leq seq_i$. The last node in the ring, initially $N - 1$, increases the sequence number every time it forwards the token. If it crashes, this task is taken over by its predecessor. A node i can determine whether it is the last node by checking if $next_i < i$.

As in the failure-sensitive variant of Safra's algorithm, $black_t$ and $black_i$ express which nodes are considered black, and when the token is sent by the initiator for the first time, $black_t = N - 1$ to guarantee it visits all nodes at least once. If a node receives an offending basic message, it colors all nodes in the ring between itself and the sender black. 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 be visited by the token.

The pseudocode of the procedures at each node is given in Algorithms 5–10. Again, the procedures should be executed without interruption, except that while waiting to become passive, in line 2 of procedure ReceiveToken, a node may perform SendBasicMessage, ReceiveBasicMessage and FailureDetector calls.

In the initialization phase, nodes provide their local variables with initial values; node 0 holds the token. At sending/receiving a basic message to/from a noncrashed process, the sender/receiver updates the corresponding counter. Basic messages received from a crashed node in $\textsc{Report}_i$ may still be accounted for by i in the control algorithm, to allow for termination detection at the next token visit to i. If the receipt of a message is accounted for in the token before its send, the receiver colors the nodes up to the sender black.

Algorithm 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;   next_i ← (i + 1) mod N;
4 Crashed_i ← ∅;  Crashed_t ← ∅;  Report_i ← ∅;
5 if i = 0 then
6     black_t ← N − 1;   seq_t ← 1;   ReceiveToken_0;
7 else
8     black_t ← i;
```

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

```
1 if j ∉ Crashed_i ∪ Report_i ∪ Crashed_t then
2     seq_m ← seq_i;   send(m, j);   count_i^j ← count_i^j + 1;
```

Algorithm 7: ReceiveBasicMessage$_i(m, j)$

```
1 if j ∉ Crashed_i then
2     if seq_m = seq_i + 1 ∨ (j > i ∧ seq_m = seq_i) then
3         black_i ← furthest_i(black_i, j);
4     count_i^j ← count_i^j − 1;
```

Algorithm 8: ReceiveToken$_i$

```
1  if seq_t = seq_i + 1 then
2      wait (passive_i);   black_i ← furthest_i(black_i, black_t);
3      CRASHED_t ← CRASHED_t \ CRASHED_i;
4      CRASHED_i ← CRASHED_i ∪ CRASHED_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 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;   black_t ← i;
22         CRASHED_i ← CRASHED_i ∪ REPORT_i;   REPORT_i ← ∅;
23     else
24         black_t ← furthest_i(black_i, next_i);
25     send(t, next_i);   black_i ← i;   seq_i ← seq_i + 1;
```

Procedure ReceiveToken$_i$ (RT$_i$) is executed when a token arrives at node i. It only proceeds if i did not receive an instance of this token before (line 1). It then waits until it becomes passive, because in the meantime the values of $count_i^j$, $black_i$ and REPORT$_i$ may still change.

Once passive, $black_i$ is set to the furthest of $black_i$ and $black_t$ (line 2). Then, the set CRASHED$_t$ is relieved of the nodes that i reported through the token before (line 3). The remaining nodes in CRASHED$_t$ are copied to CRASHED$_i$, because they will be reported when i forwards t (line 4). REPORT$_i$ is relieved of nodes in CRASHED$_t$ (line 5). The values $count_i^j$ for nodes $j \notin$ CRASHED$_i$ are accumulated in $count_t^i$ (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 (in lines 10–15) or at other nodes, respectively. If i is white (line 10), the values $count_t^j$ for nodes $j \notin$ CRASHED$_i$ are accumulated in sum_i (lines 11–13); if this sum is 0, i announces termination (lines 14–15). If no termination is detected, i checks whether its successor is in CRASHED$_t$; 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$_t$, so that t will report these crashed nodes to all alive nodes; $black_t$ is set to i, to ensure that the token visits

all nodes up to i again before termination can be detected, as all alive nodes must first achieve a consistent view on the set of crashed nodes (line 21). Next all nodes in REPORT_i are moved to CRASHED_i (line 22). If REPORT_i is empty, then $black_t$ is set to $furthest_i(black_i, next_i)$ (lines 23–24). Finally, i forwards t to $next_i$, colors itself white, and increases seq_i by one (line 25).

FailureDetector_i (FD_i) is invoked if i's failure detector reports that a node j crashed ($crashed(j)$ in 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 other nodes via the token. If j is the successor of i in the ring, NewSuccessor_i is invoked to compute a new successor of i (lines 4–5). A backup token (possibly updated compared to the original token) is sent to the new successor (line 11), if i received the token at least once (first disjunct in line 6); the second disjunct in line 6 ensures a backup token is sent when the initiator crashes before ever becoming passive. REPORT_i is added to CRASHED_t (line 7); 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 $black_t \leftarrow i$ (in line 8) it is guaranteed that the backup token visits all nodes up to i again before termination can be detected, as all alive nodes must take into account the crash of j. If no alive node has an identity greater than i, then seq_t is increased by one (lines 9–10).

Algorithm 9: FailureDetector_i

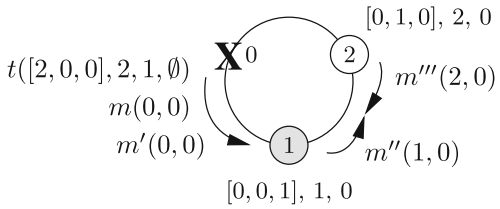
```
1  crashed(j);
2  if j ∉ CRASHED_i ∪ REPORT_i then
3      REPORT_i ← REPORT_i ∪ {j};
4      if j = next_i then
5          NewSuccessor_i;
6          if seq_i > 0 ∨ next_i < i then
7              CRASHED_t ← CRASHED_t ∪ REPORT_i;
8              black_t ← i;
9              if next_i < i then
10   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

Algorithm 10: NewSuccessor_i

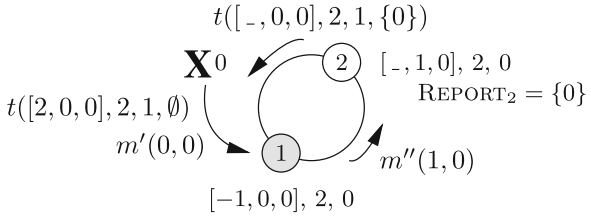
```
1  next_i ← (next_i + 1) mod N;
2  while next_i ∈ CRASHED_i ∪ REPORT_i do
3      next_i ← (next_i + 1) mod N;
4  if next_i = i then
5      wait(passive_i);
6      Announce;
7  if black_i ≠ i then
8      black_i ← furthest_i(black_i, next_i);
```

NewSuccessor_i (NS_i) computes i's new successor after $next_i$ crashed. First, $next_i$ is changed into $(next_i + 1) \bmod N$ (line 1). Then, it is repeatedly checked

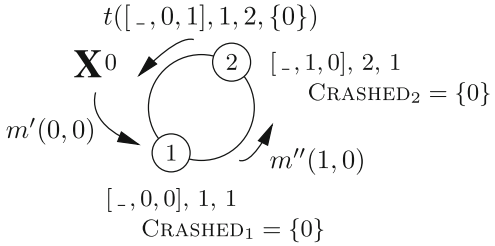
whether the new value of $next_i$ is a crashed node (line 2), and if so its value is increased by one, modulo N (line 3). After the value of $next_i$ has stabilized, i checks whether it is the only remaining alive node in the network (line 4), and if so, waits until it has become passive to announce termination (lines 5–6). Else, if $black_i \neq i$, then $black_i$ is set to $furthest_i(black_i, next_i)$ (lines 7–8).



(a) Node 0 sends two messages and forwards the token before crashing



(b) After the crash is detected, the two messages are discarded.



(c) Termination is detected in the next round

Fig. 1. Example run on a faulty network of three nodes

Example 1. We consider one possible run of our fault-tolerant algorithm on a ring of three nodes in Fig. 1. Initially all 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 ID 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 Fig. 1a 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.

In Fig. 1b, 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 the sender of m is $0 < 1$ 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 the sender of m''' is $2 > 1$ and $seq_{m'''} = 0 = seq_1$, $black_1$ is set to 2. Then, in Fig. 1c, 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 the sender of m'' is $1 < 2$ 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 $0 \in \text{CRASHED}_1$.

6 Implementation and Experimental Results

We applied a Java implementation of SafraFT to a fault-tolerant version of the Chandy-Misra routing algorithm [5] (CM) and the Afek-Kutten-Yung self-stabilizing spanning tree algorithm [1] (AKY)[1]. They form a good test bench because detecting termination is of importance for both algorithms while their messaging behaviors are distinct. Their implementations are on top of the Ibis distributed programming platform [2]. Experiments were conducted on the DAS-4 supercomputer [3]. Multiple network nodes were run on each DAS-4 compute node to achieve decently sized networks, up to 2000 network nodes. When more network nodes are placed on a single compute node, profiling shows this starts to influence the outcome of experiments. For this reason the experiments with the CM and AKY algorithms were limited to 2000 network nodes. Before each run a certain percentage of nodes, up to 90%, was randomly selected to crash after performing a certain number of events. As an aside, the experiment unveiled a delicate implementation issue. Updates of token variables in the ReceiveToken procedure must be atomic because otherwise incorrect behavior may occur if a node receives a backup token while handling the token.

[1] https://github.com/PerFuchs/safra-termination-detection-fault-tolerant.

Moreover, we abstractly emulated activity of a basic algorithm and network behavior under randomized execution scenarios[2]. The emulation experiments were performed 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, for SafraFT only, a test for each 20% interval ([1, 20], [21, 40], etc.) of crashing nodes. We repeated each test 1000 times, for a total of 42,000 runs (two probability distributions, three network sizes, and six intervals for SafraFT; one interval [0, 0] for SafraFS).

Emulation results in Fig. 2 (Emu, left) confirm that SafraFT imposes no additional control message overhead, in the absence of crashes, compared to SafraFS. Both variants tend to require the same number of token steps to detect termination after it has occurred (T_{post}), incurring on average half a round of extra token steps (R_{post}). They also require the same number of token steps before termination (T_{pre}). This is to be expected since in the absence of crashes, the operation of SafraFT is almost identical to that of SafraFS. These results are stable across the two probability distributions used in the emulator.

Performance results of emulations have to be taken with a grain of salt. In practice workloads do not always follow a smooth probability distribution, thread-scheduling policies as well as the hardware platform may to introduce biases, and basic algorithms may exhibit behavior to deal with actual node crashes. Still these synthetic results give some indication of the performance overhead SafraFT may impose, and importantly the large number of emulations helped to further increase confidence in the correctness of this algorithm.

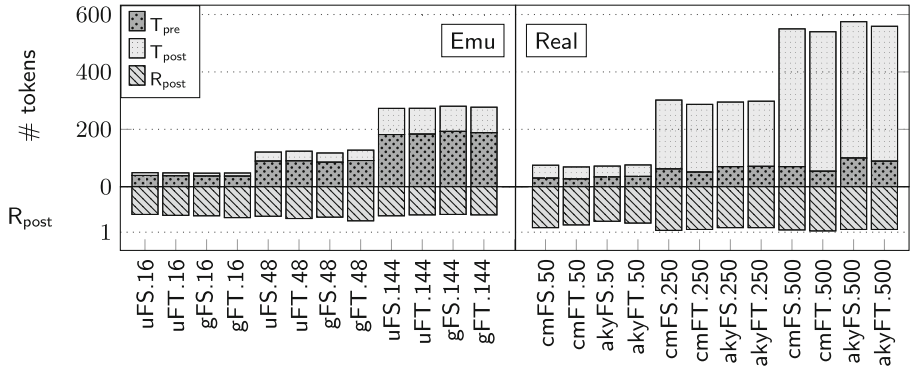


Fig. 2. Top: Tokens sent on crash-free networks. Uniform, Gaussian distributions denoted by **u**, **g**. **Bottom**: Detection delay, in token rounds after termination.

The CM/AKY results in Fig. 2 (Real, right) also show that SafraFT incurs no extra control message overhead on crash-free runs. Compared to the emulation results, there is a small increase in R_{post}. For emulation, T_{pre} is considerably larger than for CM/AKY. This difference can be attributed to the generally

[2] https://github.com/gkarlos/FTSEmu.

higher workload of the emulated basic algorithm compared to CM/AKY. For instance, the initiator of the CM algorithm does not take part in the computation after the initial broadcast, and thus is mostly passive. Moreover, in CM, nodes send estimate messages to their neighbors but not to their parent. This leaves more nodes passive compared to the randomized activities of the emulator.

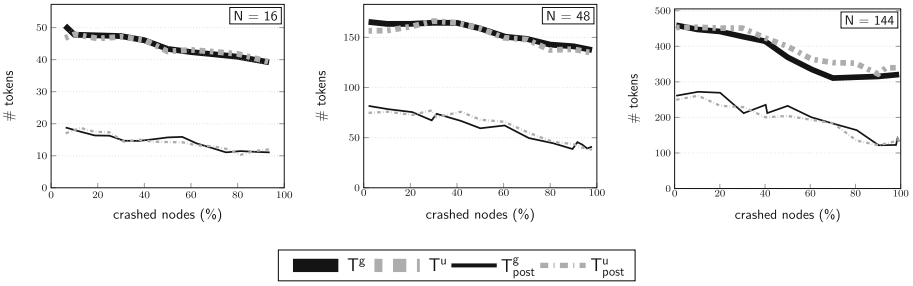


Fig. 3. Tokens sent by SafraFT on faulty networks in the emulator. Uniform, Gaussian distributions denoted by **u**, **g**.

Figure 3 shows the effect of crashes on emulation runs of SafraFT. The total number of token steps (T) and T_{post} decrease roughly in a linear fashion as failures increase, because when nodes crash, fewer nodes remain to forward the token, and our emulations of basic algorithms do not react to crashes.

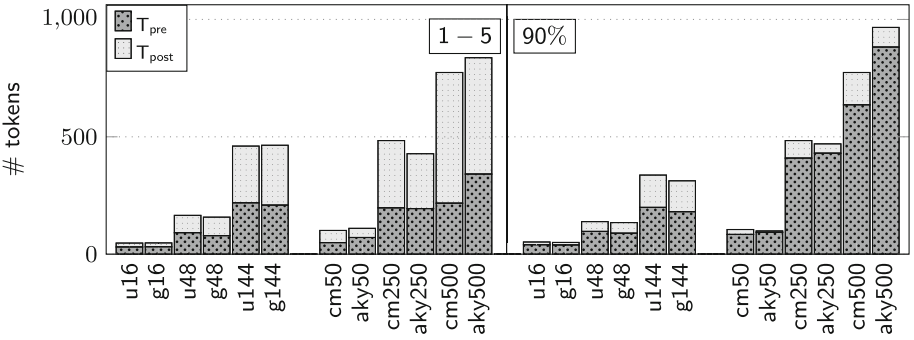


Fig. 4. Tokens sent by SafraFT with 1–5 and 90% crashed nodes in the emulator and CM/AKY experiments. Uniform, Gaussian distributions denoted by **u**, **g**.

Such a decrease is not to be expected for real-world distributed algorithms. A node crash may cause active nodes to activate other nodes, or the workload of the crashed node may be reassigned to other alive nodes, extending the trip of the token. A significant increase of T_{pre} is indeed observed for CM/AKY in Fig. 4 if many nodes (90%) crash, compared to if 1–5 nodes crash. By contrast, crashes have little effect on emulated basic algorithms, for reasons discussed before. T_{pre} remains roughly the same for small networks and actually shows a decrease on the largest one, due 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.

Table 1 shows the sum of processing times (pt), detection overhead (ov), and time to detect termination after it happened (tt), for SafraFS and SafraFT, when applied to CM and AKY. For $N \leq 1000$, pt-FS and pt-FT grow roughly linear to N. An analysis of processing times shows 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 SafraFS 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).

Table 1. CM/AKY results of SafraFS/FT on crash-free networks. Times in s.

N	Crash-Free CM			Crash-Free AKY		
	pt-FS/FT	ov-FS/FT	tt-FS/FT	pt-FS/FT	ov-FS/FT	tt-FS/FT
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**
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**
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**
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**
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**

A relatively large part of the overall processing time is spent on detecting termination because CM and AKY complete their tasks relatively quickly. For basic algorithms that take a long time to complete, the time taken for termination detection can be expected to be 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 tends to impose only little control message overhead, unlike termination detection algorithms in which every basic message is acknowledged (e.g. [9]). Remarkably, for CM the overhead of SafraFT decreases when N grows, while for AKY it increases. The reason is that the number of times nodes become passive grows significantly slower, in terms of N, for CM than for AKY.

Table 2. CM/AKY results of SafraFT on faulty networks. Times in s.

N	Faulty CM				Faulty AKY			
	pt-1-5	pt-90	tt-1-5	tt-90	pt-1-5	pt-90	tt-1-5	tt-90
50	0.06	0.11	0.02	0.01	0.07	0.11	0.01	0.002
250	0.29	0.52	0.10	0.04	0.29	0.60	0.07	0.01
500	0.71	1.18	0.30	0.08	0.65	1.48	0.19	0.03
1000	1.59	2.43	0.70	0.11	1.91	4.40	0.42	0.10
2000	5.11	6.88	2.22	0.31	4.25	9.93	1.70	0.17

Table 2 shows the sum of the processing times at all nodes and the time to detect termination after it happened for SafraFT, applied to CM and AKY, when 1 to 5 nodes (1–5) and when 90% of the nodes (90%) crash. When more nodes crash, the processing time increases, due to backup tokens, while the time to detect termination decreases, because there are fewer alive nodes.

7 Conclusion

We presented a fault-tolerant algorithm for distributed termination detection based on an improved version of Safra's algorithm. In our fault-tolerant variant message counters are maintained per node, so that counts to and from crashed nodes can be discarded. If a node crashes, the ring structure is restored locally and a backup token is sent. Strong points are: little message overhead when nodes remain active for a long time; robust against any number of and simultaneous node crashes; only one additional message per crash; the basic algorithm can be decentralized; no leader election scheme; no underflow issues. Compared to other algorithms, our algorithm generates far fewer, but larger control messages. For overall performance, fewer messages tend to be better, since more messages mean more processing at each node, as well as at the network stack.

Experiments indicate our algorithm imposes no significant extra overhead in control messages 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 on many machines.

Testing the behavior of fault-tolerant distributed algorithms on very large networks turned out to be challenging. Emulating basic algorithms by means of unrestricted randomization results in executions that refuse to terminate on large networks and do not faithfully mimic all aspects of real-life distributed algorithms. Moreover, in experiments on top of two actual algorithms, allocating multiple network nodes on a single compute node influences the results. These challenges may partly explain why [16] is the only related paper we are aware of to report experimental results, for networks of up to 2048 nodes.

Next to performing realistic experiments for larger networks, future work is to develop a version of our fault-tolerant algorithm in the presence of stable storage. In that case the memory overhead of splitting the counter in the token can be avoided, at the cost of storing message counts in stable storage.

Acknowledgement. Ceriel Jacobs provided valuable feedback on the design and implementation of our algorithm.

References

1. Afek, Y., Kutten, S., Yung, M.: The local detection paradigm and its applications to self-stabilization. Theoret. Comput. Sci. **186**(1–2), 199–229 (1997)

2. Bal, H.E., et al.: Real-world distributed computer with Ibis. IEEE Comput. **43**(8), 54–62 (2010)
3. Bal, H.E., et al.: A medium-scale distributed system for computer science research: infrastructure for the long term. IEEE Comput. **49**(5), 54–63 (2016)
4. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. J. ACM **43**(2), 225–267 (1996)
5. Chandy, K.M., Misra, J.: Distributed computation on graphs: shortest path algorithms. Commun. ACM **25**(11), 833–837 (1982)
6. Demirbas, M., Arora, A.: An optimal termination detection algorithm for rings, Technical report, The Ohio State University (2000)
7. Dijkstra, E.W.: Shmuel Safra's version of termination detection. EWD Manuscript 998, The University of Texas at Austin (1987)
8. Dijkstra, E.W., Feijen, W.H.J., van Gasteren, A.J.M.: Derivation of a termination detection algorithm for distributed computations. Inf. Process. Lett. **16**, 217–219 (1983)
9. Dijkstra, E.W., Scholten, C.S.: Termination detection for diffusing computations. Inf. Process. Lett. **11**(1), 1–4 (1980)
10. Feijen, W.H.J., van Gasteren, A.J.M.: Shmuel Safra's termination detection algorithm. In: Feijen, W.H.J., van Gasteren, A.J.M. (eds.) On a Method of Multiprogramming. Monographs in Computer Science, pp. 313–332. Springer, Heidelberg (1999). https://doi.org/10.1007/978-1-4757-3126-2_29
11. Fokkink, W.J.: Distributed Algorithms: An Intuitive Approach, 2nd edn. MIT Press, Cambridge (2018)
12. Francez, N.: Distributed termination. ACM Trans. Program. Lang. Syst. **2**, 42–55 (1980)
13. Helary, J.-M., Hurfin, M., Mostefaoui, A., Raynal, M., Tronel, F.: Computing global functions in asynchronous distributed systems with perfect failure detectors. IEEE Trans. Parallel Distrib. Syst. **11**(9), 897–907 (2000)
14. Hursey, J., Graham, R.L.: Building a fault tolerant MPI application: a ring communication example. In: Proceedings of IPDPS Workshop on High Performance Computing, pp. 1549–1556. IEEE (2011)
15. Lai, T.-H., Wu, L.-F.: An (N-1)-resilient algorithm for distributed termination detection. IEEE Trans. Parallel Distrib. Syst. **6**(1), 63–78 (1995)
16. Lifflander, J., Miller, P., Kale, L.: Adoption protocols for fanout-optimal fault-tolerant termination detection. In: Proceedings of PPoPP, pp. 13–22. ACM (2013)
17. Matocha, J., Camp, T.: A taxonomy of distributed termination detection algorithms. J. Syst. Softw. **43**(3), 207–221 (1998)
18. Mattern, F.: Algorithms for distributed termination detection. Distrib. Comput. **2**(3), 161–175 (1987). https://doi.org/10.1007/BF01782776
19. Mattern, F.: Global quiescence detection based on credit distribution and recovery. Inf. Process. Lett. **30**(4), 195–200 (1989)
20. Mittal, N., Freiling, F., Venkatesan, S., Penso, L.: On termination detection in crash-prone distributed systems with failure detectors. J. Parallel Distrib. Comput. **68**(6), 855–875 (2008)
21. Tel, G.: Introduction to Distributed Algorithms, 2nd edn. Cambridge University Press, Cambridge (2000)
22. Tseng, T.C.: Detecting termination by weight-throwing in a faulty distributed system. J. Parallel Distrib. Comput. **25**(1), 7–15 (1995)
23. Venkatesan, S.: Reliable protocols for distributed termination detection. IEEE Trans. Reliab. **38**(1), 103–110 (1989)