Experimenting with a fault-tolerant version of Safra's termination detection algorithm

Technical report for PDCS programming project

Per Fuchs

November 14, 2018

1 Introduction

This technical report presents an experiment using our fault-tolerant version of Safra's termination detection algorithm for rings (short SafraFT) [8] as control algorithm to detect termination of two different basic algorithms: a fault-tolerant version of Chandy-Misra's routing algorithm and Afek-Kutten-Yung's self-stabilizing spanning tree algorithm. Both basic algorithms build a tree containing all nodes in a distributed system. However, they cannot determine termination by themselves; that is, they cannot detect if they succeeded to build tree themselves, but they need a termination detection algorithm to verify that they completed their goal. This is the task of SafraFT; when it detects termination, it announces this finding to all nodes of the distributed system. SafraFT is a version of Safra's termination detection algorithm [6] that can deal with fail-stop failures[1]. That means it detects termination correctly even in the presence of fail-stop failures.

The following report is not self-contained. I recommend to read these texts first:

- the chapters about Chandy-Misra, Afek-Kutten-Yung, Safra's termination detection algorithm and a definition of termination detection in *Distributed Algorithms an Intuitive Approach* by Fokkink [7]
- the technical report *Fault-Tolerant Termination Detection with Safra's Algorithm* by Fokkink and Karlos [8] for a detailed explanation of SafraFT.

The next paragraphs give an overview on the structure of this report and the experiments conducted.

The fault-tolerant Chandy-Misra version is developed for this project as an extension of the fault sensitive Chandy-Misra algorithm described in [7] on page 57. An explanation of the extension and further important implementation decisions can be found in section 2.

The experiment aims to

- back up our claim that SafraFT is correct by using it in a realistic setting and verifying that termination is detected in a timely manner after actual termination occurred
- compare the performance of SafraFT to the fault-sensitive Safra implementation (abbreviated SafraFS) from [5]
- demonstrate the ability of SafraFT to handle networks of 25 up to 2000 nodes in a fault-free, 1 to 5 faults, and highly faulty (90% node failure) environment

Towards this aim, I measure the following dependent variables

- total number of tokens sent
- number of tokens sent after the basic algorithm terminated
- number of backup tokens sent
- average token size in bytes
- processing time for Safra's procedures
- time spent processing the basic algorithm's procedures
- wall time for the complete computation
- wall time between termination of the basic algorithm and detection of the fact

All these metrics are measured within the following environments:

- network size of 25, 250, 500, 1000 and 2000 nodes
- using SafraFS and SafraFT
- in a fault-free environment
- for SafraFT additionally with 1 - 5 and up 90% node failures (simulating nearly fault-free networks and highly faulty environment)

I do not aim to show the exact relationships between the dependent variables and the independent variables, e.g. the relationship between backup tokens send and the number of faults. This is because

[1] Fail-stop is a failure model that models failures as a binary, permanent property: a node is either running as normal or has failed; if a node fails, it does not send or react to any messages anymore and cannot be repaired.

the exact relationship depends heavily on the basic algorithm, the network and even the hardware the system is running on. Therefore, detailing the dependence would not be helpful to anyone considering to apply our algorithm to his system. However, this experiment should enable the reader to judge if SafraFT could be used for his system and convince him that SafraFT performance is comparable to that of SafraFS in a fault-free environment (except for its higher bit complexity). Furthermore, this report aims to show how SafraFT behaves in a faulty environment.

During my experiments, I found two bugs in SafraFT. We could fix those without much change and the results presented in section 4 are runs on the fixed version of SafraFT. The bugs and the fixes are explained in section 3.

Before performing this experiment, George Karlos applied SafraFT to a simulated basic algorithm in a multi-threaded environment emulating a distributed system. These experiments showed strong evidence towards correctness and scalability of SafraFT. The implementation, technical report and results can be found in George's bachelor thesis [9].

The experiment presented here is performed on the recommendation of a reviewer of [8] to add a 'compelling end-to-end application'.

2 Methods

This section describes the most important software design decisions taken during implementation, e.g. how to add fault-tolerance to Chandy-Misra or how to simulate faults. Later, I describe the software and machines used.

2.1 Fault-tolerant Chandy-Misra

This subsection describes the fault-tolerant extentsion of Chandy-Misra's routing algorithm [4]. For a detailed description of original the algorithm see [7, page 56]. The algorithm builds a shortest path sink-tree towards a single node, called root, in a bidirectional network. The root initializes the algorithm by sending `dist 0` messages to all its neighbours. On reception of a `dist d` message each nodes checks if the path via the sender would be shorter than its current path to the root node if so, it updates its distance variable, parent and sends `dist d` messages to all its neighbours. The algoritm terminates when all `dist d` messages have been processed.

The fault-tolerant version used for the experiments constructs a sink-tree in a bidirectional network under the assumption that a perfect failure detector[2] is present at each node. Other than the original Chandy-Misra algorithm, the fault-tolerant version requires FIFO-channels. Furthermore, I assume that the root node cannot fail because otherwise there is no sink tree to construct. The following assumes that the reader is familiar with the original Chandy-Misra algorithm and describes only the extensions necessary to add fault-tolerance.

In the context of Chandy-Misra, a node is only interested in crashes of its parent and other ancestors on its path to the root node. If a node `X` detects a crash of its parent, it sends a `REQUEST` message to all its neighbours. If a neighbour `Y` of `X` receives a `REQUEST` message, it answers with a `DIST d` message where `d` is its own current distance to the root node. `Y` only sends this message if $d < \infty$. If `Y` happens to be a child of `X`, it resets its own `dist` and `parent` variables to ∞ respectively $\perp$ and sends a `REQUEST` message to all its neighbours. In this case `Y` sends no `DIST` message as an answer to `X`.

Next, I present an intuition why this extension is fault-tolerant. Each failure only affects nodes that see themselves as children, grandchildren or deeper ancestors of the crashing node; that is, a failure only affects subtrees. The children of the failing node eventually send `REQUEST` messages to all their neighbours because the perfect failure detector guarantees that each node failure is eventually detected by them. The neighbours send `REQUEST` messages to all their neighbours if they receive a `REQUEST` message from their parent. Therefore, eventually, all nodes in the subtrees of a failing node are reached by a `REQUEST` message and reset their `dist` and `parent` values. Also, all neighbours that receive a `REQUEST` message from a node that is not their parent, answer it with their current `dist` value. This allows nodes in the affected subtree to rebuild new paths toward the root node. These new paths are correct when the answering node is not part of any affected subtree. However, if they are part of an affected subtree (e.g. grandchildren of the crashed nodes), invalid paths are introduced - as these nodes might have not been reached by any `REQUEST` message and therefore still believe they have a valid path towards the root. These invalid paths are corrected when the grandchildren are reached by the `REQUEST` message of their parent because on the receipt they send `REQUEST` messages which reset all nodes considering them parents. This behaviour of introducing invalid paths that are corrected later might lead to a bad theoretical message complexity but did not hinder the experiments.

The requirement for FIFO channels is best understood by a counterexample based on a non-FIFO network. The network used for this example is shown in fig. 1. Only important messages are mentioned; all others can be assumed to be sent and received in any order. The Chandy-Misra algorithm starts with `A` as root node sending `DIST 0` messages to `B` and `C`, which on reception consider `A` their parent and update their `dist` variable. They also send `DIST` messages to their neighbours. When `C` and `D` receive the `DIST` messages from, `B` respectively `C`, they consider `B` respecitively `C` their parents. `C` sends a `DIST 2` message to `D` - lets call it `M1`. Now `B` crashes and when `C` detects

[2] a perfect failure detector does not suspect nodes that haven't actually failed and detects each failure eventually

this, it sends a REQUEST message towards A and D - call the latter M2. If M2 overtakes M1, D resets its variabbles and on receiving M1 considers C its parent, which is correct but with an incorrect dist value of 2. All DIST messages received by D from now on have a higher distance value and are dismissed. So the error is never corrected. A straightforward fix for this is to use FIFO networks because the original problem is that M2 overtook M1[3].

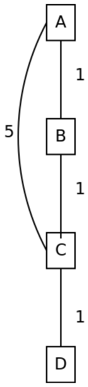

Figure 1: Non-FIFO network with A as root to demonstrate necessity of FIFO networks.

The presented fault-tolerant Chandy-Misra algorithm could be incorrect as it calculated the wrong tree in 12 out of 1500 runs, Unfortunately, I could not fix it in time for the final experiment runs. However, this does not influence the evaluation of SafraFT, because the correctness check for SafraFT and Chandy-Misra in the experiment setup are designed to work independently from each other (see section 2.5). The verification of SafraFT does show that SafraFT detected termination correctly in these cases and did not cause the corrupted Chandy-Misra result. Additionally, the bug also shows in runs with SafraFS.

Additionally to fixing the last bugs, the fault-tolerant Chandy-Misra version could be improved by relieving the necessity for FIFO-channels, a more formal and adapted proof of correctness and a thorough complexity analysis.

2.2 An adaption to Afek-Kutten-Yung

I implemented the Afek-Kutten-Yung's self-stabilizing spanning tree algorithm originally proposed in [1] and described in a message passing setting in [7, page 183ff][4]. The original Afek-Kutten-Yung operates in lock-step fashion in shared-memory environments. Therefore, it becomes extremely slow when it is used in a message passing environment with many nodes. Hence, I optimized it to allow runs up to 2000 nodes. The optimization could exhibit infinite runs. Next, I explain the optimization and present the idea why it might exhibit infinite runs and state why this is no problem for the experiments. Before, I give a short explanation of the original Afek-Kutten-Yung.

The original Afek-Kutten-Yung algorithm is self-stabilizing and builds a spanning tree of all nodes in a distributed system with shared memory. Each node is supposed to have an ID; the node with the highest ID becomes the root of the spanning tree. Nodes reset all internal variables if they detect inconsistency with their neighbours. After, they request to join the tree of their neigbour with the highest root. To ensure, that they can only join via a node that is actually part of a valid tree, i.e.

[3] During the experiments, SafraFT uses the same FIFO channels as its basic algorithm. Nonetheless, we still claim it does not require this property. This claim is straightforward to prove: SafraFT guarantees that at most one message is in any channel at all times because it forwards a single token and when a backup token is issued, it is sent to a different node than the original token and only one of both tokens is forwarded. Following the reasoning that only one message is in flight between any two nodes, SafraFT is indifferent to the property FIFO property of the channels.

[4] During my implementation, I found 3 flaws in the pseudo code given: wrong variable used, a serious bug leading to deadlocks and a case of pseudo code that guides the developer towards an implementation which exhibists null pointer exceptions. The author kindly the feedback and will correct the pseudo code in the next revision.

a node that can reach its root, the join request is forwarded to the root and the approval is returned via the same path. So, Afek-Kutten-Yung only allows nodes to join a tree after forwarding their join request to root and returning an approval. This is quite slow for big networks because each node is only allowed to handle a single join request at any time; it is a good starting point for optimizations.

Intuitively, it is not necessary to forward a join request to the root every time because every node who joined the tree has been approved before (except for the case that it has been initilized incorrectly which is handled later). Therefore, it can directly (without forwarding) approve further nodes. Following, this intuition I use an adapted Afek-Kutten-Yung algorithm that does not forward join requests but approves them directly. As stated above, the assumption, that a node has been approved before it joins a tree, does not hold for initially existing illegal trees. However, in this case at least one node does not have a consistent parent. When it detects this, it resets its variables which leads to its children reseting and so on. So, the initially existing tree will dissolve.

After implementation and successful testing of this optimization, I realized, by reading the original Afek-Kutten-Yung paper [1], that this variation could lead to infinite runs. It breaks the proof of Lemma 4.9 which states that the set of false roots is monotonically decreasing[5]. With this optimization, this cannot be proven as in the original paper because they rely on the fact that no node can join an illegal tree. This is the case in the original algorithm because no join request can be approved before reaching the root, which is not possible for an illegal root. In the optimized version, this is not the case because nodes can join an illegal tree without noting that the root does not exist. Although the original proof is broken, it is not straightforward to find a concise example of an infinite run because joining a tree takes three rounds while the process of leaving a tree takes only one. I do not attempt to prove or disprove the existence of infinite executions in the scope of this project. However, I note that (1) infinite runs are no problem for Safra[6] and (2) no infinite run occurred in my experiments.

2.3 Fault Simulation

I simulate faults by stopping Safra and the basic algorithm on the failing instance. In particular, a crashed node does not send or react to tokens or basic messages. Faults are triggered before and after interesting events, e.g. directly before or after sending a basic message or token. Before every experiment run it is determined at random which node is going to fail, on which event it is going to fail, and after how many repetitions of this event, e.g. after the 3rd time, it forwards a token.

In particular, I selected the following events to trigger a crash, if not specified differently, the crash is triggered right before or after the event:

- sending a token (1 to 3 repetitions)
- sending a backup token (1 to 2 repetitions)
- before receiving a token (1 to 3 repetitions)
- sending a basic message (1 to 5 repetitions)

The range of repetitions is limited to maximize the chance that a node meant to fail actually does so. However, a node that is planned to fail is not guaranteed to do so. For example, this leads to runs where 90% of the nodes should fail but only 88% do so. I verify for every run that the number of crashed nodes does not vary more than 10% from the expected number. For runs with only 1 to 5 failing nodes, I confirm that at least one instance failed.

Alternatively, to the chosen approach to trigger failures, I considered the more random mechanism of running a thread that kills an instance after a random amount of time. One could argue that this would be more realistic. However, I believe this kind of approach leads to less interesting failures because the vast majority of these failures would occur during idle time. Furthermore, most failures between internal events are observed as exactly the same on all other nodes. For example, other nodes cannot observe if a failure happened before an internal variable changed or after. In fact, they

[5] False roots are root variables set to a value that is not a valid ID in the network

[6] In this case announce is never called, which is correct behaviour.

can only observe a difference when the failure happens after or before sending a message to them. Hence, I have chosen a fault mechanism that focusses on these distinguishable scenarios. As one might notices, the failure points are chosen to give rise to many different situations for our Safra version to deal with. I deliberately decided against choosing special failure points with regard to the basic algorithm because this would lead to less focused testing of the fault tolerance of Safra.

2.4 Fault Detection

Our Safra version assumes the presence of a perfect fault detector. This kind of fault detection is easy to implement and integrate with the system, e.g. [7] on page 113 describes a straightforward implementation.

As building a perfect fault detector is a well-known and solved problem, but nonetheless time-consuming, I decided to avoid implementing one. For this experiment, fault detection is simulated by sending `CRASH` messages from crashing nodes to their neighbours. These crash messages are sent through different channels than basic and Safra messages because otherwise they would arrive in FIFO order with all other messages and this would exclude situations where a basic message is received after the crash of the sender has been detected.

`CRASH` messages are not broadcasted to all nodes because IBIS (the message passing library I used) does not provide broadcasting.

2.5 Offline Analysis

For the experiment, I measure some metrics before and after termination, e.g. the total token count and tokens sent after termination of the basic algorithm. To allow generation of these metrics, I need a close estimate of when the basic algorithm terminated. For this, I generate log files of events during execution and analyse these afterwards.

Termination is commonly defined by:

1. All nodes are passive
2. No messages are in the channels

To allow verification of the first part every node logs changes of its active status. The second point can be verified indirectly by logging all changes of the message counters managed by Safra's algorithm. These counters are incremented for each message sent and decremented when a message is received. Therefore, one can conclude that no messages are in flight when the sum of all counters is 0. All nodes log the aforementioned events combined with a timestamp. By sorting and keeping track of active statuses, as well as the sum of message counters, one can estimate the time of basic termination by the timestamp of the last node becoming passive while the sum of all message counters is zero. This technique is similar to the one Safra uses, but a global view on the system achieved by offline analysis allows to detect the time of basic termination precisely.

With this system in place, it is possible to determine the number of tokens sent after termination by logging each token sent event and categorizing them during the offline analysis.

Processing time metrics are determined by the same principle: processing time is logged online and is grouped into total and after termination by analysing the logs after the run. Wall time between basic termination and detection by Safra is determined by comparing the timestamp of the event causing termination with the timestamp of the to announce call by Safra.

2.6 Environment

This chapter describes software, hardware and simulated network topology used for the experiments.

2.6.1 IBIS

IBIS is a Java-based platform for distributed computing developed by researchers of the Vrije Universiteit Amsterdam [3]. Its subpart IPL is used as the message passing library for this project. I use version 2.3.1 which can be found on GitHub: https://github.com/JungleComputing/ipl/releases/tag/v2.3.1.

Communication channels in IPL are backed by TCP and provide asynchronous messaging. For this experiments, I also used IPL's ability to guarantee FIFO channels.

2.6.2 DAS 4

The experiment is conducted on the part of DAS-4 that belongs to the Vrije Universiteit Amsterdam [2]. The nodes use primarily SuperMicro 2U-twins with Intel E5620 CPUs and a Linux CentOS build with the kernel version 3.10.0-693.17.1.el7.x86_6. For communication 1Gbit/s Lan is used. At the time of the experiments, the VU had 41 nodes with 8 cores each. Therefore, multiple instances were run on each physical node to be able to test our algorithm on decently sized networks (the number of machines and instances per machine can be found in table 1). This is possible because Safra and Chandy-Misra are both communication heavy with rather low processing and memory requirements.

2.6.3 Network Topology

Chandy-Misra needs a network topology to work on. It requires a bidirectional network. To generate more interesting runs I use weighted networks.

Our Safra version needs a bidirectional ring. For simplicity in my setup, this ring is part of the network the basic algorithm runs on. That means there is always a bidirectional ring connecting all nodes within simulated networks.

All networks for the experiments are generated by choosing randomly between 1 and 5 neighbours for each node and assigning a random, unique weight between 1 and 50000 to each channel.

After this, channels with the heavy weight of 400000 are added between the root and some nodes to ensure the network stays connected when nodes fail. For this, I calculate the expected network with knowledge of the nodes predetermined to fail and add connections between nodes that could become disconnected as some other nodes fail and the root. These channels are heavy-weight to avoid using them over ‘regular’ channels which might create highly similar topologies with a lot of nodes directly connected to the root.

At last, channels are added to form a bidirectional ring in which each channel has the weight of 100000. Again the weight is chosen to avoid ‘overusing’ the ring channels for the trees built.

The network topology used for each run is recorded.

3 Discovered bugs and fixes

During the experiments, I found two bugs in SafraFT. I first give an example execution for each bug. After, I describe how they can be fixed. This fix is implemented and tested - all repetitions mentioned in section 4 were run on the fixed version of SafraFT. The old version of SafraFT, which includes the bugs, can be found in section A. The new, fixed version is presented in the technical report proposing the algorithm [8].

The first bug is that if SafraFT receives two tokens in one round, it removes all nodes from the crash report of the second token and does not forward them. This can lead to disagreement about which nodes crashed between the alive nodes. In this case, the alive nodes cannot agree on how to calculate the sum over the token message counters. Hence, they may never call announce. The bug only surfaces if the employed failure detector does not propagate failures to all nodes but relies on SafraFT to do so. Consider the following example:

1. `X` receives a token from `X-1` with $\mathit{CRASHED}_t$ set to $\{1\}$.
2. `X` is not passive and does not forward the token but it updates $CRASHED_x$ (line 4 of `ReceivedToken`).
3. `X-1` crashes.
4. `X-2` detects the crash of `X-1`.
5. `X-2` sends a backup token with $\mathit{CRASHED}_t$ set to $\{1\}$.
6. `X` receives this token.
7. `X` removes `1` from $\mathit{CRASHED}_t$ (line 3 of `ReceivedToken`).
8. When `X` forwards the token, `1` is not in the crash set and `X+1` might never detect the crash of `1`.
9. `X` and all nodes before it know that `1` crashed but some nodes after `X` do not.
10. They disagree on which counters to use for calculating the sum and never call `announce`.

The second bug is caused by the fact that SafraFT updates its token variables when a token is received but only increases its sequence number when the token is forwarded. However, if it forwards a backup token, it uses the sequence number of the updated token variables. This can lead to a situation in which SafraFT ignores a valid token because its token number is too high. This situation is presented in the following example:

1. `X` with $\mathit{seq}_x = a$ receives token `T` with $\mathit{seq}_t = a + 1$, it updates its token variables (line 2 of `ReceivedToken`).
2. `X` is not passive and does not forward the token
3. `X` detects the crash of `X+1`.
4. `X` forwards a backup token T_1 with the sequence number `a+1` (`X` is not the biggest node ID in the ring).
5. T_1 completes the full round.
6. It reaches `X` with the sequence number `a+2`; `X` drops the token because it expected a token with the sequence number `a+1` (line 1 of `ReceivedToken`).
7. The token has been lost and SafraFT will never call `announce`.

Both bugs have the same root cause, namely that the updates in `ReceivedToken` are not atomic with the updates of other variables in `HandleToken`. Therefore, both can be fixed with the same changes which I present next:

- Move line 3 to 5 from `ReceivedToken` to the beginning of `HandleToken` after line 1.
- Instead of updating all token variables (line 2 of `ReceivedToken`), update only the $CRASHED_t$ set in `ReceivedToken`[7].
- Check if the receiver (`j`) is not in $CRASHED_i \cup \mathit{REPORT}_t \cup CRASHED_t$ in line 1 of $SendBasicMessage_i$.

[7] This means only populating $\mathit{CRASHED}_t$ with the value of the current token, not removing $\mathit{CRASHED}_i$ from it because this functionality moves to `HandleToken`

The last two points are necessary to avoid the overhead of sending many messages to a node, that has crashed, even though this could have been avoided because the sender is aware that this node crashed but did not update $CRASEHD_t$. This fixes the first bug because $CRASHED_t$ is only updated once. The second bug is fixed because the seq_t is only updated atomically with seq_i.

Algorithm	Network	Faults	Repetitions per basic algorithm	#Instances / DAS-4 node
SafraFS on CM / AKY	50/250/500/1000	0	> 50	50
SafraFS on CM / AKY	2000	0	> 50	100
SafraFT on CM / AKY	50/250/500/1000	0	> 50	50
SafraFT on CM / AKY	2000	0	> 50	100
SafraFT on CM / AKY	50/250/500/1000	5n	> 50	50
SafraFT on CM / AKY	2000	5n	> 50	100
SafraFT on CM / AKY	50/250/500/1000/	90%	> 50	50
SafraFT on CM / AKY	2000	90%	> 50	100

Table 1: List of all configurations run. Per physical DAS-4 node with 8 cores multiple multiple virtual instances in their own processes where run (columm '#Instances / DAS-4 node'). The amount of physical node equates to network size divided by instances.

4 Results

In the next sections, I present the main results of my experiment to support our claim that SafraFT is correct, to show how SafraFT compares to SafraFS and to exemplify the performance of SafraFT under the presence of faults.

The observations are based on runs of the system described in section 2 on the DAS-4 cluster at the Vrije Universiteit of Amsterdam. I measured runs on networks from 50 to 2000 nodes (see section 5 for an explanation of the upper limit) for SafraFT and SafraFS. SafraFT was also tested with 1 to 5 nodes failing per run (dubbed 5n) and with 90% node failure. table 1 presents how many repetitions of each configuration were run. The table also shows that for runs with 2000 nodes more instances shared the same physical machine (due to limited hardware availability). This might lead to differences between the experiment results of smaller networks and networks with 2000 instances.

The implementation can be found at GitHub[8]. The README of this project is the manual on how to conduct experiments and how to interpret the results. I used one version to generate all results[9]. The raw data used for this report can be obtained on request from Wan Fokkink (https://www.cs.vu.nl/ wanf/).

The results of runs with CM or AKY are quite similar and mostly show the same effect on the metrics. Therefore, I analyse both together and explicitly state if an observation or explanation only applies to one of both.

4.1 Correctness of SafraFT

The experiment is aimed to support our paper with a practical, correct application of our algorithm. Towards this goal, I build multiple correctness checks into the experiment.

To assure nothing happens after termination detection, the application logs if any messages are received or actions are executed after termination has been detected and announced. The analysis tools provided with the experiment point these logs out to make sure they are not overlooked.

To prove termination is not detected too early, I use offline analysis (see section 2.5) to determine the point of actual termination and verify that detection happened after. All 1124 runs using SafraFT under the presence of faults confirm that termination was never detected too early.

However, the experiment revealed that the framework for termination chosen to develop SafraFT is not complete and does not cover all cases for my experimental setup. SafraFT is developed for the following and commonly used definition of termination:

1. All nodes are passive
2. No messages are in the channels

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

[9] Commit Hash: aa4e52c56885b565776604be99456f58ac8866b9

This definition is based on the fact that a node is either an initiator or can only become active if it receives a message. Anyhow, in the presence of failures and if additionally, the outcome of the algorithm depends on the set of alive nodes, nodes might get activated by the detection of a failure. For example, when Chandy-Misra builds a sink tree, nodes that detect a crash of their parents will become active afterwards to find a new path towards the root. This fact leads to the following concrete scenario in which the definition of termination assumes termination too early: let us consider the situation that all nodes are passive and no messages are in the channel. In other words, the system terminated by our definition. Node `X` forwards the token to `Y` and crashes afterwards. Node `Y` calls announce after receipt of the token. Assume node `Z` is a child of `X` and detects the crash of its parent, it becomes active after termination has been formally reached and announced. By sending out `REQUEST` messages, it might activate other nodes again. To conclude, the definition of termination that our algorithm is built upon does not fully capture the reality of our basic algorithms which could lead to an early detection of termination.

I propose an extended definition of termination to close this gap between theory and practice:

1. All nodes are passive
2. No messages are in the channels
3. Termination is postponed until the last node failure that leads to action is detected

All repetitions of this experiment have been analysed according to the common definition of termination used to develop SafraFT and the extended definition. As stated above, SafraFT did never detect termination too early according to its definition. According to the extended definition, it detects termination to early in 13 out 1124 runs with crashes. However, due to the fact that repairing the constructed trees after detecting a parent crash is quite fast and there is a short time window to do so while the announce call propagates to all nodes, only in 2 of these cases that leads to the situation that basic activity happened after detected termination.

I carefully reviewed each repetition in which termination is detected too early according to the extended definition to verify that early termination detection is in fact, caused by a situation as described above. The logs of these runs provide a summary of all detections of parent crashes close to the announce call to ease this procedure.

4.2 Comparision of Safra versions

This section compares SafraFS and SafraFT. Additionally, it analyses how the network size influences both algorithms.

The number of tokens sent in total and after termination is presented in fig. 3 for CM and in fig. 3 for AKY.

The key observation is that SafraFS and SafraFT behave highly similar. SafraFT sends slightly fewer tokens on average. Also, its results show somewhat less variance.

As one would expect, the number of tokens grows linearly with the network size. Note that the first network size is 5 times smaller than the second; for bigger networks the size doubles for each run.

The same can be observed for tokens sent after termination. By far most tokens are sent after termination, as one would expect.

The bit complexity of SafraFS is constant. In this experiment, each token of SafraFS contains 12 bytes. SafraFT has a bit complexity linearly to the network size (when no faults occur). For a network of 50 nodes, each token has 220 bytes; a token in a 2000 nodes network counts 8020 bytes. The growth can be described by $bytes = 4 * \langle networksize \rangle + 20$.

I measured two kinds of timing metrics in this experiment. On the one hand, there are the wall time metrics of total time and total time after termination. Both were recorded in elapsed seconds between two events. For the total time, these events are the start of the basic algorithm and termination detection event at the last node. Total time after termination is defined as the number of seconds between the actual termination (extended termination definition from section 4.1) and the event of a node calling announce. On the other hand, there are basic, Safra and Safra after termination

Network	Basic	Safra FS	Safra FT	Overhead FS	Overhead FT	Safra FS	Safra FT
50	0.462	0.022	0.034	4.76%	7.36%	0.005	0.010
250	3.327	0.123	0.235	3.70%	7.06%	0.028	0.104
500	11.18	0.264	0.636	2.36%	5.69%	0.059	0.336
1000	26.926	0.589	1.148	2.19%	4.26%	0.120	0.541
2000	122.707	1.428	2.663	1.16%	2.17%	0.233	1.047

Table 2: Total processing times (left) and processing times after termination (right) in seconds and overhead over Chandy-Misra caused by Safra in percent

Network	Basic	Safra FS	Safra FT	Overhead FS	Overhead FT	Safra FS	Safra FT
50	0.865	0.035	0.049	4.05%	5.66%	0.005	0.010
250	2.485	0.152	0.261	6.12%	10.50%	0.024	0.081
500	4.692	0.301	0.520	6.42%	11.08%	0.047	0.177
1000	13.806	0.676	1.460	4.90%	10.58%	0.114	0.699
2000	47.572	1.626	6.034	3.42%	12.68%	0.302	3.848

Table 3: Total processing times (left) and processing times after termination (right) in seconds and overhead over Afek-Kutten-Yung caused by Safra in percent

processing times (including the time needed to send messages). These are the accumulated times all instances needed to process basic or Safra functions. Total times and processing times are measured in a different way and should not be compared directly for multiple reasons. First, while total times include idle times, e.g. time spent for logging, processing time do not include these. Secondly, total time is wall time between two events and processing times are accumulated over all processes. One particular example of when this leads to differences is that time spent concurrently by two processes counts double in processing time metrics but only once in wall time metrics.

One can observe in table 2 and table 3 that SafraFT uses more processing time than SafraFS and much of the additional time is spent between actual termination and termination detection. The processing time increases linearly with the network size for SafraFS. For SafraFT doubling the network size leads to a 2 or threefold increase in processing time. Most of the additional processing time of SafraFT is spent between termination and termination detection. One can also see that the processing time increased more than twofold for networks of 1000 nodes to 2000 nodes.

A good candidate to explain the difference in processing time between SafraFS and SafraFT is the different token size of SafraFS and SafraFT: it takes longer to send bigger tokens. Also, this theory is supported by the fact that most additional time is spent after termination - Safra is only concerned with moving the token around after termination. To confirm this idea, I ran an ad-hoc experiment of excluding the time spent to send messages from the processing time. Indeed, this experiment revealed that most of the differences can be tributed to writing tokens onto the wire. This confirms that these differences are caused by the higher bit complexity of SafraFT. This would explain the total increase in the timing from SafraFS to SafraFT, as well as the bigger influence of network size - bigger networks lead to even larger tokens.

The qualitative result is not surprising. However, we are surprised by the quantitative size it has. We would have expected that the difference between SafraFS and SafraFT in processing time is observable but much lower. I discuss this issue in section 5.

The processing time tables 2 and 3 also present a comparison of the time spent on the basic algorithm and both Safra versions. Although SafraFT uses significantly more time, the overhead on the processing time of the basic algorithm stays moderate with a maximum of 12.68% for 2000

Network	Safra FS	Safra FT	Δ	SafraFS	SafraFT	Δ
50	0.148	0.161	1.09	0.017	0.025	1.47
250	0.340	0.466	1.37	0.1	0.217	2.17
500	0.610	1.064	1.74	0.225	0.655	2.91
1000	1.195	1.863	1.56	0.495	1.166	2.36
2000	2.483	3.747	1.51	1.05	2.310	2.20

Table 4: Wall times total (left) and after termination (right) for SafraFT and SafraFS in seconds with ratios run with CM

Network	Safra FS	Safra FT	Δ	SafraFS	SafraFT	Δ
50	0.528	0.523	0.99	0.028	0.034	1.21
250	0.798	0.864	1.08	0.100	0.185	1.85
500	1.065	1.193	1.12	0.194	0.384	1.98
1000	1.987	2.555	1.29	0.543	1.268	2.34
2000	6.016	8.811	1.46	1.962	5.406	2.76

Table 5: Wall times total (left) and after termination (right) for SafraFT and SafraFS in seconds with ratios run with AKY

nodes and AKY.

The same pattern of SafraFT using more time and reacting stronger to an increase in network size is visible for total times in table 4 and table 5 for CM and AKY respectively.

For SafraFS and CM, roughly half of the time is spent after termination for small networks. In big networks, the part of the time spent after termination is lower because the fraction spent by the basic algorithm becomes dominant. The systems using SafraFT spent half or more of their time to detect termination.

For AKY most of the time is spent before termination (except for 1000 and 2000 nodes and SafraFT). This shows that the time measurements highly depend on the basic algorithm. AKY is a synchronized algorithm and needs more time than CM.

I would like to note that the low processing time overhead of Safra is not in contradiction to a large amount of wall time spent after termination. These seemingly opposing results arise from the difference between wall time and processing time: the basic algorithm is much more active at the beginning that is when it accumulates a lot of processing time; while Safra causes a lot of idle time in the end when all processes wait for their predecessor to pass on the token. This idle time is not included in processing time, but wall time does include it.

To conclude, the experiments confirm that the message complexity of SafraFT remains as for the fault-sensitive version, but its higher bit complexity causes it to use more time, which leads to a later termination detection. Still, SafraFT causes only a moderate processing time overhead between 2% and 13%.

4.3 Influence of faults

In the following paragraphs, I present and explain the data generated by runs in the presence of node crashes. I run two highly different scenarios: one considering networks with 1 to 5 nodes failing and one with 90% of all nodes crashing. These scenarios are chosen to show SafraFT in both the realistic case of a low number of faults to handle, as well as an extreme case; with the aim to confirm that SafraFT handles both cases correctly and without unreasonable deterioration in any metric.

Network	No faults	5n	Δ	90%	Δ	No faults	5n	Δ	90%	Δ
50	69	101	1.46	105	1.52	42	52	1.24	21	0.50
250	287	484	1.69	544	1.90	236	287	1.22	75	0.32
500	540	774	1.43	1100	2.04	486	557	1.15	138	0.28
1000	1051	1907	1.81	2187	2.08	977	1294	1.32	236	0.24
2000	2062	4028	1.95	4396	2.13	1973	2561	1.30	460	0.23

Table 6: SafraFT and CM: Tokens in total (left) and after termination (right) for different fault scenarios compared to fault-free networks.

Network	No faults	5n	Δ	90%	Δ	No faults	5n	Δ	90%	Δ
50	76	110	1.45	91	1.20	40	39	0.97	6	0.15
250	298	428	1.44	470	1.58	227	234	1.03	40	0.18
500	559	837	1.50	966	1.73	470	496	1.06	84	0.18
1000	1088	1975	1.82	1915	1.76	962	1020	1.06	180	0.19
2000	2135	4081	1.91	3847	1.80	1936	2153	1.11	367	0.19

Table 7: SafraFT and AKY: Tokens in total (left) and after termination (right) for different fault scenarios compared to fault-free networks.

I used the extended definition of termination to determine the point of time of actual termination to generate the metrics shown in this section.

4.3.1 Tokens

For both the networks where between 1 and 5 nodes failed, as well as for the highly faulty runs with 90% node failure, the number of tokens increased compared to runs without any faults. Runs with 90% failure produced even more tokens than runs with only 1 to 5 node failing - except in networks of 2000 nodes where 5n exhibits a higher tokens sent average. The data is presented fig. 4 / 5 and table 6 / 7 for CM respectively AKY.

Otherwise, the two types of fault simulation have a highly different influence on the tokens and tokens after termination metrics.

5n configurations lead to a strong increase in the variance of tokens sent and in tokens sent after termination. This seems reasonable because runs with failing nodes might lead to more different situations than runs without fails, e.g. one failing node could easily cause an extra token round when it leads to a backup token being issued and forwarded (this token is marked black until it reaches the node it originates from), while by contrast, a single failing node that is a leaf in a Chandy-Misra sink tree and that crashes just before the call of announcing at the successor causes no further activity. The same example provides an idea of why the variability increases in big networks. That is because one extra round in a big network has a much higher impact on the token count than in a small network.

Opposed to the results for 5n, networks with up to 90% node failure lead to a similar low variance in tokens and tokens after termination as fault-free networks. A likely explanation is that the low survival rate of 1 out of 10 instances leads to fewer different scenarios than in the fault scenario treated in the last paragraphs.

Even though only one 10th of the nodes survive to participate in the latter token rounds, the highly faulty networks produced more tokens than any other network of the same size. That is most likely due to the high amount of backup tokens generated (also shown in fig. 4 and 5).

Network	No faults	5n	Δ	90%	Δ
50	220	228	1.04	282	1.28
250	1020	1028	1.01	1315	1.29
500	2020	2025	1.00	2606	1.29
1000	4020	4027	1.00	5198	1.29
2000	8020	8028	1.00	10362	1.29

Table 8: SafraFT and CM: Token size averages in bytes for both fault scenarios compared to token size with zero faults.

Network	No faults	5n	Δ	90%	Δ
50	220	227	1.03	284	1.29
250	1020	1025	1.00	1320	1.29
500	2020	2026	1.00	2603	1.29
1000	4020	4028	1.00	5187	1.29
2000	8020	8028	1.00	10354	1.29

Table 9: SafraFT and AKY: Token size averages in bytes for both fault scenarios compared to token size with zero faults.

Different from all other networks, highly faulty networks exhibit a much lower token to token after termination ratio caused by the low number of nodes alive in the last rounds.

As expected, faults lead to more tokens being sent. This is caused mostly because a fault detection requires an additional round. However, SafraFT handles faults kindly - neither 5n nor 90% scenarios lead to an overhead higher than 2.13 compared to a fault-free execution. Hence, not every fault causes an extra round, but all faults together cost an extra round. This result holds for all network sizes, both basic algorithms and for two highly different fault scenarios.

4.3.2 Backup tokens

The average number of backup tokens sent for either fault simulation or network size is lower than the number of faults. This is due to the fact that SafraFT only issues backup tokens when the fault of its successor is detected via the fault detector but not if this fault is noticed by receiving a token. There are even runs where 1 to 5 nodes fail but no backup token is issued, up to networks containing 2000 nodes.

The other extreme that more backup tokens are issued than faults occur exists as well. This can be explained by my decision to let nodes fail after issuing a backup token. For example, nodes `A`, `B` and `C` follow each other in the ring, node `C` fails, which is detected by `B`, and a backup token is issued. After, issueing the backup token, `B` fails, on detection `A` issues a backup token towards `C`. Only then `A` detects the failing of `C` and issues a second backup token to its new successor.

Although both extremes exist, the average number of backup tokens is close to the number of faulty nodes.

4.3.3 Token size

The average token size increases with faults because the IDs of faulty nodes are propagated by the token (see tables 8 and 9). The influence on token size is between 5 and 8 bytes for networks with 1 to 5 failing nodes. 90% node failure leads to a linear increase of token bytes to the network size of a factor 1.29.

Network	No faults	5n	Δ	90%	Δ	No faults	5n	Δ	90%	Δ
50	0.034	0.058	1.71	0.109	3.21	0.010	0.015	1.50	0.012	1.20
250	0.235	0.285	1.21	0.516	2.20	0.104	0.104	1.00	0.035	0.34
500	0.636	0.705	1.11	1.182	1.86	0.336	0.298	0.89	0.076	0.23
1000	1.148	1.594	1.39	2.427	2.11	0.541	0.702	1.30	0.110	0.20
2000	2.663	5.111	1.92	6.879	2.58	1.047	2.222	2.12	0.309	0.30

Table 10: SafraFT and CM: Total processing times (left) and after termination (right) in seconds for different fault scenarios compared to fault-free networks.

Network	No faults	5n	Δ	90%	Δ	No faults	5n	Δ	90%	Δ
50	0.049	0.073	1.49	0.111	2.27	0.010	0.01	1.00	0.002	0.20
250	0.261	0.287	1.10	0.596	2.28	0.081	0.073	0.90	0.011	0.14
500	0.520	0.648	1.25	1.475	2.84	0.177	0.186	1.05	0.027	0.15
1000	1.460	1.911	1.31	4.399	3.01	0.699	0.423	0.61	0.098	0.14
2000	6.034	4.254	0.71	9.931	1.65	3.848	1.701	0.44	0.172	0.04

Table 11: SafraFT and AKY: Total processing times (left) and after termination (right) in seconds for different fault scenarios compared to fault-free networks.

4.3.4 Processing Time

The observations of this chapter are backed by tables 10 and 3. As for tokens, one can see an increase in total processing times under the presence of faults compared to fault-free runs, which is no surprise because more tokens are sent. This also explains why the processing time is higher for the 90% scenario than in the 5n configurations. There is one exception: the processing time for 2000 nodes and AKY in the 5n scenario decreased.

Processing times after termination for 90% drastically decreased for both algorithms and all network sizes - except for CM and 50 nodes where one sees a 20% increase. The general result can be explained by fewer tokens being sent after termination. The exception cannot be explained with the data at hand because it does only occur for one algorithm.

The influence of 5n scenarios on processing time after termination is highly varied. In many configurations, it shows no big effect, e.g. 250 nodes and CM. In others, it decreases significantly, e.g. 1000 and AKY. And finally, one observes a sharp increase for 50 or 2000 and CM. The available data does not allow for a good explanation. One highly interesting fact is that 5n scenarios lead to more tokens being sent after termination (see table 6 and 7) but to less processing time after termination in many configurations. This situation might be explained by the fact that the sum of all token counters is calculated by most nodes if no nodes failed and only by some nodes when other nodes crashed - due to the different *blackUntil* values.

Network	No faults	5n	Δ	90%	Δ	No faults	5n	Δ	90%	Δ
50	0.161	0.186	1.16	0.254	1.58	0.025	0.033	1.32	0.030	1.20
250	0.466	0.599	1.29	1.052	2.26	0.217	0.235	1.08	0.092	0.42
500	1.064	1.237	1.16	2.334	2.19	0.655	0.624	0.95	0.180	0.27
1000	1.863	2.840	1.52	4.851	2.60	1.166	1.525	1.31	0.271	0.23
2000	3.747	8.005	2.14	12.823	3.42	2.310	4.271	1.85	0.676	0.29

Table 12: SafraFT and CM: Total times (left) and after termination (right) in seconds. For different fault scenarios compared to fault-free networks.

Network	No faults	5n	Δ	90%	Δ	No faults	5n	Δ	90%	Δ
50	0.523	0.627	1.20	0.518	0.99	0.034	0.033	0.97	0.007	0.21
250	0.864	0.913	1.06	1.466	1.70	0.185	0.174	0.94	0.023	0.12
500	1.193	1.514	1.27	3.410	2.86	0.384	0.405	1.05	0.058	0.15
1000	2.555	3.340	1.31	10.58	4.14	1.268	0.889	0.70	0.151	0.12
2000	8.811	7.816	0.89	33.77	3.83	5.406	3.080	0.57	0.313	0.06

Table 13: SafraFT and AKY Total times (left) and after termination (right) in seconds. For different fault scenarios compared to fault-free networks.

4.3.5 Total time

Total times in faulty networks are presented in tables 12 and 13. In line with the observations from the processing time section, one observes:

- an increase in total time spent on fault scenarios
- less time spent after termination by highly faulty networks - except for 50 nodes and CM
- varied results for 5n scenarios and time spent after termination

None of these observations is suprising. Total time includes time spent by SafraFT as well as the basic algorithm. It is to be expected that faulty scenarios take more time because the faults need to be handled: the tree fixed and the token passed on at leat an extra round. Highly faulty networks become very small. Hence, SafraFT needs less time after termination to detect the fact because the token needs to cover only 10% of the network. As explained before, 5n runs naturally to varied results because they lead to wider range of situations (see section 4.3.1).

To conclude, the total times measurements are as expected. In particular, they show that SafraFT handles faults gracefully: the total times increase by a maximum of 4.14 times (partly due to the basic algorithm) and in most cases less time or a similiar amount of time is spent after termination to detect the fact.

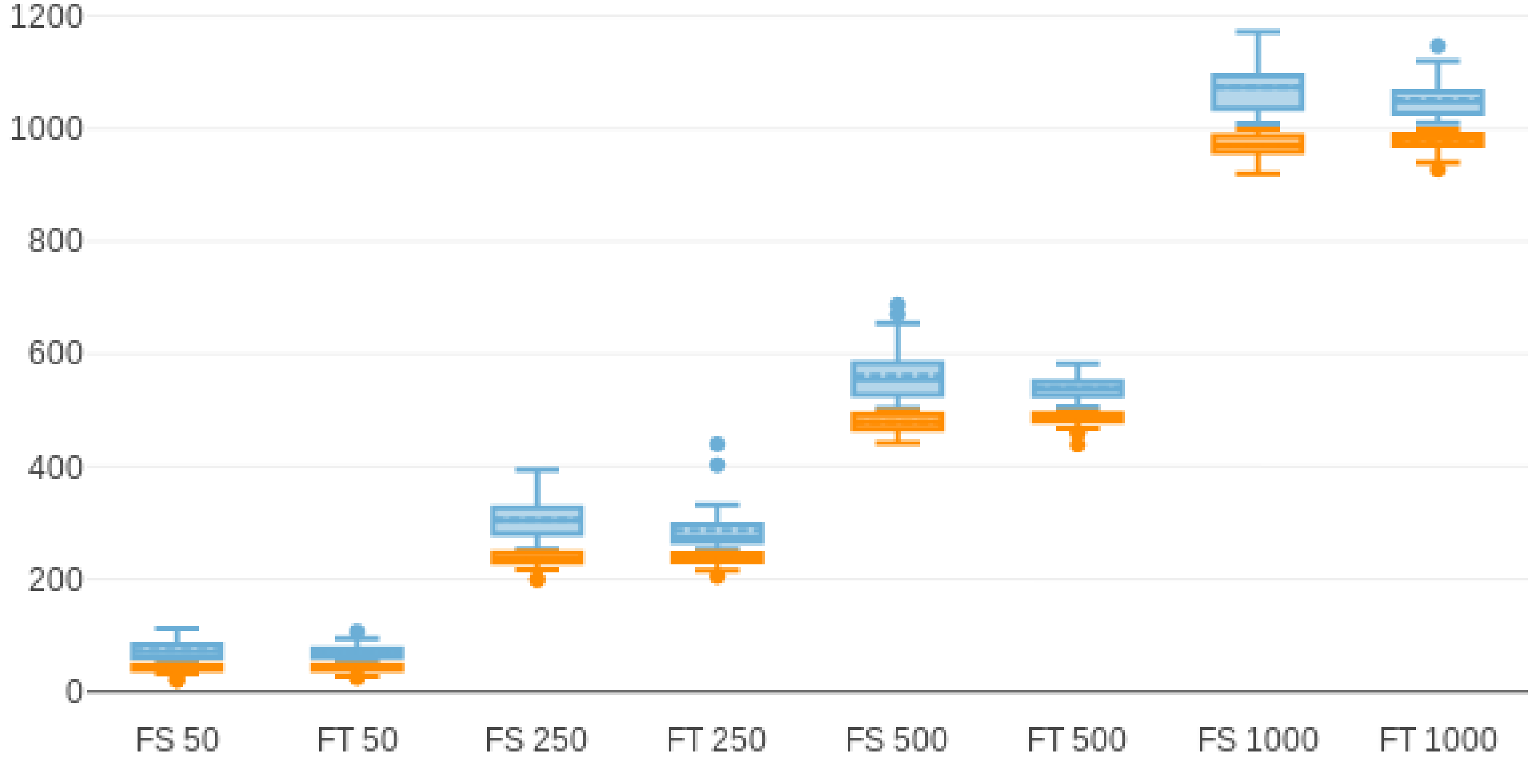

Figure 2: Tokens (blue) and tokens after termination (orange) for SafraFT and SafraFS in fault free runs for all network sizes with the basic algorithm Chandy-Misra.

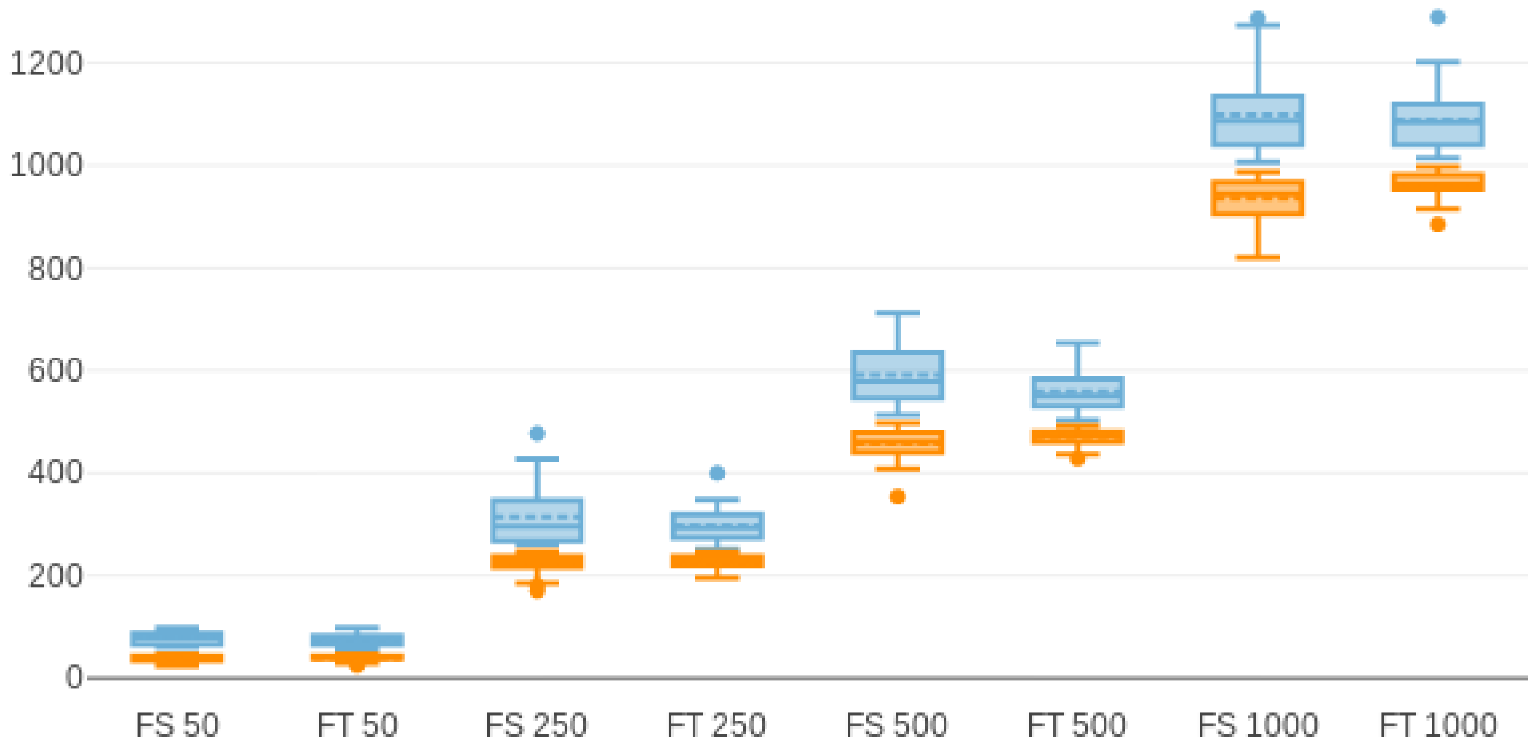

Figure 3: Tokens (blue) and tokens after termination (orange) for SafraFT and SafraFS in fault free runs for all network sizes with the basic algorithm Afek-Kutten-Yung.

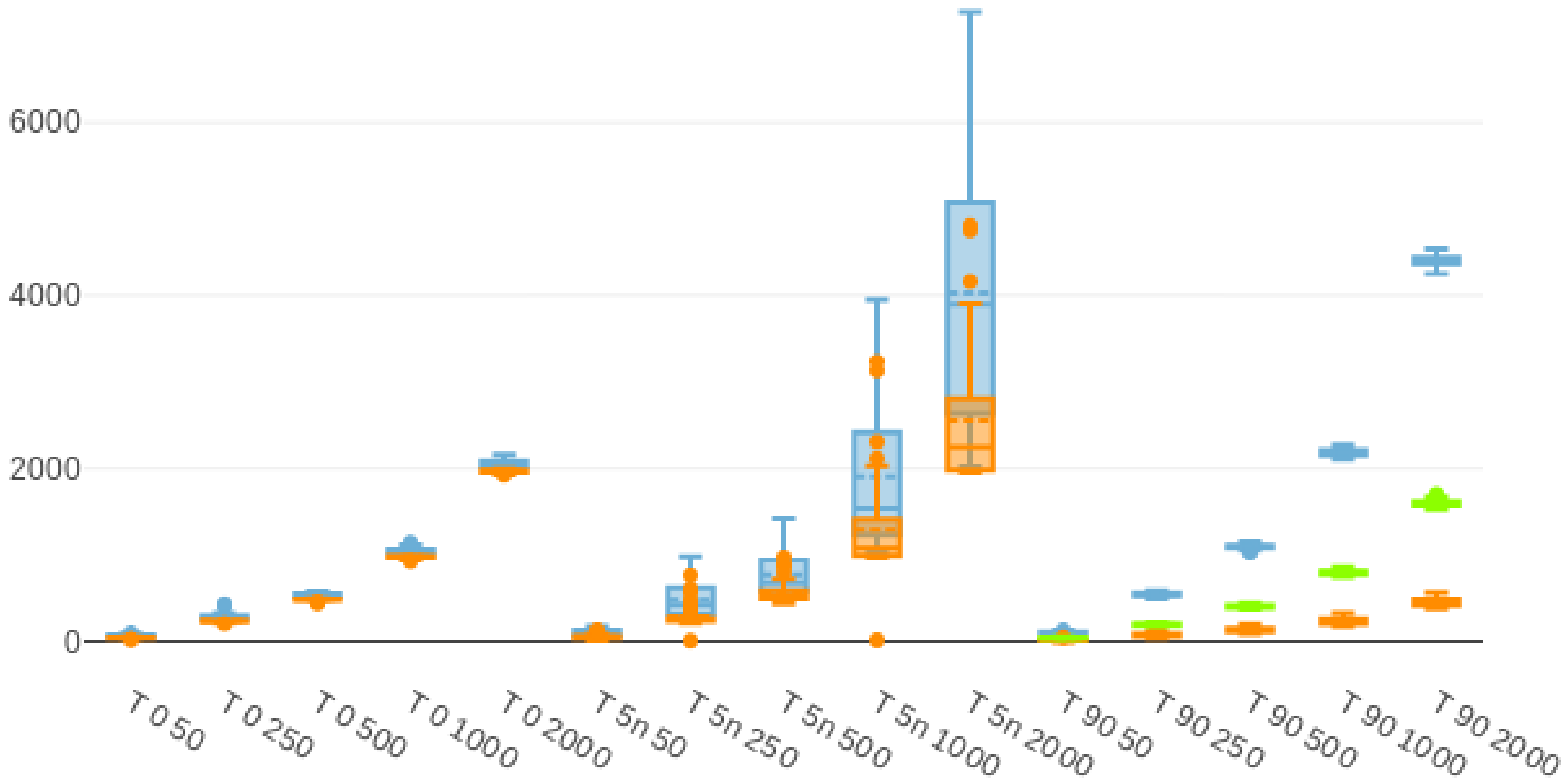

Figure 4: SafraFT and CM: Tokens (blue) and tokens after termination (orange) for SafraFT on the x-axis for 5 network sizes and 5n and 90% fault configurations on the y-axis. Backup tokens in green, shown only for 90%

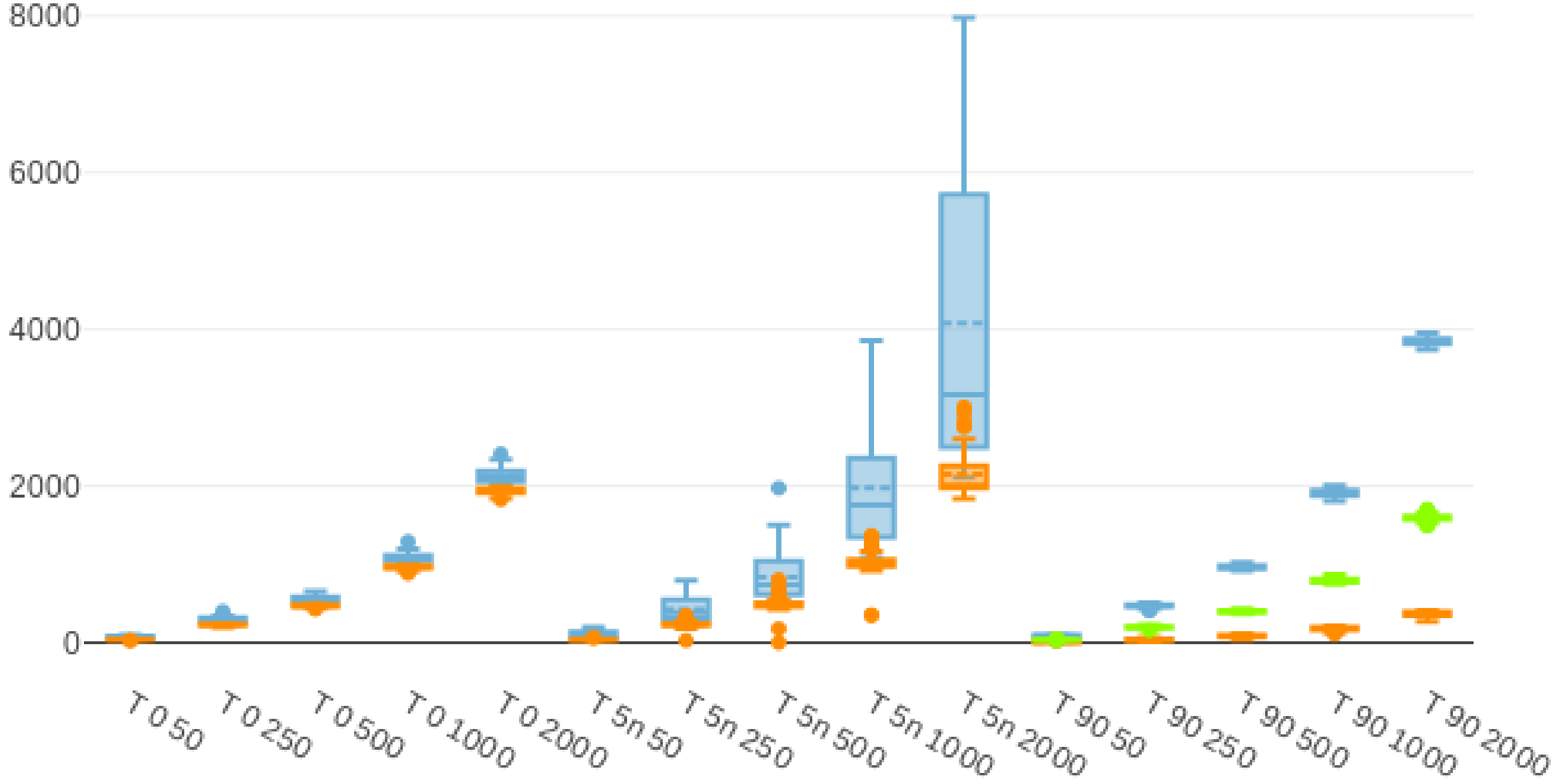

Figure 5: SafraFT and AKY: Tokens (blue) and tokens after termination (orange) for SafraFT on the x-axis for 5 network sizes and 5n and 90% fault configurations on the y-axis. Backup tokens in green, shown only for 90%

5 Discussion

First, this section concludes what the results imply about SafraFT. Next, I discuss limitations and improvements of my approach.

An important aim of the experiment is to show the correctness of SafraFT. Towards this aim, the results leave no doubt that SafraFT behaved correctly in all 2149 runs. These runs cover a wide range of different environments from small networks of 50 nodes to big networks with 2000 instances and for each network size the fault-free case, as well as two quite different fault scenarios. The basic algorithm is carefully chosen to exhibit an interesting and varied message pattern in terms of the number of messages and communication partners. Furthermore, Chandy-Misra leads to many changes between the active and the passive state of each node. The events on which to simulate node failure are deliberately chosen to put high pressure on the ability of SafraFT to recover faults and should trigger many different executions. All in all, the situations tested are designed to test SafraFT thoroughly under stress. Still, termination is detected in every case; with no case of early detection.

However, the experiment shows that the definition of termination used to develop SafraFT, although very common and widely accepted, does not cover all situations arising in practice. The definition assumes passive nodes only become active when they receive a basic message. In reality, nodes can also change to an active state when they detect the crash of another node and need to take corrective actions. When I realized this, I extended the definition of termination as described in section 4.1 and evaluated the results of the experiment according to the original definition and the extension. As stated above, SafraFT does not detect termination early according to the original definition. When the extended version is used, SafraFT detects early termination in only 14 out of 1124 runs with crashes. Such cases are unlikely in praxis because it is less likely there than in our experiment that a fault happens just before termination, because in our experiment many faults are triggered when a token is forwarded or received while in praxis most faults ought to happen within the basic algorithm or idle time because the vast majority of the time is spent for these purposes. Furthermore, one can actively reduce the chance of early termination detection by using a failure detector that propagates failures fast to all nodes and then enforces an extra token round. Finally, this is a general problem of termination detection in the presence of faults: when a basic algorithm can be activated by detecting a fault, this can happen directly before actual termination but then there is inevitably a short period of time in which termination can be announced because detection of the crash and propagation of this information cannot be instantly.

The practical comparison between SafraFT and SafraFS shows that there is no change in message complexity and confirmed the hypothesis that bit complexity raises linearly with the network size. This higher bit complexity is the reason why SafraFT takes longer to detect termination after actual termination and leads to a higher processing time overhead over the basic algorithm than for SafraFS. This processing time overhead is higher than we expected.

A possible explanation can be found in my experiment setup: each physical node simulates between 50 and 100 instances. Each of this instances uses multiple threads; altogether there are at least 4 times as many threads as cores on each machine. This leads to threads being preempted by the operating system (its stopped and control is given to another thread, e.g. one of another instance) which happens more often for threads that try to send big messages. This theory is supported by the fact that the effect is stronger for networks with 2000 nodes. However, experimental confirmation of this hypothesis could not be gained within the time limits of this project.

I would like to point out that my experimental setup imposes that the time taken by SafraFT to detect termination is long. This is because Chandy-Misra and Afek-Kutten-Yung complete their tasks in a few seconds and then SafraFT needs roughly the same time to detect termination. However, if I had chosen a basic algorithm that takes multiple hours to complete, the seconds taken by SafraFT to detect termination would be seen in a different perspective. Furthermore, the low processing time overhead (less than 13% in all runs of this experiment) of SafraFT plays a bigger role in long-running jobs.

Towards the performance of SafraFT in the presence of faults, I conclude that the number of backup tokens issued is not limited by the number of faults, but the average is lower than the number of faults. Also, the increase in bytes per token is only constant (5 to 8 bytes) for a few faults and increases by 1.29 times for 90% node failure. It is difficult to predict general relationships between the number of faults, network size and other measured metrics, e.g. time or token sent. Still, the experiments clearly show that even in big networks and with 90% node failure, the algorithm performs well and that none of the metrics shows exponential growth. An especially interesting metric is the number of tokens sent in faulty networks: it shows relative independence from the number of failing nodes. For both fault scenarios of five or fewer failing nodes and 90% node failure on all network sizes, at most 3 times as many tokens were sent as for a fault free run. This indicates that independent from the number of faults, one more token round is necessary to detect termination. To conclude, SafraFT handles faults in a sound and kind manner.

In conclusion, the experiments present strong evidence towards correctness of SafraFT, give a realistic comparison between the fault-tolerant version and the original version of Safra and demonstrate SafraFT's ability to handle faults. In the next paragraphs, I point out possible improvements for this project.

We would have liked to run repetitions with even bigger networks than 2000 nodes. This was hindered by the limited physical resources at hand - only 42 nodes with 8 cores each. Therefore, even for 2000 instance multiple of these had to run in parallel on the same core. That leads to slow runs for big networks and possibly unclear timing metrics.

The situation could be improved by:

- More hardware.
- More efficient use of the existing hardware, e.g. a more efficient implementation of the `MessageUpcall` class by using a producer/consumer scheme overall messages to greatly reduce the number of active threads.
- Compromising between a realistic setup and an efficient setup, e.g. by using only one Java and Ibis instance per physical node and different threads for all instances on that node, instead of processes.

IPL does not provide rather important primitives as broadcasting or barriers. So both had to be implemented by me during the project. Especially, implementing robust and performant barriers showed to be more work than expected, as one IPL feature (signals) breaks with 2000 nodes and it demonstrated to be impossible to connect each node to one master node to orchestrate actions. Hence, I recommend checking if a messaging library with a more comprehensive library exists, e.g. MPI or Akka[10].

With regardsd to showing the general relationship between network size and behaviour in the presence of faults is not the main goal of this project, I had hoped to come to more precise conclusions regarding this matter. To do so one would need to run the experiments on bigger networks and preferably concentrate on a specific but relevant fault scenario. Even with this additional data, it could prove very hard to attain general conclusions as faults do not only influence SafraFS but the whole system, e.g. a fault always leads to fewer nodes in the system and therefore changing the network size, or it leads to different behaviour of the basic algorithm which in term changes Safra's activity.

Finally, this project leaves the open question: how to define termination in the presence of faults? The commonly used definition seems to be fundamentally flawed by the fact that most fault-tolerant algorithms need to react to a fault. Therefore, they become active when they detect a fault. This is not trivially fixed because a fault can happen at any time and is not predictable. An interesting starting point for further research on this topic can be found in the literature about self-stabilizing algorithms, which use the notion of *final fault*. Note that neither the extended definition, which I proposed, nor the notion of *final fault* are practical solutions because they can only be checked after execution by offline analysis.

[10] An implementation of the Actor model for the JVM written in Scala

Appendices

A Buggy SafraFT version

This is the SafraFT version as proposed before these experiments. It includes two bugs which have been fixed in the current version [8].

Algorithm 1: Initialization$_i$

```
for $j = 0$ to $N - 1$ do
    $count_i^j \leftarrow 0$;
    $count_t^j \leftarrow 0$;
end
$black_i \leftarrow i$;
$seq_i \leftarrow 0$;
$\textsc{Crashed}_i \leftarrow \emptyset$;
$\textsc{Crashed}_t \leftarrow \emptyset$;
$\textsc{Report}_i \leftarrow \emptyset$;
$next_i \leftarrow (i + 1) \bmod N$;
if $i = 0$ then
    $black_t \leftarrow N - 1$;
    $seq_t \leftarrow 1$;
    HandleToken$_0$;
else
    $black_t \leftarrow i$;
end
```

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

```
if $j \notin \textsc{Crashed}_i \cup \textsc{Report}_i \cup \textsc{Crashed}_t$ then
    $seq_m \leftarrow seq_i$;
    $sender_m \leftarrow i$;
    $send(m, j)$;
    $count_i^j \leftarrow count_i^j + 1$;
end
```

Algorithm 3: ReceivedBasicMessage$_i$

```
$m \leftarrow \mathit{dequeue}(\textsc{MessageQueue}_i)$;
if $\mathit{sender}_m \notin \textsc{Crashed}_i$ then
    if $(\mathit{sender}_m < i \wedge \mathit{seq}_m = \mathit{seq}_i + 1) \vee$
     $(\mathit{sender}_m > i \wedge \mathit{seq}_m = \mathit{seq}_i)$ then
        $\mathit{black}_i \leftarrow \mathit{furthest}_i(\mathit{black}_i, \mathit{sender}_m)$;
    end
    $\mathit{count}_i^{\mathit{sender}_m} \leftarrow \mathit{count}_i^{\mathit{sender}_m} - 1$;
end
```

Algorithm 4: ReceivedToken$_i$

```
if $\mathit{seq}_t = \mathit{seq}_i + 1$ then
    $\mathit{update}(t)$;
    $\textsc{Crashed}_t \leftarrow \textsc{Crashed}_t \setminus \textsc{Crashed}_i$;
    $\textsc{Crashed}_i \leftarrow \textsc{Crashed}_i \cup \textsc{Crashed}_t$;
    HandleToken$_i$;
end
```

Algorithm 5: FailureDetector$_i$

```
$\mathit{crashed}(j)$;
if $j \notin \textsc{Crashed}_i \cup \textsc{Report}_i$ then
    $\textsc{Report}_i \leftarrow \textsc{Report}_i \cup \{j\}$;
    if $j = \mathit{next}_i$ then
        NewSuccessor$_i$;
        if $\mathit{seq}_i > 0 \vee \mathit{next}_i < i$ then
            $\textsc{Crashed}_t \leftarrow \textsc{Crashed}_t \cup \textsc{Report}_i$;
            $\mathit{black}_t \leftarrow i$;
            if $\mathit{next}_i < i$ then
                $\mathit{seq}_t \leftarrow \mathit{seq}_i + 1$;
            end
            $\mathit{send}(t, \mathit{next}_i)$;
        end
    end
end
```

Algorithm 6: NewSuccessor$_i$

```
$\mathit{next}_i \leftarrow (\mathit{next}_i + 1) \bmod N$;
while $\mathit{next}_i \in \textsc{Crashed}_i \cup \textsc{Report}_i$ do
    $\mathit{next}_i \leftarrow (\mathit{next}_i + 1) \bmod N$;
end
if $\mathit{next}_i = i$ then
    $\mathit{wait}(\mathit{passive}_i)$;
    Announce;
end
if $\mathit{black}_i \neq i$ then
    $\mathit{black}_i \leftarrow \mathit{furthest}_i(\mathit{black}_i, \mathit{next}_i)$;
end
```

Algorithm 7: HandleToken$_i$

```
$wait(passive_i)$;
$black_i \leftarrow furthest_i(black_i, black_t)$;
$\textsc{Report}_i \leftarrow \textsc{Report}_i \setminus \textsc{Crashed}_t$;
if $black_i = i \vee \textsc{Report}_i = \emptyset$ then
    $count_t^i \leftarrow 0$;
    for all $j \in \{0, \ldots, N{-}1\} \setminus (\textsc{Crashed}_i \cup \{i\})$ do
        $count_t^i \leftarrow count_t^i + count_i^j$;
    end
end
if $black_i = i$ then
    $sum_i \leftarrow 0$;
    for all $j \in \{0, \ldots, N{-}1\} \setminus \textsc{Crashed}_i$ do
        $sum_i \leftarrow sum_i + count_t^j$;
    end
    if $sum_i = 0$ then
        Announce;
    end
end
if $next_i \in \textsc{Crashed}_t$ then
    NewSuccessor$_i$;
end
if $next_i < i$ then
    $seq_t \leftarrow seq_t + 1$;
end
if $\textsc{Report}_i \neq \emptyset$ then
    $\textsc{Crashed}_t \leftarrow \textsc{Crashed}_t \cup \textsc{Report}_i$;
    $\textsc{Crashed}_i \leftarrow \textsc{Crashed}_i \cup \textsc{Report}_i$;
    $\textsc{Report}_i \leftarrow \emptyset$;
    $black_t \leftarrow i$;
else
    $black_t \leftarrow furthest_i(black_i, next_i)$;
end
$send(t, next_i)$;
$black_i \leftarrow i$;
$seq_i \leftarrow seq_i + 1$;
```

References

[1] Yehuda Afek, Shay Kutten, and Moti Yung. “The local detection paradigm and its applications to self-stabilization.” In: *Theoretical Computer Science* 186.1-2 (1997), pp. 199–229.

[2] Henri Bal et al. “A medium-scale distributed system for computer science research: Infrastructure for the long term.” In: *Computer* 5 (2016), pp. 54–63.

[3] Henri Bal et al. “Real-world distributed computer with ibis.” In: *Computer* 43.8 (2010), pp. 54–62.

[4] Mani Chandy and Jayadev Misra. “Distributed computation on graphs: Shortest path algorithms.” In: *Communications of the ACM* 25.11 (1982), pp. 833–837.

[5] Murat Demirbas and Anish Arora. *An optimal termination detection algorithm for rings.* Tech. rep. Technical Report OSU-CISRC-2/00-TR05, The Ohio State University, 2000.

[6] Edsger Dijkstra. “Shmuel safra’s version of termination detection.” In: *NATO ASI SERIES F COMPUTER AND SYSTEMS SCIENCES* 173 (1999), pp. 297–302.

[7] Wan Fokkink. *Distributed Algorithms: An Intuitive Approach.* The MIT Press, 2018. ISBN: 0262037661, 9780262037662.

[8] Wan Fokkink and Karlos George. *Fault-Tolerant Termination Detection with Safra’s Algorithm.* Tech. rep. Vrije Universiteit Amsterdam, 2018.

[9] George Karlos. “A Fault-Tolerant Variant of Safra’s Termination Detection Algorithm.” B.S. Thesis. Vrije Universiteit Amsterdam, 2016.