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

Technical Report

Per Fuchs

October 24, 2018

1 Introduction

This technical report presents an experiment using our fault tolerant Safra version (short SafraFT) [**safraFT2018**] to detect termination of a fault tolerant version of Chandy Misra's routing algorithm and Afek-Kutten-Yung's self-stabilizing spanning tree algorithm. The fault-tolerant Chandy Misra version is developed for this project as an extension of the fault sensitive Chandy Misra algorithm described in [**Fokkink:2018**] on page 57. An explanation of the extension and further important implementation decisions can be found in section 2.

The experiment aims to

- backup 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 [**demirbas2000optimal**]
- demonstrate the ability of SafraFT to handle networks of 25 and 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 send
- number of tokens send after the basic algorithm terminated
- number of backup tokens send
- 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 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.

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 [**karlos**].

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

2 Methods

This section describes the most important software design decission 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. For a description of the original algorithm refer to [**Fokkink:2018**]. The fault-tolerant version used for the experiments constructs a sink tree in an undirected network under the assumption that a perfect failure detector [1] 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 answer to `X`.

Next, I present an intuition why this extentsion is fault-tolerant. Each failure only affects nodes that see themselves as children, grandchildren or deeper ancestor 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 example based on a none 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 send 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 receive consider `A` there 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 this, it sends a `REQUEST` message towards `A` and `D` - call the latter `M2`. If `M2` overtakes `M1`, `D` resets its variabbles and on receive of `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

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

because the original problem is that `M2` overtook `M1`. [2]

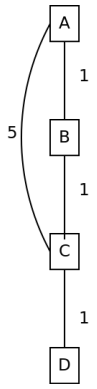

Figure 1: None FIFO network with `A` as root to demonstrate necessity of FIFO networks.

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, to fixing the last bugs, 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 implement Afek-Kutten-Yung's self-stabilizing spanning tree algorithm as described in [**fokkink:2018**]. The original Afek-Kutten-Yung is a synchronized algorithm for shared-memory environments. Therefore, it becomes extremely slow when it is used in a message passing environment with many nodes. Hence, I optimize it to allow runs up to 2000 nodes. The optimization exhibits infinite runs. In our experiment setup, these runs do not occur. Next, I explain the optimization, give a counter example to its correctness and explain why the infinite runs are unlikely in our setup.

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 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)

[2] 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 proof: 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.

– 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 an 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 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. [**Fokkink:2018**] 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 send through different channels than basic and Safra messages because otherwise they would arrive in FIFO order with them and this would exclude situations where a basic messag 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 1. 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 more closely.

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. 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. 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 decent sized networks (the number of machines and instances per machine can be found in **??**). 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 an undirected network. To generate more interesting runs I use weighted networks.

Our Safra version needs an undirected ring. For simplicity in my setup, this ring is part of the network the basic algorithm runs on. That means there is always an undirected 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 heavyweight 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 an undirected 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.

For the experiments, I am not interested in the relationship between network topology and the measured metrics because this kind of analysis seems too detailed to show the correctness of our Safra version or to compare our Safra version with the traditional Safra version. Therefore, the exact network topology is not recorded. Anyhow, I output the depth of resulting trees and the nodes per level for runs up to 500 nodes. This information is recorded to allow for quality checks and ensure the experiments were not run on trivial networks. Generating these statistics for runs with over 500 nodes turned out to be time-consuming. Hence, I decided not to generate them. However, if the network topology leads to none trivial, deep trees for 500 nodes and less, it seems highly unlikely that it would not for even more nodes. Furthermore, I verified that the topology generated for more than 500 nodes still leads to interesting cases by spot sample.

3 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 for SafraFT and SafraFS. SafraFT was also tested with 1 to 5 nodes failing per run (dubbed 5n) and with 90% node failure. **??** presents how many repetitions of each configuration were run.

The raw data including a manual on how to interpret it can be found TODO here.

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.SS

3.1 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 proof 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 1000 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

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 algorithm which could lead to an early detection of termination.

To verify if situations like this actually occur during the experiment, I analysed the logs generated according to the definition of termination used to develop SafraFT (see above) and the following definition:

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

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 17 out 1500 runs. However, due to the fact that repairing the sink tree after detecting a parent crash is quite fast and there is

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	0.01
250	3.327	0.123	0.235	3.7%	7.06%	0.028	0.104
500	11.18	0.256	0.636	2.29%	5.69%	0.057	0.336
1000	25.971	0.589	1.075	2.27%	4.14%	0.12	0.484
2000	122.707	1.428	2.663	1.16%	2.17%	0.233	1.047

Table 1: Total processing times (left) and processing times after termination (right) in seconds and overhead over Chandy-Misra caused by Safra in percent

a short time window to do so while the announce call propagates to all nodes, only in 12 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.

3.2 Comparision of Safra versions

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

The number of tokens sends in total and after termination is presented in **??** for CM and in **??** for AKY.

The key observation is that SafraFS and SafraFT behave highly similar. SafraFT sends slightly fewer tokens in average. Also, its results shows less variance. Most likely these differences are caused by implementation details and are not generalizable.

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 send 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 node network counts 8020 bytes. The growth can be described by $bytes = 4 * < networksize > +20$.

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 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 3.1) and the event of a node calling announce. On the other hand, there are basic, Safra and Safra after termination 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 1 and table 2 that SafraFT uses more processing time than SafraFS and much of the additional time is spent between actual termination and termination detection.

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	0.01
250	2.485	0.152	0.261	6.12%	10.5%	0.024	0.081
500	4.692	0.301	0.52	6.42%	11.08%	0.047	0.177
1000	13.806	0.676	1.46	4.9%	10.58%	0.114	0.699
2000	41.283	1.574	4.896	3.81%	11.86%	0.275	2.783

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

Network	Safra FS	Safra FT	Δ	SafraFS	SafraFT	Δ
50	0.148	0.161	1.09	0.017	0.025	1.47
250	0.34	0.466	1.37	0.1	0.217	2.17
500	0.586	1.064	1.82	0.214	0.655	3.06
1000	1.195	1.741	1.46	0.495	1.06	2.14
2000	2.483	3.747	1.51	1.05	2.31	2.2

Table 3: Wall times total (left) and after termination (right) for SafraFT and SafraFS in seconds with ratios run with CM

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 betweeen termination and termination detection. One can also see that the processing time increased more than twofold for networks of 1000 nodes to 2000 nodes. This is most likely due to the fact that 2000 node networks were simulated by running 125 instances instead of 50 instances per physical node.

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 a 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. Therefore, I believe 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. A possible explanation can be found in my experiment setup: each physical node simulates between 50 and 125 instances. Each of this instances uses multiple threads; alltogether there are at least 4 times as many threads as cores on each machine.

The huge quantitative effect could in part be caused by threads being preempted 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, confirmation of that assumption is not in the scope of this project.

The processing time **??** also presents 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 stays moderate with a maximum of 11.86% for 2000 nodes and AKY.

The same pattern of SafraFT using more time and reacting stronger to an increase in network size

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	0.1	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	5.375	7.383	1.37	1.704	4.191	2.46

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

is visible for total times in table 3 and table 4 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 measurments 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 overheads between

3.3 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.

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

3.3.1 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 token sent average. The data is presented fig. 3 and table 5.

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

Between 1 and 5 node failures 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 as runs without fails e.g. one failing node could easily cause an extra token

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	0.5
250	287	493	1.72	543	1.89	236	289	1.22	75	0.32
500	540	774	1.43	1101	2.04	486	557	1.15	142	0.29
1000	1052	2021	1.92	2190	2.08	977	1320	1.35	239	0.24

Table 5: Tokens in total (left) and after termination (right) for different fault scenarios compared to fault-free networks.

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), at the same time, a single failing node that is a leaf in a Chandy Misra sink tree and that crashes just before the call of announce 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.

The phenomenon explained in the last paragraph most likely causes the extremely high maximum of tokens sent and over linear increase of the average number of tokens in networks with 2000 nodes for 1 to 5 nodes failing.

Other than networks with one to 5 failing nodes, 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 less 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. 3) As mentioned above there is one exception: networks with 2000 cause more token to be generated in 5n networks than in networks with 90% node failure.

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.

The network size has no clear influence on the overhead in the number of tokens send before or after termination for the 5n scenario up to networks with 500 nodes. For bigger networks, more nodes lead to higher overheads.

The size of networks with 90% node failures is positively correlated to the overhead of tokens in total and has no influence on the overhead of tokens after termination.

3.3.2 Backup tokens

The average amount 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 were more backup tokens are issued than faults occur exists as well. This can be explained by my decision to have node failing 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.

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	1.0	2608	1.29
1000	4020	4027	1.0	5195	1.29

Table 6: Token size averages in bytes for both fault scenarios compared to token size with zero faults.

Network	No faults	5n	Δ	90%	Δ	No faults	5n	Δ	90%	Δ
50	0.034	0.058	1.71	0.109	3.21	0.01	0.015	1.5	0.012	1.2
250	0.235	0.276	1.17	0.497	2.11	0.104	0.1	0.96	0.035	0.34
500	0.636	0.705	1.11	1.142	1.8	0.336	0.298	0.89	0.076	0.23
1000	1.075	1.371	1.28	2.209	2.05	0.484	0.471	0.97	0.097	0.2

Table 7: Total processing times (left) and after termination (right) in seconds for different fault scenarios compared to fault-free networks.

3.3.3 Token size

The average token size increases with faults because the IDs of faulty nodes are propagated by the token (see table 6).

The influence on token size is constant 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 1.15.

3.3.4 Processing Time

The observations of this chapter are backed by table 7.

As for tokens, one can see an increase in processing times before termination under the presence of faults compared to fault-free runs. Which is no surprise because more tokens were sent.

For runs with 1 to 5 failing node, a higher variability in the processing times before and after termination becomes apparent (not shown in the tables of the report). Most likely this is because of the higher diversity of scenarios possible if some nodes fail as explained in section 3.3.1. As the processing time grows for this runs, so does the processing time after termination.

For highly faulty networks one observes results in line with the results for tokens in these networks. The variability for processing time before or after termination is not raised compared to fault-free networks. The processing time taken is even higher than the one for less faulty networks. Less processing time after termination is spent than for fault-free networks because fewer tokens need to be sent.

An interesting observation is that the difference between fault free processing time to processing time with faults does not necessarily increase with the network size: with 90% node failure the differences sink.

3.3.5 Total time

Total time in faulty networks is presented in table 8. In line with the observations from the processing time section, one observes:

- an increase in total time spent for fault scenarios
- a higher variability for time spent before and after termination for less faulty networks

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	0.03	1.2
250	0.466	0.584	1.25	1.006	2.16	0.217	0.223	1.03	0.091	0.42
500	1.064	1.237	1.16	2.263	2.13	0.655	0.624	0.95	0.18	0.27
1000	1.741	2.548	1.46	4.468	2.57	1.06	1.167	1.1	0.251	0.24

Table 8: Total times (left) and after termination (right) in seconds. For different fault scenarios compared to fault-free networks.

– less time spent after termination by highly faulty networks

For a network with 90% node failure, the overhead for total time decreases for network sizes between 50 and 1000 nodes and increases to its maximum for 2000 nodes. The overhead of time after termination sinks with higher network sizes.

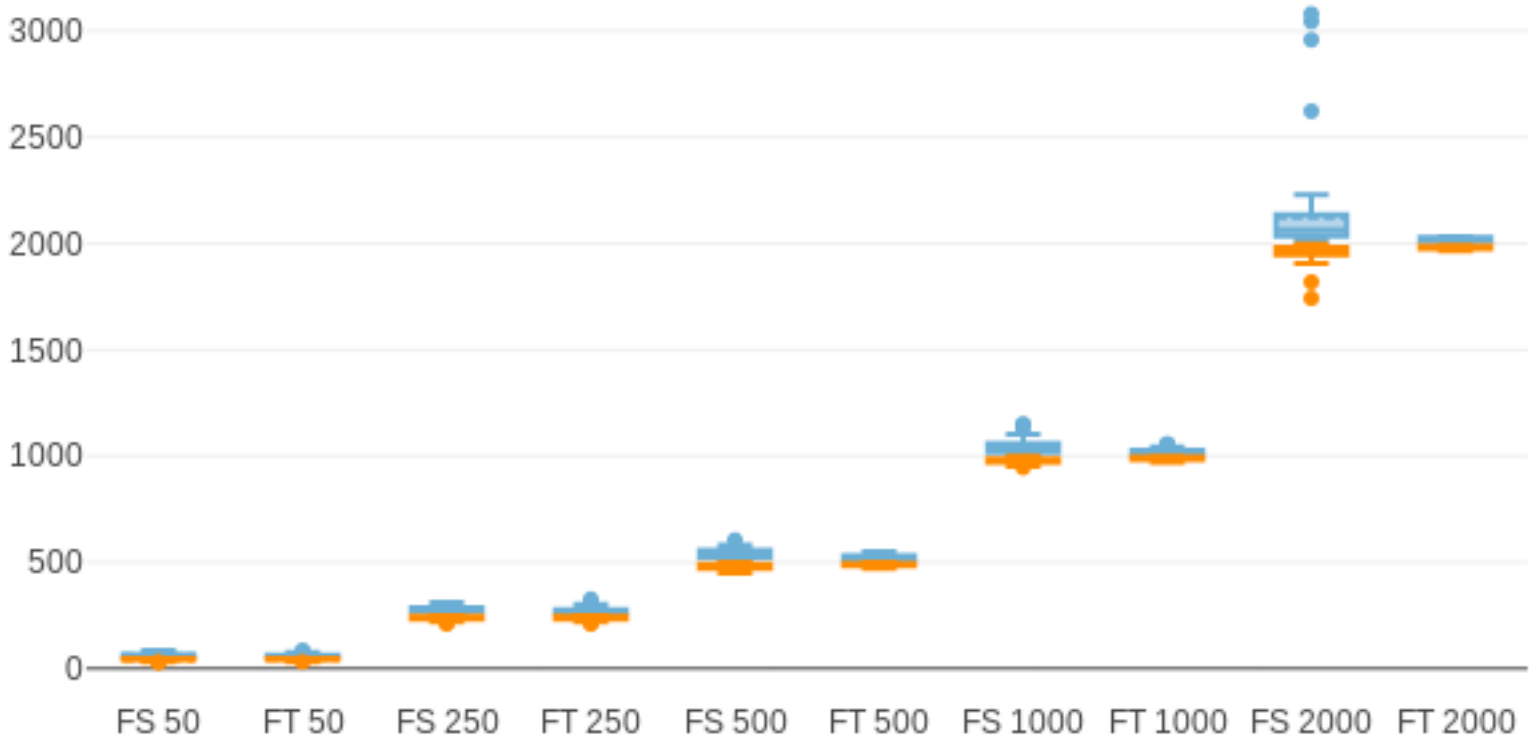

Figure 2: Tokens (blue) and tokens after termination (orange) for SafraFT and SafraFS in fault free runs for all network sizes with the basic algorithm Chandy-Misra.

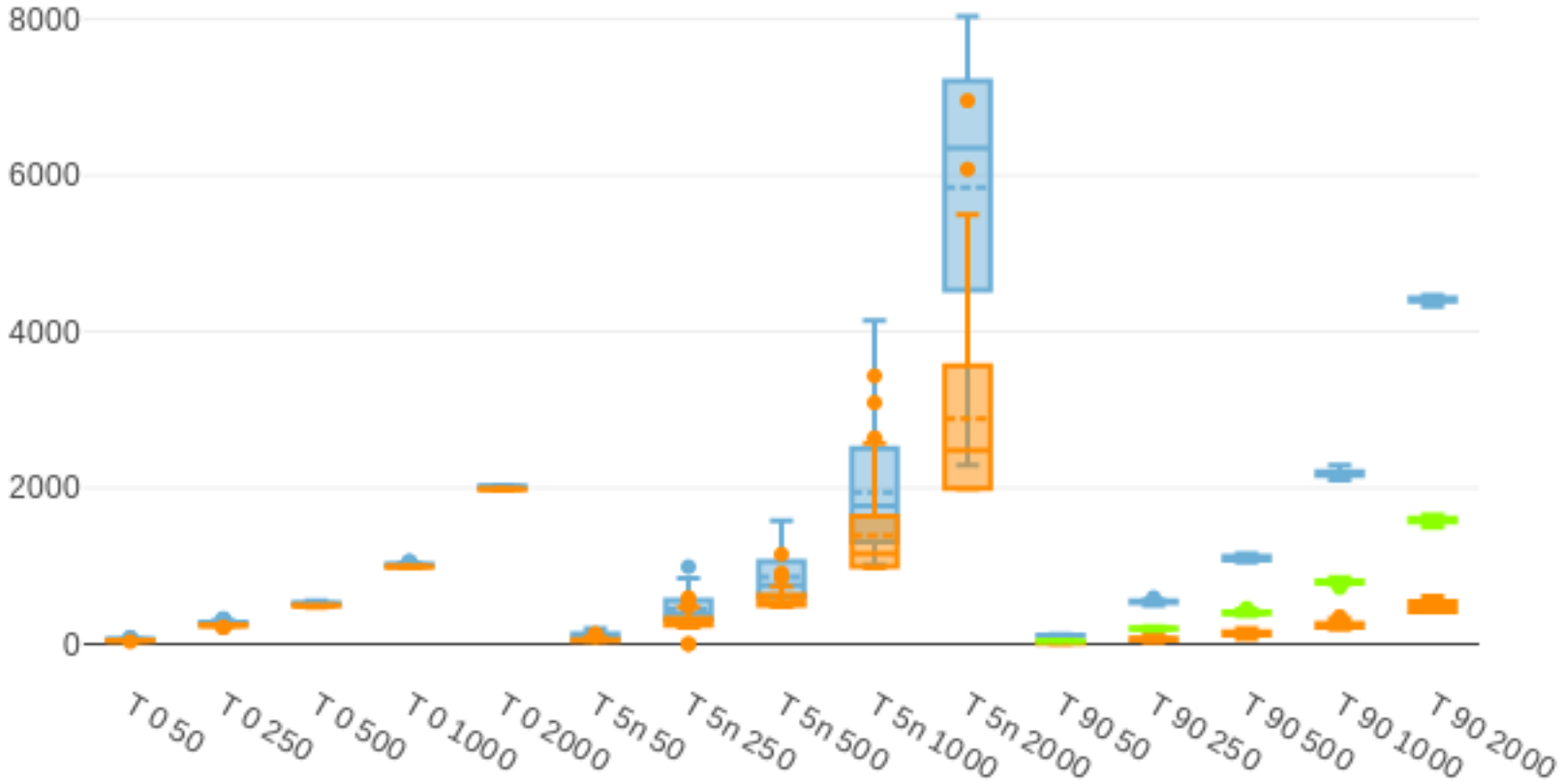

Figure 3: Tokens (blue) and tokens after termination (orange) for SafraFT on the x-axis for 5 network sizes and 5n and 90% fault configurations on the y-axis. Backup tokens in green, shown only for 90%

4 Discussion

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

The most important aim of the experiment is to show correctnes of SafraFT. Towards this aim, the results leave no doubt that SafraFT behaved correctly in all 1500 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 SafraFT ability to recover faults and should trigger many different configurations. 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 praxis. 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 and evaluated the results of the experiment according to that extension. This second evaluation showed that this does not happen often (17 out of 1000 runs) and only in 12 of these cases it leads to a corrupted result of the basic algorithm.

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 fault 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.

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.

I would like to point out that my experimental setup imposes that the time taken by SafraFT to detect termination is unreasonably long. This is because Chandy Misra completes its task in no time and then SafraFT starts the seemingly long process of completing a token round. 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 TODOof SafraFT plays a bigger role in long-running jobs.

Towards SafraFT performance 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 (8 bytes) for a few faults and grows only linearly with the network size 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 none of the metrics show exponential growth.

In total, the experiment reaches its goals, it presented strong evidence towards correctness of SafraFT, gives a realistic comparison between the fault tolerant version and the original version of Safra and demonstrates 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 lead to slow runs for big networks and serious scalability issues with IPL (the communication library used). Furthermore, 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) did break with 2000 nodes and it demonstrated to be impossible to connect each node to one master node to orchestrate actions. Hence, I recommend using more machines and a more robust communication library if one wants to use my implementation for bigger networks.

Although showing the general relationship between network size and behaviour in the presence of faults is not the main goal of this project, I 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.