Applying fault tolerant Safra to Chandy Misra

Technical Report

Per Fuchs

August 5, 2018

1 Introduction

This technical report presents an experiment using our fault tolerant Safra version (short SafraFT) (**ourpaper**) to detect termination of a fault tolerant Chandy Misra routing 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. The extension and further important implementation decissions 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 occured
- compare the performance of SafraFT to the fault sensitive Safra implementation (abbriviated SafraFS) from (**safraPaper**)
- demonstrate the ability of SafraFT to handle networks with 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 duration between termination of the basic algorithm and detection of the fact

All these metrics are measured within the following enviroments:

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

I do not aim to show the exact relationships between the dependent variables and the indepent variables e.g. the relationship between backup tokens send and the amount 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 can be used in practice and convince him that SafraFT does not perform worse than SafraFS in a fault free environment (except for its higher bit complexity). As well as, giving a good feeling for the trends of the relationships between environment and the measured metrics.

Before performing this experiment George applied SafraFT to a simulated, random basic algorithm in a multi threaded enviroment emulating a distributed system. These experiments showed strong evidence towards correctness and scalabilty of SafraFT. The implementation, technical report and results can be found here: (**georgework**). Also, they helped with my project as a reference implementation and as assurance that I would not run into serious implementation issues. So I would like to thank George for its groundwork of providing a running implementation of SafraFT.

The experiment presented here is performed as reaction to reviewers feedback to add ‘compelling end-to-end application’.

2 Methods

2.1 Chandy Misra

- src Distributed algorithms by Fokking - either write and connec this ro just reference this in the next chapter

2.2 Fault tolerant Chandy Misra

The fault tolerant Chandy Misra version used for our experiments constructs a sink tree in an undirected network under the assumption that a perfect failure detector is present at each node (a detector that does not suspect nodes that haven't actually failed and also will detect each failure eventually). Other than, the original Chandy Misra algorithm this version requires FIFO-channels. Furthermore, I assume that the root node cannot fail because otherwise there is no sink tree to construct.

As far as Chandy-Misra is concerned nodes are only interested in crashes of their parents and other ancessor on their path to the root node. If a node `X` detects a crash of its parent, it sends a `REQUEST` message to each neighbour. If a neighbour `Y` of `X` receives a `REQUEST` message, it answers with a `DIST d` messages where `d` is its own distance. To save message `DIST d` is only send if $d < \infty$. If `Y` happens to be a child of `X`, it resets its own `dist` and `parent` value to ∞ respectively $\perp$ and sends a `REQUEST` message to all its neighbours.

The requirement for FIFO channels is best understood by a counterexample example on a none FIFO network. The network used for this example is shown in fig. 1. Only important messages are mentioned; all other 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` message 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 TODO` 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 TODO. 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.

SafraFT uses the same FIFO channels as its basic algorithm during the experiment. Although, we still claim it does not need this property. This claim is straighforward to proof: SafraFT guarantees that at most one message is in any channel at all times because it forwards only a single token and when a backup token is issued, it is send to a different node than the original token and only the first of both tokens is forwarded afterwards. Hence, I do not see the use of FIFO channels during the experiment as a problem.

I argue that this augmented Chandy Misra algorithm constructs correct sink trees in presence of fail-safe failures. Each failure only affects nodes that see themselves as children, grandchildren and so on; that is, it only affects subtrees. Because the perfect failure detector guarantees that each node failure will eventually be detected at the children of the failed node, they eventually send `REQUEST` message to all their neighbours. The neighbours send `REQUEST` message 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 `REQUEST` message and reset their `dist` and `parent` values. Also all neighbours that receive a `REQUEST` message of a node that is not their parent, answer the `REQUEST` message with their current `dist` value. This allows nodes in the affected subtree to rebuild new

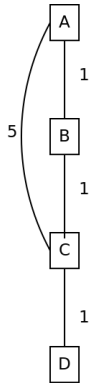

Figure 1: None FIFO network with `A` as root to demonstrate necessity of FIFO networks.

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. grand children of the crashed nodes), invalid paths are introduced - as these nodes might not be reached by any `REQUEST` message and therefore still believe they have a valid path towards root. These invalid paths are corrected when the grandchildren are reached by the `REQUEST` message of their parent because on receipt they send `REQUEST` messages which reset all nodes considering them parents. This possible behaviour of introducing invalid paths that are corrected later, might lead to a bad theoretical message complexity but did not hinder the experiments. This indicates that this behaviour is not often triggered in praxis.

The presented fault tolerant Chandy Misra algorithm could be improved by relieving the necessity for FIFO-channels, a more formal proof of correctness and a thourough complexity analysis.

Anyhow, my work suggests that the Chandy Misra version is correct because I run it more than times and compare the constructed trees to the expected trees which were determined offline. The algorithm yielded the correct sink tree every time.

2.3 Fault Simulation

I simulate faults by stopping Safra and the basic algorithm on the failing isntance. In particular, a crashed node does not send or reacts 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)
- 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 experiment that a reasonable amount of nodes have failed with regard to the planned amount.

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 a approach leads to less interesting failures because the vast majority of these failure would occure 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 raise to many different situations for our Safra versio to deal with. I deliberately decided against chosing special failure points with regard to the basic algorithm because this would be lead to less focussed 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 its neighbours. This leads to a realistic fault detection model because these messages are sent asynchronlously through the same channels as all other messages. Hence, closely imitating a perfect failure detector based on heartbeats. In particular, a fault is detected *eventually after* it happened as with every failure detector. `CRASH` messages are not broadcasted to all nodes because IBIS (the message passing library I used) does not provide broadcasting.

After running the experiments, I noticed a limitation due to my fault detection mechanism: configurations in which a message is received from a node after its crash was detected, do not arise because I use FIFO channels for all messages (see Fault Tolerant Chandy Misra **??**). This could be easily fixed by using dedicated channels for crash messages - in IBIS this would manifest in dedicated "crash message" ports on each node. Then receiving crash messages would be out-of-band with regular message and the order of receiption would depend on sending time, the Linux scheduler and IBIS implementation details. The last two points are the reason why I decided against that approach at first.

2.5 Offline Analysis

For the experiment, I meassure 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 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 for each message 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 similiar to the one Safra uses but a global view on the system achieved by offline analysis allows to detect basic termination immediatedly when it occurs.

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.

All processing metrics are archieved by the same principle: they are logged online when they occur and are grouped into total and after termination by analysing the logs. 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 announce call by Safra.

2.6 Environment

2.6.1 IBIS

IBIS is a Java-based platform for distributed computing developed by researchers of the Vrije Universiteit Amsterdam. Its suppart 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 was 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 was used. At time of the experiments the VU had 41 nodes with 8 cores per node. Therefore, multiple instances were run on each physical node. This is possible because Safra and Chandy Misra are both communication heavy with rather low processing and memory requirements.

2.6.3 Network Topology

The basic algorithm I use take a network topology to work on as input. They require 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 algorithms run on. That means no matter which network topology the basic nodes are running on, there is always an undirected ring available in the network topology.

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 heavyweight 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 predetermined failing nodes and add more connections between probably

disconnected nodes and the root. These channels are so heavyweight to avoid using them over 'regular' channels which might create highly similiar topologies with a lot of nodes connected to the root directly.

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 our metrics because this kind of analysis seems to detailed to show 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. These metrics are recorded in the final result to allow for quality checks and ensure the experiments were not run on trivial networks. Generating this 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 with 500 nodes and less, it seems highly unlikely that it would not with even more nodes. Furhtermore, I verified that the topology with more than 500 nodes generates still interesting cases by spot sample.

Algorithm	Network	Faults	Repetitions
SafraFS	50/250/500/1000/2000	0	40 each TODO
SafraFT	50/250/500/1000/2000	0	TODO
SafraFT	50/250/500/1000/2000	5n	TODO
SafraFT	50/250/500/1000/2000	90%	TODO

Table 1: List of all configurations run.

3 Results

Over 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 meassured 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. section 3 presents how many repetition of each configuration were run.

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

3.1 Correctness of SafraFT

The experiment is aimed to support our paper with a practical, correct application of our alogorithm. 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 to early, I use offline analysis (see **??**) to determine the point of actual termination and verify that detection happened after. All

However, the experiment revealed that the framework for termination chosen to develop SafraFT is not complete and does not cover all cases for my experiment 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. However, under 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 root. The idea described above leads to the following concrete scenario: lets consider the situation that all node are passive and no message are in the channel. In other words, the system terminated by our definition. Node `X` forwards the token 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` message, it might activate other nodes again.

To conclude, the definition of termination that our algorithm is build upon does not fully capture our choice of basic algorithm which could lead to an early detection of termination.

To verify if situations like this actual occur during the experiment, I analysed the logs generated according to both defintions of termination - the one used to develop SafraFT and the same definition but extended with the assumption that termination is postponed until the last crash leading to activity in the basic algorithm has been detected.

As stated above, SafraFT did never detect termination too early according to the first definition. According to the extended definition it detects termination to early in However, due to fact that repairing the sink tree 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 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 send in total and after termination is presented in **??**. The key observation is that SafraFS and SafraFT behave highly similiar. There is no notable difference between the medians, the variance and the ratio between total tokens and tokens after termination for the same network size - except for a unusual high variance in tokens after termination SafraFS for 1000 nodes and a high variance in total tokens for SafraFS for 2000 nodes As one would expect, the number of tokens seems to grow linearly with the network size. Note that the first network size is 5 times smaller than the second after the network size doubles for each run.

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 420 bytes; a token in a 2000 node network counts 16020 bytes. The growth can be described by $bytes = 8* < networksize > +20$.

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. These events are start of the Safra and basic algorithm until each instance is informed of termination for total time. Total time after termination is defined as the ammount of seconds between the actual termination (as defined in **??**) and the event of an node calling announce. On the ohter hand, there are basic, Safra and Safra after termination processing times (including the time needed to send of messages). These are the accumulated times all instances needed to process basic or Safra functionality. 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, time spent for logging (where the process did not execute or methods), 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 for 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 observer in table 2 that SafraFT uses much more processing time than SafraFS and most of this time is spent between actual terminatio and termination detection. Furthermore, one sees

Network	Basic	Safra FS	Overhead FS	Safra FT	Overhead FT
50	1.248	0.035 (0.014)	2.8	0.059 (0.026)	4.73
250	11.202	0.200 (0.069)	1.79	0.469 (0.228)	4.19
500	62.525	0.488 (0.128)	0.78	1.304 (0.740)	2.09
1000	185.131	1.057 (0.228)	0.57	3.812 (2.486)	2.06
2000	1190.031	3.100 (0.465)	0.26	13.794 (9.005)	1.16

Table 2: Processing times before and after termination (in braces) in seconds and overhead caused by processing Safa's methods in percent

that while SafraFS timing grow much slower than SafraFT timings and that the difference becomes bigger. This hints for a change in time complexity between the two versions.

A small subexperiment of excluding the time spent to send messages from the processing time revealed that most of this difference can be tributed to writting tokens onto the wire. As we know from the previous section, the number of token send does not differ between the algorithms. 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 change of time complexity. The time complexity would change because the increase in network size leads to more token being sent (as in SafraFS) but also to bigger tokens being sent.

Most of the additional time for SafraFT is spent after termination. By the hypothesis introduced in the last paragraph, this is explained by most tokens being sent after termination (see **??**).

The processing time table 2 also presents a comparision of the time spent for the basic algorithm and both Safra versions. Although SafraFT uses significantly more time, the overhead on the processing time stays moderate.

The same pattern of SafraFT using more time and reacting worse to an increase in network size is visible for total times in **??**. Again the majority of this time is spent after actual termination for the same reason as before. I would like to note that the low processing time overhead of Safra is not in contradiction to the large amount of wall time spent after termination. This seemingly opposing results arise from the difference between wall time and processing time: the basic algorithm is much more active in the beginning that when it accumulates a lot of processing time; while Safra causes a lot of idle time at the end when all processes wait for their predecessor to pass on the token.

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

3.3 Influence of faults

In the following paragraphs, I present and explain the data generated by runs under the presence of node crashes. I run to really 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, the 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
50	61 (44) // 0.72	102 (54) // 0.53	1.67 (1.23)	106 (13) // 0.12	1.74 (0.30)
250	268 (241) // 0.90	433 (305) // 0.70	1.62 (1.27)	544 (64) // 0.12	2.03 (0.27)
500	529 (488) // 0.92	976 (661) // 0.68	1.84 (1.35)	1086 (124) // 0.11	2.05 (0.25)
1000	1032 (985) // 0.95	1586 (1308) // 0.82	1.54 (1.33)	2174 (225) // 0.10	2.11 (0.23)

Table 3: Tokens before and after termination (in braces). For different fault scenarios compared to fault-free networks.

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. More so highly faulty networks more faults. The data is presented **??** and **??**.

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, as well as, tokens sent after termintaion. This seems reasonable as runs with failing node might lead to more different situations as runs without fails e.g. one failing node could easily cause an extra token when it leads to an backup token being issued and forwarded (this token is marked black until for a whole run), at the same time, a single failing node that is a leaf in a Chandy Misra sinke tree and that crashes just before the call of announce at the successor causes no further activity. The same example provides an idea 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.

A example crafted similiarly would explain the extremely high maximum of tokens sent and overlinear 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 similiar variance in tokens and tokens after termination as a fault free networks. A likely explanation is that the low survival rate of 1 out of 10 leads to less different scenarios than 995 nodes surviving as in the fault scenario treated 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 **??**)

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.

3.3.2 Backup tokens

The average amount of backup tokens sent for either fault simulation and every network size is lower than the number of faults. This is because 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. That result is suprising because the simulated fault detector in my setup detects faults in a really timely manner. Still, there are some runs were 1 to 5 nodes fail but no backup token is issued for all

Network	No faults	5n	Δ	90%	Δ
50	0.059 (0.026)	0.085 (0.031)	1.44 (1.19)	0.159 (0.013)	2.69 (0.50)
250	0.469 (0.228)	0.643 (0.274)	1.37 (1.20)	1.709 (0.066)	3.64 (0.29)
500	1.304 (0.740)	1.956 (0.967)	1.50 (1.31)	3.972 (0.165)	3.05 (0.22)
1000	3.812 (2.486)	5.296 (3.258)	1.39 (1.31)	8.333 (0.386)	2.19 (0.16)

Table 4: Processing times before and after termination (in braces) in seconds. For different fault scenarios compared to fault-free networks.

network sizes.

The other extreme were more backup tokens are issued than faults occure exists as well. This can be explained by my decission to have node failing after issuesing 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.

3.3.3 Processing Time

The observations of this chapter are backed by table 4.

As for tokens, one can see an increase in processing times before termination under the presents of faults compared to fault free runs. This increase is stronger for runs with more faults. Which is no surprise as these runs also produced more tokens.

For runs with 1 to 5 failing node a higher variability in the processing times before and after termination becomes apparent. A likely reasons for this has been explained in the **??**. 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 less tokens need to be send.

3.4 Total time

In line with the observations from the two sections before, one observes:

- an increase in total time spent which is more pronounced on more faulty networks
- a higher variability for time spent before and after termination for less faulty networks
- less time spent after termination by highly faulty networks

The absolute numbers and relative increases are presented in table 5.

Network	No faults	5n	Δ	90%	Δ
50	0.232 (0.077)	0.289 (0.072)	1.250 (0.940)	0.418 (0.023)	1.800 (0.300)
250	1.333 (0.665)	2.163 (0.680)	1.620 (1.020)	3.537 (0.122)	2.650 (0.180)
500	3.178 (1.699)	4.343 (1.848)	1.370 (1.090)	6.834 (0.280)	2.150 (0.160)
1000	7.124 (4.634)	9.308 (5.739)	1.310 (1.240)	12.696 (0.678)	1.780 (0.150)

Table 5: Total times before and after termination (in braces) in seconds. For different fault scenarios compared to fault-free networks.

4 Discussion

Analyse and discuss data

why does this agree with our hypotheses or not? point out and explain anomalies

Limitations of my approach improvements of my approach By FIFO channels plus fault detection via these channels the case that a node crash is detected before its last message was received is not covered backup token send event is generated even if the node crashes before many backup tokens might be generated because of the locality of my failure detector

What do my results imply about Safra?

Comparision to george experiments?

Our setup of an basic algorithm completing its work relatively fast and termination detection taking some time afterwards clearly shows a drawback of fault tolerant Safra. However, a system with a long running basic algorithm e.g. multiple hours, would put this times in a whole different perspective. Then the seconds taken to detect termination would be less of an issue and the moderate processing time overhead demostrated far more important.