Applying fault tolerant Safra to Chandy Misra

Technical Report

Per Fuchs

July 29, 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
- time spent processing Safra's procedures
- time spent processing the basic algorithm's procedures
- total time spent for the complete computation until termination has been detected

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.

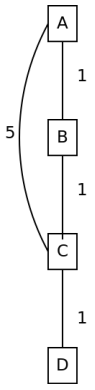

Figure 1: None FIFO network with `A` as root to demonstrate necessity of FIFO networks.

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 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 ran it more than times and compared the constructed trees to the expected trees which where 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. 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

We measure tokens before and after termination - define - tokens after termination - actual termination (not real point of time for termination) defintion: 1. No messages in the channels 2. All nodes are passive 3. Termination is postponed until all nodes that need to take action on any fault, detected the regarding fault. Only nodes which parent node fails need to take action and this might trigger further events in any other node. - detected termination

- needs means of determining termination independent and closer to actual termination than Safra - by logging token count, time spent events, active status change, message counter change (mean to determine no messages in the cahnnel) and parent crash detected - based on this detecting actual termination - by logging token count, time spent events. Dividing in before and after termination.

- Also using it for verification see **??**

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.

3 Results

* Metrics definition * How many runs * Independent variables * Dependend variables
Show results
- Verification - CM result compared to actual result - no warn and error messages - networks to be interesting

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

What do my results imply about Safra?

Comparision to george experiments?