Applying fault tolerant Safra to Chandy Misra

Technical Report

Per Fuchs

July 28, 2018

1 Introduction

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.
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 and IBIS implementation details. The last point is the reason why I decided against that approach at first.

2.5 Offline Analysis

2.6 Enviroment

2.6.1 IBIS

2.6.2 DAS 4

2.6.3 Network Topology

3 Results

* Metrics definition * How many runs * Independent variables * Dependend variables Show results

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
What do my results imply about Safra?
Comparision to george experiments?