A Fault-Tolerant Variant of the Mahapatra-Dutt Termination Detection Algorithm

Kristinn Björgvin Árdal

August 26, 2017

Contents

Abstract

In distributed systems it is important to know when a distributed algorithm has finished its computation. No single process can decide when an algorithm has terminated with only its local information. A termination detection algorithm is used to aggregate information to a process which can then declare termination shortly after the distributed algorithm has terminated.

Process crashes are inevitable in large distributed systems. Because of this, distributed algorithms that can operate correctly even in case of process crashes are being developed. Algorithms that have this property are called fault-tolerant algorithms. If we want to be able to detect termination for such algorithms, we also need a fault-tolerant termination detection algorithm. There are several nice algorithms for this problem, most notably Lai and

Wu's algorithm and Tseng's algorithm. We designed and implemented a new fault-tolerant termination detection algorithm that has similar performance to Lai and Wu's termination detection algorithm. We base our algorithm on an existing fault-sensitive termination detection algorithm by Mahapatra and Dutt.

Using a simulation environment we implemented our fault-tolerant termination detection algorithm. Implementation was done alongside designing the algorithm for quick testing of ideas. In addition we also implemented Lai and Wu's algorithm and used the simulation environment to compare the algorithms to each other, as well as to a fault-tolerant version of Safra's termination detection algorithm that had already been implemented in the simulation environment.

The simulations show how the different algorithms behave when either the number of processes in the network is increased or the number of process crashes is increased. This has different effects on the different processes, most notably there was a decrease in the number of control messages for our algorithm and a decrease in detection delay for Lai and Wu's algorithm with the introduction of crashes.

1 Introduction

Termination detection is a problem first proposed by Francez [5], and Dijkstra and Scholten [3] in 1980. Their formulation of the problem was simple: Francez described it in such a way that each process has a final state, and when all processes have reached their respective final states the computation has terminated. Dijkstra and Scholten imagined a graph where nodes in the graph can send messages to a successor and receive messages from a predecessor, and when no more messages are in transit the computation has terminated. The definition has been refined over the years and is now much more general, but before discussing this we have to define what a distributed system is.

A distributed system is a set of processes distributed over a network with a set of communication channels between them. In this paper we are only interested in fully connected networks, which means that there is a communication channel between each pair of processes. This is because only in fully connected networks it is guaranteed that failing processes never make the network disconnected. The processes are running a distributed algorithm for which termination needs to be detected. We will call this the *basic computation* and all messages sent by the algorithm *basic messages*. Matocha and Camp wrote a taxonomy of termination detection algorithms [11], where they defined a model for a distributed system. Each process can be either passive or active and their behaviour is dictated by the following rules.

1. Initially, each process in the system is either active or passive.
2. An active process may become passive at any time.
3. Only an active process may send a basic message to another process.
4. A passive process can only become active if it receives a basic message.

Termination is then defined as all processes being passive and no basic messages being in transit. Intuitively active processes are taking part in the basic computation while passive processes are waiting until someone tells them to do some work. When all processes have become passive, everyone is waiting for someone else to tell them what to do, and if there are no messages in transit no one will start working again.

Since termination is a global state of the system, no single process can declare termination with only local knowledge. The termination detection algorithm is there to gather information alongside the basic computation such that termination can be declared shortly after termination

has been achieved. A termination detection algorithm runs concurrently to the basic computation and ideally does not interfere with the basic computation. All messages introduced by the termination detection algorithm are called *control messages*.

The performance of a termination detection algorithm is measured using several metrics which need to be taken into account when choosing an algorithm. The two most common metrics are detection delay and message complexity. Detection delay is a measure of how long after termination has occurred, termination is declared by the control computation. Message complexity is a measure on the number of control messages sent. Bit complexity is also important, it measures the total number of bits sent by the termination detection algorithm, but for many cases this is simply the message complexity multiplied by a constant. For algorithms where the size of the messages is dependent on for example the number of processes in the system or the number of crashed processes, this metric might be important.

Some termination detection algorithms only work when the basic computation is a diffusing computation. This is where initially only one process is active. Alternatively algorithms can work when the basic computation is a non-diffusing computation. This is where initially more than one process can be active.

1.1 Fault-tolerance

Fault tolerance is defined as the ability to continue operating correctly in the presence of faulty processes. An algorithm that is not fault-tolerant is called fault-sensitive. As the number of processes in distributed systems goes up, the probability of a process crashing in a given time period increases linearly. Distributed algorithms running on a large number of processes for a long time are therefore very likely to incur a faulty process. Since fault-sensitive algorithms can not handle crashes, they would need to be restarted each time a process crashes. Fault-tolerant basic algorithms are therefore desirable as they would not need to be reset.

Algorithms may have different degrees of fault-tolerance, defined by the number of process crashes the algorithm can handle. A k-fault-tolerant algorithm can handle any k processes crashing.

We want to be able to determine when termination has occurred for fault-tolerant basic algorithms. A fault-sensitive termination detection algorithm would not help, because even though the basic computation would be able to finish correctly in case of a failure, the termination detection algorithm may not be able to continue operating correctly.

There are different types of process failures defined by different failure-models. The three most common failure models are fail-stop, fail-recover and arbitrary failure. In this thesis we use the fail-stop failure model, where when a process crashes it stops working completely. The fail-recover failure model is such that after a process has crashed it can recover and join the computation again, while the arbitrary failure model, after the failure happens the process can send arbitrary messages and receive messages from other processes. The arbitrary failure model is most often associated with hacked processes.

When a process crashes, the information that it has crashed may be sent to the other processes; this is done with a failure detector. For fault-tolerant termination detection a perfect failure detector is needed [13], which satisfies two properties:

- A process that does not crash during the run is never suspected to have crashed.
- Each crashed process is eventually suspected by every other process that does not crash during the run.

Each process is equipped with a perfect failure detector. It is easy to implement such a detector if there is a known upper bound on messages delay. In that case heartbeat messages can be

used to know when a process has crashed. Each process sends a heartbeat message to let other processes know that it has not crashed, if the heartbeat message is late by more than the upper bound on message delay it is safe to say that the process has crashed.

1.2 Objectives

Using the Mahapatra-Dutt [10] fault-sensitive termination detection algorithm, we derive a novel fault-tolerant termination detection algorithm. The fault-sensitive algorithm is a tree algorithm where a tree called the active tree, which includes all active processes, is maintained. In the algorithm a static tree structure is decided on before the algorithm starts. The active tree is built such that it is always a subset of the static tree structure. This makes it possible to have an efficient active tree with regard to detection delay. The fault-tolerant variant is a non-blocking fault-reactive algorithm that fixes the static tree structure when crashes occur. This is done by re-attaching the processes that have become detached from the static tree back to the root of the tree. Additional information is then sent to the root process to make sure that it knows when all processes have re-attached. To minimize the number of control messages the information is sent with other control messages as needed. The algorithm is $(n-1)$-fault-tolerant, which means that as long as there is at least one process still running the algorithm can detect termination. The small detection delay is an important factor of the fault-sensitive algorithm and this is preserved when making the fault-tolerant variant.

We implement both the fault-sensitive version as well as the fault-tolerant version of the algorithm in a simulation environment created by G. Karlos [6]. One objective is to compare different termination detection algorithms in a practical setting, as most are only compared at a theoretical level. Another objective is to see if and how having a simulation environment helps in developing a fault-tolerant distributed algorithm. Lai and Wu's algorithm is a tree based fault-tolerant termination detection algorithm and therefore has similar properties to our algorithm, this was the reason it was chosen as an algorithm to compare to.

Using the simulation environment we compare the performance of several distributed termination detection algorithms, fault-sensitive and fault-tolerant. Specifically we measure the detection delay and the message complexity. We compare the metrics we achieve using the simulation environment with the expected values for each algorithm. This is to see if the simulations are giving us reasonable values. Discrepancies give us insights into how the simulation environment can be improved, or possibly that expectancies were wrong.

This thesis is structured as follows. First we discuss how other authors have created fault-tolerant termination detection algorithms. After that the Mahapatra-Dutt algorithm is explained in detail. This leads into how the algorithm was transformed to being fault-tolerant where some ideas from other authors are used. Additionally, a pseudo-code is given for the fault-tolerant version. The simulation environment is then explained in detail. After that we show simulation results of all algorithms implemented and discuss the results of the simulations. In the discussion, improvements on both the fault-tolerant version of the Mahapatra-Dutt algorithm and the simulation environment are given. Lastly we have the conclusion where the results are summarized and future work is discussed.

2 Achieving Fault-Tolerance

There already exist fault-tolerant termination detection algorithms achieving fault-tolerance with many different methods. There are also sub-classes of fault-tolerance; a natural split in the algorithms is into fault-preventive and fault-reactive algorithms. Fault-preventive algorithms do work before any process crashes, to make sure that when a crash occurs it is handled properly.

Fault-reactive algorithms however pay the price upon a crash. Often the message complexity of fault-preventive algorithms is dependent on the degree of fault-tolerance, while for fault-reactive algorithms it depends on the number of crashes during the run.

The first fault-tolerant termination detection algorithm is Misra's algorithm [12]. This algorithms is fault-preventive and achieves fault-tolerance by having multiple tokens going through the network at the same time. The number of tokens in the network determines the degree of fault-tolerance and is therefore determined beforehand with the degree of fault-tolerance in mind. Having k tokens in the network multiplies the number of control messages by k.

Various other algorithms have been proposed since then and we look at three of them: the Lai and Wu algorithm, Mittal et al. and Lifflander et al. [8, 13, 9] The reason we look at these algorithms is that both Lai and Wu and Lifflander et al. base their algorithms on the Dijskra-Scholten termination detection algorithm [3] and Mittal et al. defines a general transformation of fault-sensitive termination detection algorithms to fault-tolerant termination detection algorithms.

Lai and Wu [8] were the first to propose a fault-reactive termination detection algorithm. They based their algorithm on the Dijkstra-Scholten termination detection algorithm [3] as previously stated and identified three issues that needed to be taken into account for the algorithm to become fault-tolerant. The first problem has to do with the root of the tree crashing. The root of the tree takes care of declaring termination in the algorithm and as such another process should take its place. When this happens all processes need to agree on the new root. Lai and Wu fix this issue by having a deterministic algorithm for choosing a new root, each process has a unique identity and the process with the lowest identity that has not crashed becomes the new root. The second issue is that when a process crashes the other processes need to know whether there are any undelivered messages from the crashed process. To fix this issue they make an additional assumption on the network such that any communication channel in the network can be flushed. Flushing a communication channel means to deliver all messages that are in transit in that channel. For processes to be able to do this a new *return-flush* message is introduced. When sent to a process this messages is returned to the sender by the network when the communication channel between the two processes has been flushed. This way the process also knows when the channel has been flushed. The third problem is that a crashed process might disconnect the tree. This is solved by flattening the active tree and making all processes join the active tree again. This is the biggest drawback of the algorithm as the amount of work needed to be done by the root when a crash occurs is in the worst case linear with the size of the network. Since the number of crashes is also linked to the size of the network this algorithm does not scale to very large networks. There is an additional condition for declaring termination which is that all processes need to have reported the same set of crashed processes to the root. The set is sent with the acknowledgement messages.

Lifflander et al. [9] took a different approach: instead of tolerating all kinds of crashes they only try to recover from ones that are most likely to occur in real systems. As with Lai and Wu's algorithm this algorithm is based on the Dijskra-Scholten termination detection algorithm. They implement three different extensions, Indep, RelLazy and RelEager, which provide different levels of fault-tolerance. Fault-tolerance is obtained by keeping some network information at each process. Indep only handles independent failures; processes that do not communicate with each other are called independent. Each process keeps track of which process is its grandparent and the tree is fixed by making processes whose parent crashed reattach to their grandparent. Dependent failures are fixed using RelLazy and RelEager only if possible, otherwise the algorithm states that an unrecoverable failure has occurred. RelEager aims to fix the tree as quickly as possible by increasing the amount of network information kept at each process. Instead of just having information about a processes grandparent the algorithm also has each process

keep a list of possible grandchildren. This algorithm scales very well to large networks because all fixes done to the tree structure are done at a local level. The drawback of this algorithm is however that not all crashes can be recovered from and is in fact not even 1-fault-tolerant as root crashes are not addressed.

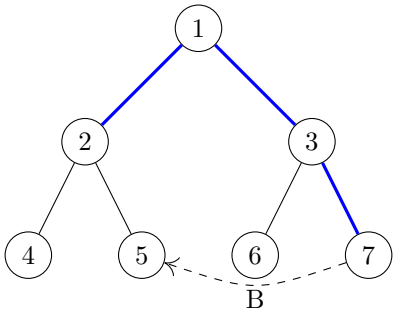
Mittal et al. [13] proposed a transformation to make fault-sensitive termination detection algorithms fault-tolerant. The idea of the transformation is roughly to run the fault-sensitive termination detection algorithm until a crash occurs. At that point the processes stop the current instance of the fault-sensitive algorithm and start a new instance of the termination detection algorithm on the set of working processes. Failure detector types can be ordered from weakest to strongest where a stronger failure detector type can detect all failures that a weaker failure detector type can. Mittal et al. proved that a perfect failure detector is the weakest type of failure detector needed for fault-tolerant termination detection. The fault-sensitive termination detection algorithm has to be able to handle a non-diffusing computation and work for a fully connected network. This transformation creates a non-blocking fault-reactive termination detection algorithm which can handle any number of crashes as long as one process remains active.

As stated previously there exist other fault-tolerant termination detection algorithms. See [14] for a literature study. Additionally, we have a fault-tolerant variant of Safra's algorithm [6] which has currently not yet been published, but is used in this thesis.

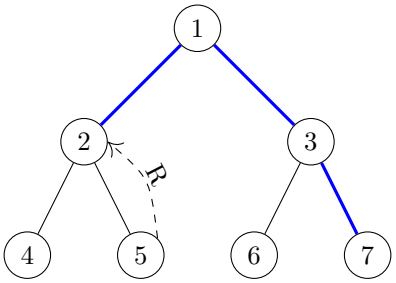
Safra's algorithm is a token ring-algorithm. This is where a token is passed around the network in a pre-determined ring structure gathering information about the network. The algorithm is centralized, there is a leader process that sends out the initial token and when it receives it back decides whether or not termination has been reached. Each process keeps track of the number of basic messages it has sent minus the number of basic messages it has received. Additionally, each process has a color, white or black. Initially each process is colored white and when a process receives a basic message it changes its color to black. The token goes around the network, only being sent forward when the process holding the token is passive. The token sums the value of all the basic messages sent minus the number of basic messages received. The token also has a color, it starts out as white but changes to black when it reaches a black process. Each process is changed to white by the token before it is sent from it. Termination can be declared by the leader when the sum of basic messages sent minus the number of basic messages received is equal to zero, and the token is white upon reaching the leader process again.

The improved version of Safra's algorithm has two big improvements. The first is that not all basic messages need color the process that receive it black, only in certain cases does this apply. The second is that the algorithm is made decentralized. To do this a change is needed in the way processes are colored. Instead of only coloring itself, a process can color all processes up to a certain process in the direction of the ring structure. The token now keeps track of which process is the farthest away black process instead of having a color of its own.

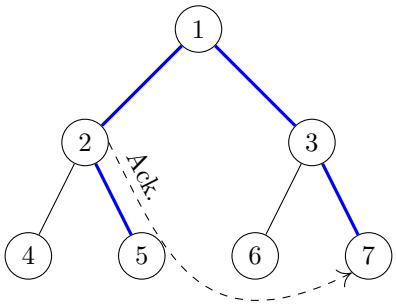
The fault-tolerant version is based on the improved version of the algorithm, but a few adjustments are needed to make it fault-tolerant. In case of a crash, the ring structure needs to be recovered. Each process keeps track of their successor, and when a process is notified that its successor has crashed, a new successor is computed. When a process crashes, its value for the number of sent messages minus the number of received messages cannot be accessed anymore. Because of this, processes keep track of the number of sent basic messages minus the number of received basic messages per process. That way when a process has crashed, its value can be disregarded. This means that each process has $n - 1$ counts to keep track of. The token has n counts, one for each process. Intuitively, these counts represent the total number of basic messages sent to or from a process that have not yet been delivered. When all basic messages have been received and all processes are passive, the value for each of these counts in the token will eventually become zero for all running processes.



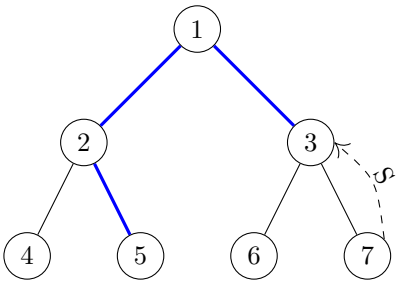
(a) Basic message is sent from process 7 to process 5.



(b) Resume messages is sent from process 5 to its parent process, 2.



(c) Acknowledgement message send from process 2 to 5 which sends it further to process 7



(d) Stop message sent from process 7 to process 3, its parent process.

Figure 1: Active tree is represented with a thick solid blue line and static tree is represented by all solid lines. Messages being sent are represented with dashed lines.

3 The Mahapatra-Dutt Algorithm

3.1 Fault-sensitive algorithm

We base our fault-tolerant algorithm on the static tree termination detection algorithm by Mahapatra and Dutt [10]. This algorithm uses so-called static tree structure covering the entire network that has been decided upon before the start of the algorithm. In their paper they use slightly different terminology compared to what we have seen before; passive processes are now called idle processes and active processes are called busy processes. Processes are either in a so-called active tree or they are not, if they are, the process is said to be active, otherwise inactive. A process is said to be loaded if it is busy or it has sent basic messages that have not been acknowledged by the control algorithm, otherwise it is said to be free.

The algorithm has a predetermined tree structure which determines how the active tree grows. The active tree is a subset of the static tree but grows and shrinks throughout the computation. Termination is declared by the root when it is the only process left in the active tree and it is free. An acknowledgement is required for every basic message; this acknowledgement is sent after the process receiving the basic message has joined the active tree. Before leaving the active tree, a process must have no children in the active tree and be free, meaning that all basic messages sent by it have been acknowledged. This is done to make sure that when a process leaves the tree, all processes that were made busy because of it have joined the active tree. Termination

can therefore not be declared before they leave the active tree as well.

The algorithm can handle non-diffusing computations because the active tree starts with all the processes in the network. To leave the active tree a process sends a STOP message to its parent process. To join the active tree again a process sends a RESUME message to its parent in the static tree. Both RESUME messages and basic messages need to be acknowledged. This is done with an ACKNOWLEDGE message in both cases.

Intuitively this algorithm works by making sure that any busy processes will join the active tree before the process that sent the basic message that made it busy leaves the active tree. When the active tree has disappeared, all processes have become idle and all basic messages have been sent. At that point, termination can be declared.

Figure 1 has an example sequence of messages being sent. In figure 1a we assume that processes 2 and 7 are busy but have not sent any basic messages that have not been acknowledged. Process 7 starts by sending a basic message to process 5 and then becomes idle. When process 5 receives the basic message it becomes busy but can not acknowledge the basic message yet since it is not in the active tree. Process 5 proceeds to join the active tree by sending a RESUME message to process 2, as is shown in figure 1b. Since process 2 is already in the active tree, it sends an ACKNOWLEDGE message back to process 5. When process 5 receives the message, it knows that it has joined the active tree and sends an ACKNOWLEDGE message to 7. This step is shown in figure 1c. Finally process 7 has received an acknowledgement for its basic message and may leave the active tree, as shown in figure 1d. Only leaf processes in the active tree may leave the tree, so after this processes 3 and 5 are the only ones that may leave the tree.

With this example you may have spotted that the processes need to have some extra information; process 5 has to know that it should send the ACKNOWLEDGE message to process 7. This information could be kept at the node but Lai and Wu add this information to the RESUME and ACKNOWLEDGE messages. Each RESUME and ACKNOWLEDGE message has information about the sender and receiver of the basic message that induced the control messages.

The algorithm supports non-FIFO ordering of messages. So a STOP message might arrive before the RESUME message that is supposed to add the process to the active tree. This is handled by counting the difference in the number of STOP and RESUME messages sent by each child. If one more STOP message has been sent, the node is not a part of the active tree.

This algorithm has some nice properties. The worst-case detection delay is $\Theta(D)$ where D is the depth of the tree. The worst-case message complexity is $\Theta(MD + N)$ and the average message complexity is $\Theta(M + N)$ where M is the number of basic messages, D is the depth of the tree and N is the number of processes. We would like to keep these metrics low after the conversion to a fault-tolerant algorithm.

3.2 Fault-tolerant algorithm

The idea behind the fault-tolerant variant of the Mahapatra-Dutt algorithm is to fix the static tree when a crash occurs and continue with the algorithm. We introduce new variables that are present in each process. Each process keeps track of a set of known crashed processes; this is updated whenever the node is notified of a new crash by the failure detector. Additionally each node keeps track of the number of crashed nodes in a variable we shall call KC; the value of this variable is attached to all messages, both basic and control. The value of this variable is updated whenever a message is received with a higher KC-value attached to it or when a crash notification is received and the size of the set of crashed processes is greater than the current value. Lastly we add a list of variables, CHILDCOUNTS, used to calculate the size of the tree. When a STOP message is sent, the sum of CHILDCOUNTS plus one is sent; this should equal the size of the subtree rooted at that process. Each process keeps track of the reported size of the

tree from each of its children. We define a new condition for declaring termination, which is that the size of the tree is equal to the number of nodes that have not crashed.

While designing the fault-tolerant variant, we ran into a few issues. These issues are the same ones that Lai and Wu [8] ran into when making the LTD variant of the Dijkstra-Scholten algorithm fault-tolerant. This was to be expected since both algorithms are tree algorithms. We however solve them in a slightly different way.

The first issue comes from the fact that the algorithm is a centralized algorithm where the root of the tree is in charge of declaring termination. If this root crashes, a new process has to take its place. We solve this in the same fashion as Lai and Wu: each process has a unique process identifier and the node with the smallest identifier becomes the root. Since this can be calculated by each process independently, no communication is needed to decide this.

The second issue is that when a crash occurs, all children of the crashed process have become disconnected from the tree. We decided on having all the children of the crashed process become children of the root. We have devised the algorithm in such a way that the processes could in theory attach to nodes other than the root, but for the implementation this is the simplest design. Here we ran into the problem that the root has to know when all processes have reattached to the tree. For this reason we introduced the ChildCounts variables. The size of the tree has changed and therefore needs to be re-calculated. To do this the active tree is reset, meaning that all nodes become part of the active tree and the ChildCounts variables are reset, setting it to 0 for each child. This way the size of the tree is counted again with the Stop messages. The idea of resetting the active tree comes from Mittal et al. [13] but can also be seen as an adaptation of how Lai and Wu [8] fix the issue. We make sure every process is part of the active tree and information sent to the root lets it know whether or not all processes have reattached to the tree.

The way we reset the active tree is that the count for the difference in Stop and Resume messages is set to zero for all children. Additionally only Stop and Resume messages where the value of KC sent with the message matches the value stored in the process are accepted. This way those types of control messages sent before resetting the active tree do not interfere. All basic messages that were sent before the crash and had not yet been acknowledged still need to be acknowledged, except those sent to a crashed process. Since a process joins the active tree as soon as it knows that a crash has happened it can also send all acknowledgements it had not already sent. All acknowledgements that a process had yet to send because of Resume messages can also be disregarded. The reason we can do this is that eventually all processes will know of all crashes that have occurred and therefore will have joined the active tree, making an acknowledgement for joining the active tree redundant.

We split up the Acknowledge messages into two distinct messages, one for acknowledging basic messages, MAck, and one for acknowledging resume messages, RAck. Additionally, we make every process keep track of all basic messages and Resume messages it has received and not yet acknowledged. This is done to be able to send acknowledgements for basic messages immediately when a process is notified that a crash has happened. Additionally, acknowledgements for Resume messages sent before the process crash can be thrown away since all processes will eventually become part of the active tree when they are informed of the process crash.

The third issue we encountered is what to do with messages still in transit from a crashed process. This is simply solved by making every process block all further messages from a process as soon as they are notified of it crashing.

We add an additional condition on both leaving the active tree and declaring termination, that KC equals the number of crashed processes known by the process in question. This is to make sure that the process knows of all changes that have been made to the tree structure before leaving the tree. Specifically it has to know if its parent has crashed, as the Stop message should then be sent to another process.

List of variables in each process:
$nnodes$ ▷ Number of nodes in the distributed system
ID ▷ Unique process identifier, $[0, nnodes - 1]$
$root$ ▷ Current root of the static tree
$parent$ ▷ ID of parent process
$children$ ▷ Child processes of the node
$idle$ ▷ Boolean stating whether the process is idle or busy
$free$ ▷ Boolean stating whether the process is free or loaded
$inactive$ ▷ Boolean stating whether the process is part of the active tree or not
KC ▷ Number of processes that have crashed
$child\_inactive[\,]$ ▷ Active status of children, 1 means not in active tree
$child\_counts[\,]$ ▷ Child subtree sizes
$num\_unack\_msgs[\,]$ ▷ List, number of unacknowledged basic messages sent to each process
$Crashed$ ▷ Set of processes whose crash notifications have been received
$owned$ ▷ Boolean stating whether the process has to send an acknowledgement or not
$ownedBy[\,]$ ▷ Processes that have yet to receive an acknowledgement for a basic message
$rAckNext$ ▷ Processes that have yet to receive an acknowledgement for a resume message

function RESET
    $free \leftarrow 0$
    $inactive \leftarrow 0$
    for all $i \in Children$ **do**
        $child\_inactive[i] \leftarrow 0$
        $child\_counts[i] \leftarrow 0$
    end for
    for all $i \in \{1, \ldots, nnodes\}$ **do**
        $num\_unack\_msgs[i] \leftarrow 0$
    end for
    $owned \leftarrow 0$
    $ownedBy \leftarrow [\,]$
end function
function UPDATE_KC(otherKC)
    if $otherKC > KC$ **then**
        for all $i \in ownedBy$ **do** ▷ Acknowledge all basic messages received
            SEND_MESSAGE(M_ACK, i)
        end for
        RESET ▷ All nodes are reset
    end if
end function
function SEND_MESSAGE(m, $target$) ▷ Is run when a message is being sent
    if m is basic message **then**
        $num\_unack\_msgs \leftarrow num\_unack\_msgs + 1$
    else if m is STOP **then**
        $count \leftarrow 1 + \sum_{i \in children} child\_counts[i]$
        append $count$ to m
    end if
    Send m, KC to $target$ ▷ KC is sent with each message
end function
function RECEIVE_MESSAGE($sender$, m, $message\_KC$) ▷ Is run when a message is received
    UPDATE_KC($message\_KC$)

```
if m is basic message then
    idle ←False
    free ←False
    if inactive then
        SEND_MESSAGE(RESUME, parent)
        inactive ←False
        owned ←True
        ownedBy ← ownedBy ∪ {sender}
    else
        SEND_MESSAGE(ACKNOWLEDGE, sender)
    end if
else if m is M_ACK then
    num_unack_msgs[sender] ← num_unack_msgs[sender] − 1
else if m is RESUME then
    if message_KC ≠ KC then
        return
    end if
    child_inactive[sender] ← child_inactive[sender] − 1
    if inactive then
        inactive ←False
        SEND_MESSAGE(RESUME, parent)
        rAckNext.append(sender)
    else
        SEND_MESSAGE(R_ACK, sender)
    end if
else if m is R_ACK then
    if message_KC ≠ KC then
        return
    end if
    if owned then
        for p ∈ ownedBy do
            SEND_MESSAGE(M_ACK, p)
        end for
        owned ←False
        ownedBy ← [ ]
    end if
    for p ∈ rAckNext do
        SEND_MESSAGE(R_ACK, p)
    end for
    rAckNext ← [ ]
else if m is STOP then
    if message_KC ≠ KC then
        return
    end if
    if COUNT(m)> child_counts[i] then
        child_counts[i] ← COUNT(m)
    end if
else if m is DCONN then
    children ← children \ {sender}
```

```
    else if m is CONN then
        children ← children ∪ {sender}
        inactive ←False
    end if
end function
function CRASH_NOTIFICATION(p)          ▷ Process is notified that process p has crashed
    Crashed ← Crashed ∪ {p}
    block channel from p
    if p ∈ children then
        children ← children \ {p}
        rAckNext ← rAckNext \ {p}
    end if
    num_unack_msgs[p] ← 0
    if p ∈ ownedBy then
        ownedBy ← ownedBy \ {p}
    end if
    UPDATE_KC(Size(Crashed))
    if p = root then
        NEWROOT(root)                                  ▷ Computes new root of the tree
        if root ∈ children then                        ▷ New root is a child of this process
            children ← children \ {root}
        end if
    end if
    if p = parent∧ID≠ root then
        parent ← root
        SEND_MESSAGE(CONN, root)
    end if
end function
function TURN_FREE          ▷ Called when process is idle and ∑_i num_unack_msgs = 0
    free ←True
end function
function TURN_INACTIVE                  ▷ Called when free =True, inactive =False,
∀i ∈ children : child_inactive = 1 and KC = Size(Crashed)
    inactive ←True
    if ID≠ root then
        SEND_MESSAGE(Stop, parent)
    end if
end function
function DECLARE_TERMINATION            ▷ Called by the root when inactive =True and
1 + ∑_{i∈children} child_counts[i] = nnodes− Size(Crashed)
    TERMINATION
end function
```

4 Simulation Environment

The simulation environment was originally written by G. Karlos [6] using the Ibis Java library to handle threads and Java's synchronized methods to handle correct variable access. His repository was forked and changed substantially, most notably with the addition of new termination

detection algorithms and additional input parameters. It is a multi-threaded environment where the basic algorithm of each process is run by a single thread. The input parameters that can be specified are which control algorithms to run, how many processes are in the network, the number of processes that crash, the level of activity, the maximum wait time, and the maximum number of basic messages. Writing to a comma-separated value file has not been implemented for all algorithms, only the ones written by G. Karlos.

Choosing an algorithm is done by specifying a number: 1, 2, and 3 are algorithms implemented by G. Karlos, they are Safra's termination detection algorithm [4], an improved version of that algorithm, and a fault-tolerant version based on the improved version, respectively. Algorithms 4 and 5 are the Mahapatra-Dutt static tree termination detection algorithm [10] and our fault-tolerant version of this algorithm, respectively. Finally, algorithms 6 and 7 are the LTD variant of the Dijkstra-Scholten [7] termination detection algorithm modified to only handle diffusing computations and the Lai-Wu fault-tolerant termination detection algorithm [8] based on the LTD variant of the Dijkstra-Scholten algorithm.

We chose to implement the Lai-Wu algorithm to compare it with our fault-tolerant termination detection algorithm. Since we were implementing the Lai-Wu algorithm, it made sense to also implement the fault-sensitive algorithm they based their fault-tolerant termination detection algorithm on. The LTD variant of the Dijsktra-Scholten algorithm was therefore also implemented. Both the Dijkstra-Scholten and the Lai-Wu termination detection algorithm implementations only work on diffusing computations. This is only a slight change to the algorithm and should not affect the performance of the algorithms negatively during the simulations.

The simulation environment has many random length delays used to simulate either network latency or a process working. All random numbers are drawn from a uniform random distribution using the `java.util.Random` library.

The number of processes is an integer in the range $4-1024$ but is also limited on the hardware that it is run on as well as which algorithm is run. The number of processes simulated might depend on the hardware, as each thread needs a certain amount of memory to operate. Modern computers should however not have any problem simulating 1024 processes. The 1024 maximum is only an artificial limit set on the input parameter and changing this would not be difficult.

The number of processes that crash is an integer greater than or equal to zero but lower than the number of processes. The simulation chooses which processes to crash uniformly random one after another. The processes chosen are then crashed in the order they were chosen with a random delay in between the crashes. The random delay is drawn from the range of 0 to 2000 ms. After each crash has occurred, all non-crashed processes are notified of the crash within 5 seconds of the crash happening. This delay is randomly chosen per processes being notified on the range of 0 to 5000 ms.

The level of activity is an integer in the range $2-4$ and determines how many basic messages every active process sends. When a process becomes active, it randomly chooses a number, $nActivity$, below or equal to the level of activity chosen. The process then performs $nActivity+1$ actions. Each action consists of a random sleep period of up to 1000 ms and sending messages to a number of processes up to the level of activity. This means that with level of activity ℓ, the number of messages sent by a process when it becomes active is up to about ℓ^2. What this means in practice is that with a level of activity of 2 the computation usually dies out quite quickly, while level 3 is not likely to stop unless the number of processes is very small. When the activity has been simulated the process becomes passive and becomes active again only when a basic message is received.

This is where the maximum number of basic messages option comes in handy. This parameter has one special input value which is -1. If the simulation environment is given this value, there is no limit set on the maximum number of messages. Any non-negative value given will be the

limit on the number of basic messages allowed to be sent. This limit is not a hard limit, and there might be one or two extra messages sent. The reason for this is that two or more processes might check to see if the limit has been reached at the same time when the limit has almost been reached. If the processes subsequently send messages this might take the number of messages over the limit.

Lastly, we have the maximum wait time, which is given in milliseconds. This is a time out mechanism which stops all threads when the time has been reached. This is important if there is an algorithm that is not able to declare termination or if something went wrong in the run. This is a very important tool during development of the algorithm to find bugs or flaws in the algorithm, because it allows for automation of running the algorithm many times without having to worry about runs possibly never ending. This might be the case if there are any deadlocks present in the implementation or if the implementation contains some other types of errors.

Processes send messages to each other through methods in a network object, which in turn calls a receive method in the receiving process. At the time of writing, both the basic algorithm and control algorithm have to be written for each control algorithm. The same basic algorithm is used for each of the control algorithms, but since they are written in the same class, the basic algorithm has been copied for each of the algorithms. There is also a nodeCrasher class that decides which processes to crash and then calls a crash method in the appropriate process object and notifies the other processes of the crash as well. Running on top of all this is the simulator which initiates all of the objects and starts the simulation; this is also where the timeout limit is set. If the object has not received a notification of termination within the time limit, it stops the processes and prints a warning stating that termination was not declared.

The detection delay is measured by the network object, each time a process becomes passive, it sends a message to the network that logs the time. When termination is declared, the detection delay is calculated to be the time of declaration minus the time at which the last process turned passive. Similarly, the number of basic messages as well as the number of control messages are recorded in the network class and printed when termination has been declared.

For the static tree algorithm, there needs to be some way of determining the tree structure before the algorithm starts. The way we have done it is by using an echo algorithm to create the tree each time a simulation is run. The algorithm starts at the root and sends a message to all the processes in the network in random order. When a process receives the message for the first time, it sets the sender to be its parent and starts sending to other processes to see if they want to be its children. All messages received after the first time are acknowledged right away with a reject message. When a process has received either a reject message or an accept message from all processes, it sends an accept message to its parent. When the root has received a reject message or an accept message from all processes, it signals that the basic algorithm can be started.

The simulation environment is a prototype and needs much work to be able to be used effectively in developing distributed control algorithms. More on how this could be done can be found in section 6.

The code for the simulation environment and all the algorithms implemented can be found at `https://github.com/KristinnArdal/TDS`.

5 Results

5.1 Simulations

Two sets of simulations were made on the algorithms. The first set of simulations was made on all algorithms. All simulations use the activity level of 3, as this makes it very likely that

the maximum number of basic messages set will be reached. These simulations were used to see the relation between number of processes in the network with both the detection delay and the message complexity. The number of processes in the system varied between 128 and 1024 processes, doubling the number of processes in each step. In these simulations the number of basic messages was kept the same. The number of messages was chosen such that using only 128 processes the simulation would not take too much time and for 1024 processes all processes would most likely be active at some point during the simulation. With these conditions in mind, the number of basic messages chosen was 10 000. This way the average number of messages sent by a process is 78 and 10 for 128 and 1024 processes respectively.

The second set of simulations was made on the three fault-tolerant algorithms. These simulations introduced crashes into the system. The number of processes in the network used for these simulations was set to 512. The reason for this is that we wanted to use one of the previously used process counts to be able to compare with the simulations without crashes. Additionally we wanted the simulations to run somewhat quickly, and more processes in the network means that it takes less time to send the required basic messages. The number of processes that we crashed varied from 2 to 64, doubling the number each step. It is worth noting that crashing n processes takes on average n seconds. Notifying all other processes of the crashes might take an additional 5 seconds. All algorithms require that all processes have been notified of all crashes that occur before being able to declare termination. This means that the simulations would at minimum take around $n+5$ seconds. Before introducing crashes, a simulation with 512 processes and 10 000 messages took on average slightly less than 30 seconds. What we expect to see then is an increase in detection delay for 32 and 64 crashes.

5.2 Safra's algorithm

Let us first look at the three algorithms that had already been implemented by G. Karlos. These algorithms are token algorithms, and the number of control messages is not directly related to the number of basic messages. We first look at simulations without crashes introduced. In figure 2 we can see how the number of control messages depends on the number of processes in the network. The fault-tolerant version of the algorithm is based on the improved Safra's algorithm and the number of control messages is approximately the same. The number of control messages is approximately linear with regard to network size for all three of the algorithms.

The improved Safra's algorithm improves the detection delay of the algorithm, as can be seen in figure 3. Here we see that the detection delay of the improved Safra's algorithm and the fault-tolerant Safra's algorithm are pretty much the same. As with the number of control messages, this is because the fault-tolerant version is based on the improved Safra's algorithm. We also see that the detection delay linearly increases with the number of processes in the network. This was to be expected, as the token has to visit more processes to gather the required information for declaring termination. Sending the token around the network takes an amount of time that is linear with respect to the size of the network.

Now let us introduce crashes to the system and see how the detection delay and the number of messages depend on the number of crashed processes. We only introduce crashes for the fault-tolerant Safra's algorithm and we have 512 processes as stated before. As previously stated, we expect an increase in detection delay for both 32 and 64 crashes. We however did not make any prediction on the number of control messages. Looking at figure 4 we see an increase in control messages starting at 32 crashes. The increase in control messages is even more dramatic when we have 64 process crashes. We want to also see how the number of control messages changes when the crashes do not happen at the end of the basic computation. We do this simply by restricting our view to only 16 or fewer crashes. The number of control messages decreases

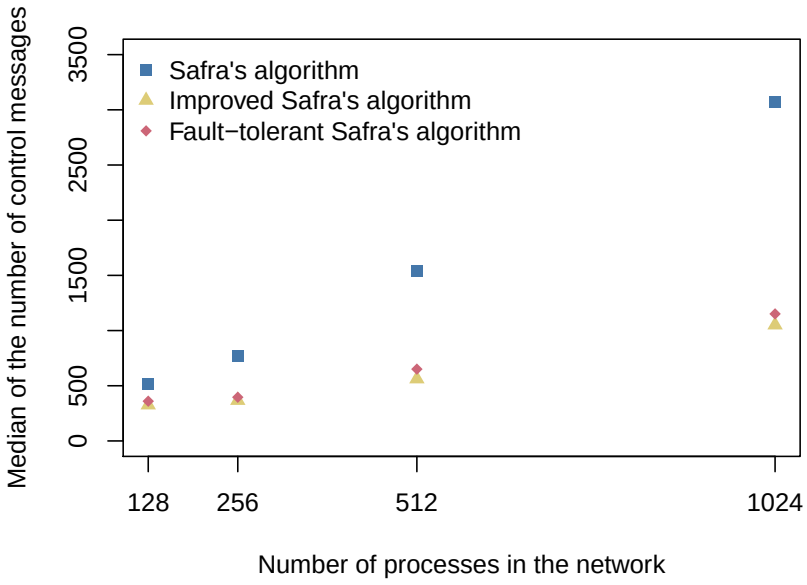


Figure 2: The number of control messages with respect to the number of processes in the network.

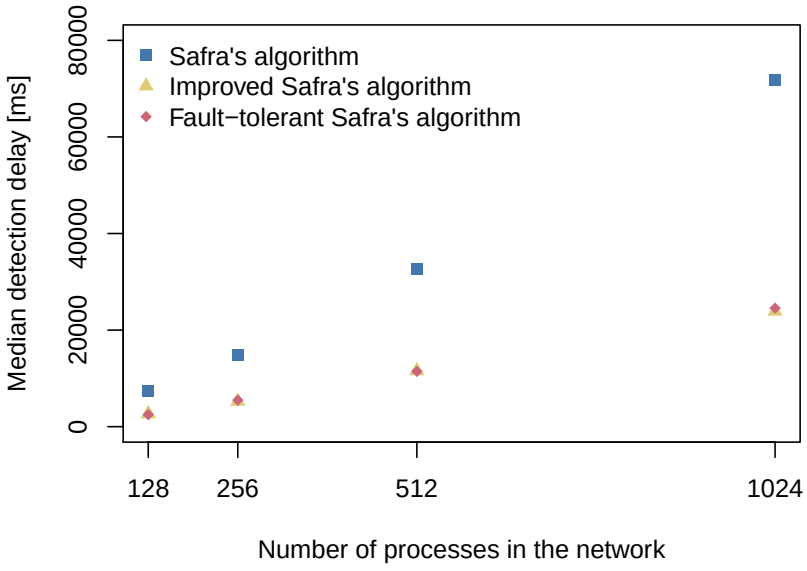


Figure 3: Detection delay with respect to the number of processes in the network.

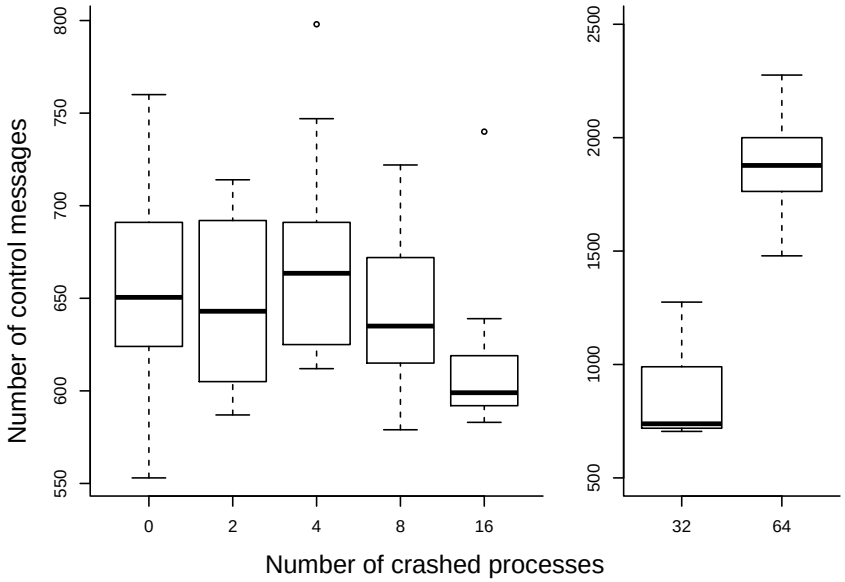


Figure 4: The number of control messages of fault-tolerant Safra's algorithm with respect to number of crashed processes in the network. The first part has crashes occurring before the basic computation has terminated, while the second part has crashes occurring after the basic computation has terminated.

slightly with the number of crashed processes. This is in line with what we saw in figure 2; when process crashes occur, the number of processes in the system decreases and therefore the number of control messages decreases as well.

The detection delay of the algorithm with respect to the number of crashes can be seen in figure 5. As we suspected, a huge increase in detection delay happens at 32 and 64 process crashes. We would also like to see what happens to the detection delay when all crashes happen before the end of the basic computation. We do this by simply limiting our view to the first part of figure 5. We can see that the crashes have hardly any effect on the detection delay when they happen before the end of the basic computation. The detection delay stays pretty much constant. We see a slight increase, but this increase is such a small part of the detection delay.

5.3 Static tree algorithm

For the static tree algorithm we have both the fault-sensitive version and the fault-tolerant version. The fault-tolerant version is a fault-reactive algorithm, meaning that we should see an approximately identical number of control messages when there are no crashes. In figure 6 we see that the number of control messages increases with the size of the network. This is to be expected as the tree structure grows, since a RESUME messages has to travel farther up the tree on average. The increase in control messages is approximately linear with the number of processes. The detection delay is around the same for both algorithms and can be seen for the fault-tolerant version in figure 7. The detection delay of this algorithm is totally different from what we saw in Safra's algorithm. The detection delay is only around 100 ms for 128 and 256 processes but jumps to around 125 ms for 512 and 1024 processes. Sending a message takes on

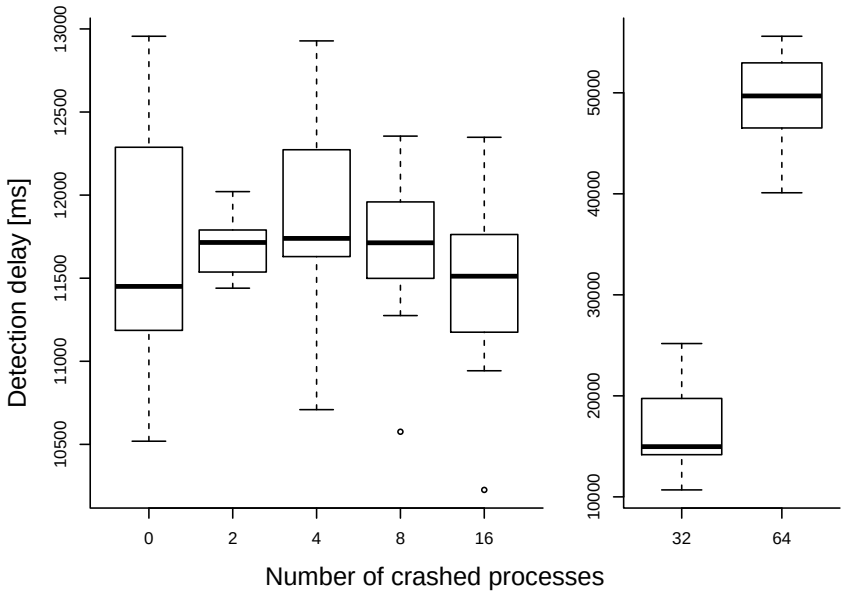


Figure 5: Detection delay of fault-tolerant Safra's algorithm with respect to number of crashed processes in the network. The first part has crashes occurring before the basic computation has terminated, while the second part has crashes occurring after the basic computation has terminated.

average 25 ms, which means that on average 4 sequential control messages need to be sent after the last process has turned passive for the control algorithm to declare termination. This goes up to around 5 sequential control messages when the network has 512 or 1024 processes respectively. This increase comes from the structure of the static tree: increasing the number of nodes, the average depth of the static tree increases; this increase leads to an extra control message at the end of the simulation.

Now let us introduce crashes into the system. As before we expect an increase in detection delay at around 32 process crashes. We see an increase in both control messages and detection delay at exactly this number of crashed processes in figures 8 and 9 respectively. The figures are split in two parts, fewer than or equal to 16 process crashes and more than 16 process crashes, because of the dramatic increase in both figures. We want to see the effect of process crashes that happen during the basic computation on the number of control messages and detection delay. We therefore limit our view to only 16 process crashes or fewer. We see that when we introduce process crashes, the distribution for the number of control messages starts to spread out. Additionally the number of control messages decreases. The reason for the decrease in control messages is threefold: acknowledgement messages can be disregarded because of a crash, processes might stay in the active tree for a longer time when a crash occurs, and finally the tree structure becomes flatter and flatter every time a process crashes.

Processes staying in the active tree for a longer time results in a reduction in Resume messages. The reason a process might stay longer in the active tree when a crash occurs is that it might hear from another process that a process has crashed, in which case it will not leave the active tree until all of its children and itself have been notified of the same number of process crashes. The decrease in number of control messages is quite significant, or about 10% in total

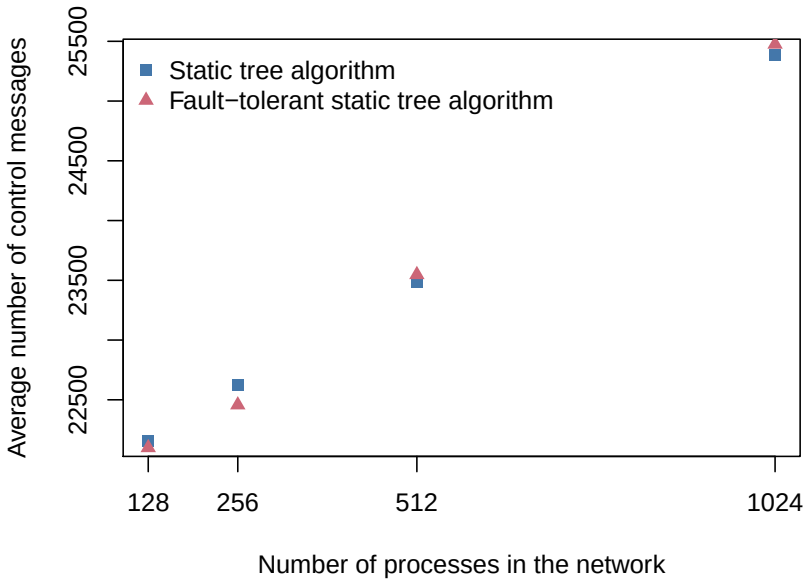


Figure 6: The number of control messages with respect to the number of processes in the network.

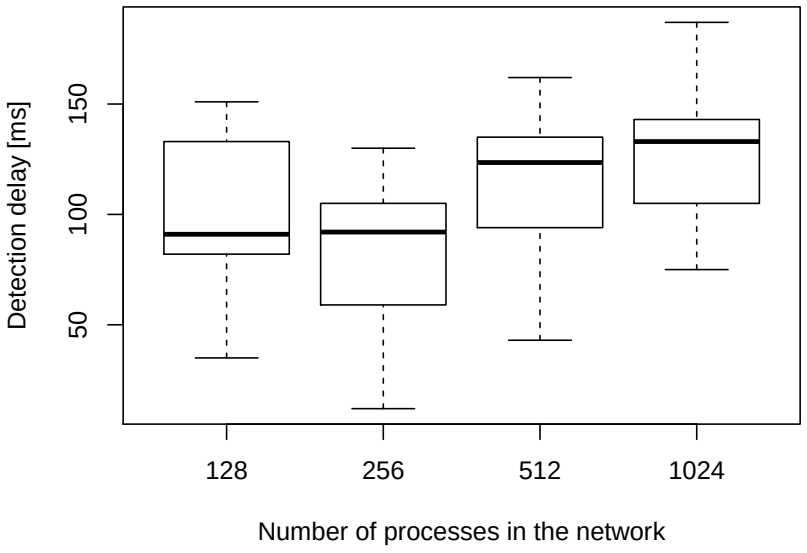


Figure 7: Detection delay with respect to the number of processes in the network.

when we have 16 process crashes. The simulation has processes crash one after another with a random delay in between. Notifying all the processes of the crash however takes more time than the delay in between crashes. Changing this behavior in the simulation by making the time between crashes longer on average than the time it takes to notify all other nodes might change this pattern entirely. Then we might see an increase in the number of control messages.

Having a flat tree or shallow tree structure means that there will be fewer control messages, as joining the tree might take fewer RESUME messages. This however also makes that a few processes have to do most of the work of the control algorithm. For this reason we do not want the tree to be very flat. When a process crashes all of its children reattach to the root process making the tree become flatter. However, in case of few process crashes this flattening effect on the tree will be very limited.

We now look at the effect that process crashes during the basic computation have on the detection delay. As before we limit our view to only 16 process crashes or fewer. In the first part of figure 9 we see that when we introduce process crashes, the detection delay decreases. We have already discussed that process crashes make the tree structure become flatter. The detection delay is directly related to the depth of the active tree when termination occurs. For this reason the detection delay decreases with the introduction of crashes. We should see the detection delay become lower and lower with the number of process crashes but this does not happen. The reason for this might be that the root has more control messages to receive and the increase from this is what we see. Another reason might just be that more simulations are needed to get a better approximation of the detection delay. This reason however seems unlikely, given the fact that we would expect a steady decrease with the number of process crashes, but we see the inverse relation. Unlike the number of control messages, the detection delay should not be affected by how the crashes are simulated, given that they occur and are handled before the basic computation has terminated.

5.4 Lai and Wu algorithm

We now look at the last two algorithms, the LTD variant of the Dijkstra-Scholten termination detection algorithm and the Lai-Wu termination detection algorithm. The Lai-Wu algorithm is a fault-reactive variant of the Dijkstra-Scholten algorithm based on the LTD variant. This means that the number of control messages for the algorithms should be identical in a system where there are no crashes. Looking at figure 10, we can see that this is the case.

The number of control messages approaches 10 000 when the number of processes increases. Without crashes, the maximum number of control messages for the algorithms is equal to the number of basic messages, and in these simulations we had 10 000 basic messages. For the original Dijsktra-Scholten algorithm the number of control messages would be exactly equal to the number of basic messages. The LTD variant is an optimization on the number of control messages. The reason we see an increase in the number of control messages with the number of processes is how the LTD variant reduces the number of control messages. The algorithm works by having each basic message require an acknowledgement message. The LTD variant makes it possible for a process to acknowledge multiple basic messages from the same process in one message. The simulation chooses a process at random to send a basic message to. Increasing the number of processes makes it less likely that the same process will be chosen more than once in a short amount of time.

The detection delays for the algorithms are almost identical, and we can see the detection delay of Lai and Wu's algorithm in figure 11. The algorithm has an average detection delay of around 275 to 300 ms for 128 to 1024 processes. This means that there are on average 11 to 12 sequential control messages needed after termination has occurred for termination to be

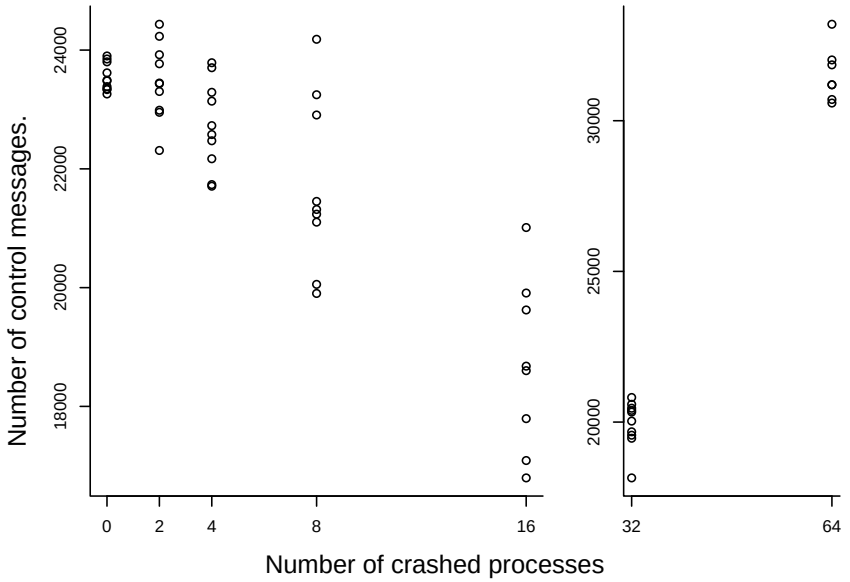


Figure 8: The number of control messages of the fault-tolerant static tree algorithm with respect to the number of crashes processes in the network. The first part has crashes occurring before the basic computation has terminated, while the second part has crashes occurring after the basic computation has terminated.

declared.

As with the other algorithms we now introduce crashes to the network. We expect the number of control messages to increase with the number of crashes, and as with the other algorithms we expect to see an increase in the detection delay when we have 32 or more process crashes. In figure 12 we see the number of control message in the network with crashes. There is a definite increase in control messages from the very start, but the increase becomes much more dramatic at 32 process crashes. For 32 and 64 process crashes, processes are crashing after the basic computation has terminated. Because of this processes often leave the active tree directly after they are notified of a crash. We would therefore expect the number of control messages to increase linearly when we have more than 32 processes. When we look only at process crashes that happened during the basic computation, this however looks slightly different. Looking at only 16 or fewer process crashes, the increase is not really linear; there is a small spike in the number of control messages when we have only 2 process crashes, but for $4, 8, 16$ process crashes it becomes almost linear. The reason for this might be that the first crashes create more control messages than the next ones. The simulation environment is on average sending notifications for 5 process crashes at any given time. The way Lai and Wu handle process crashes is that they have every process send their set of detected process crashes with their acknowledgement messages. Only when a process has been notified of all crashes it has seen from other processes, does it leave the active tree. If notifications for multiple processes are going around, this might make the processes having to wait. Having process crashes being handled before the next process crashes might give us more insight into how this works.

If we now look at the detection delay, we can see in figure 13 a large increase in detection delay at 32 process crashes. Here we have the same reason as we had for the other processes; process

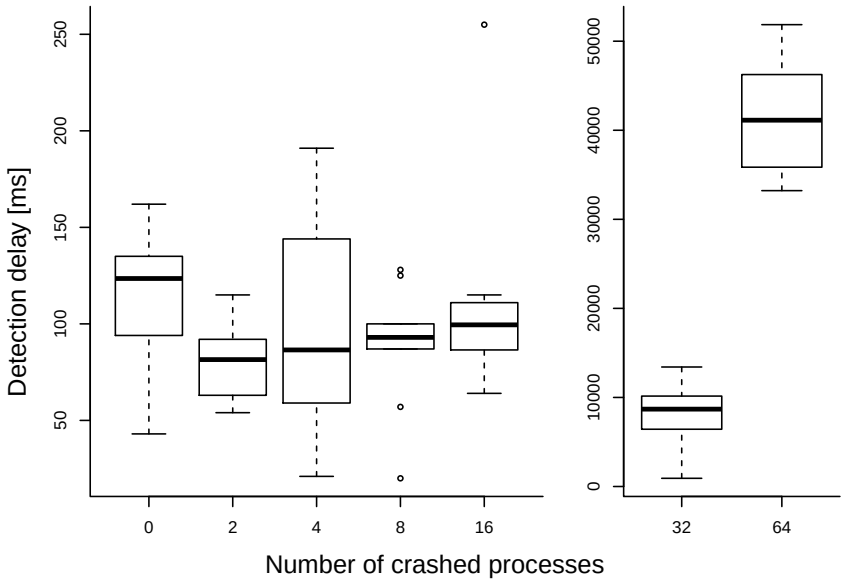


Figure 9: Detection delay of fault-tolerant static tree algorithm with respect to the number of crashed processes in the network. The first part has crashes occurring before the basic computation has terminated, while the second part has crashes occurring after the basic computation has terminated.

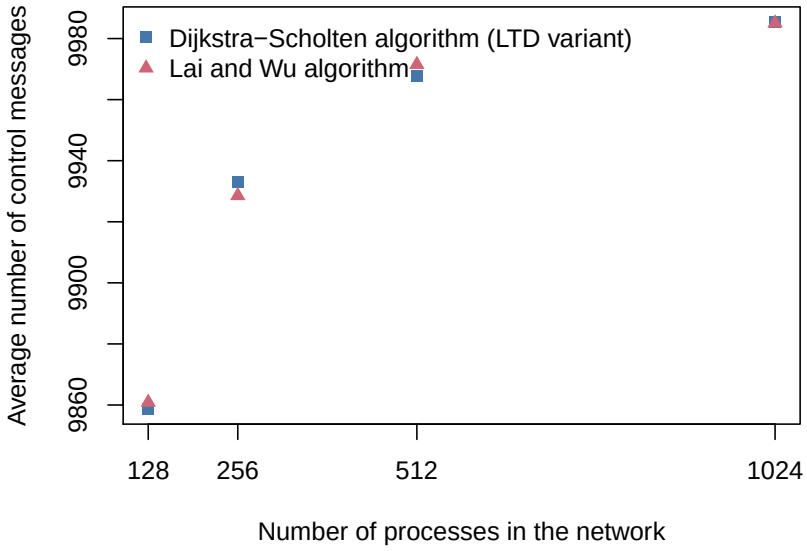


Figure 10: The number of control messages with respect to the number of processes in the network.

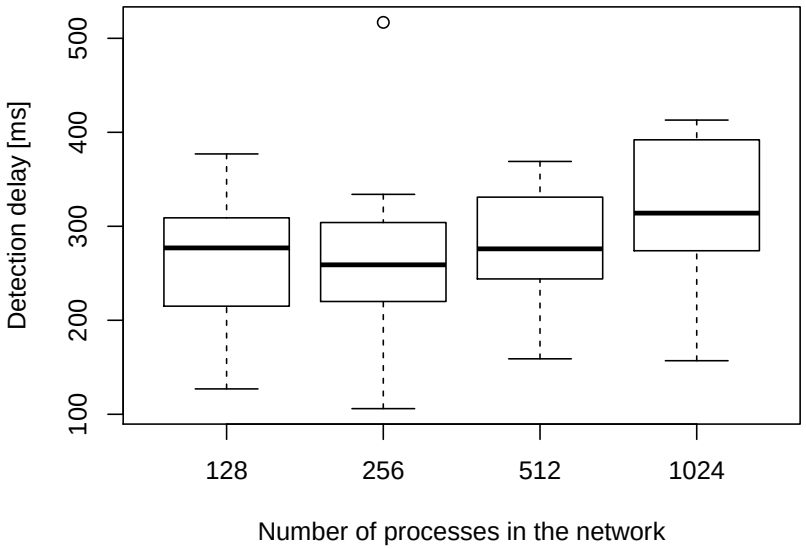


Figure 11: Detection delay with respect to the number of processes in the network.

crashes are happening at the end of the simulation, and all processes have to be notified of the crashes before the control algorithm can declare termination. We now look at only crashes that happened during the basic computation; we limit our view to only 16 or fewer crashes. Here we see a very strange effect: the detection delay decreases dramatically when crashes are introduced; the detection delay is cut in half when we have crashes. This might seem counter-intuitive, but the reason for this comes from the way the tree is built. When the algorithm starts, the active tree grows quickly with each basic message and becomes quite deep. When a process crashes, the tree is flattened. The tree does not become as deep as it was before, since so many processes are already a part of the active tree. The more time the algorithm has after the last crash, the more likely it is to grow bigger. This can be seen in the graph, as for 2 and 4 process crashes the detection delay is slightly larger than for 8 and 16 process crashes.

This result suggests that the Lai-Wu algorithm is more suited for non-diffused computations as the tree is less likely to grow as deep.

6 Discussion

There are two parts of this thesis that need to be discussed: the fault-tolerant version of the algorithm and the simulation environment. We start with discussing the fault-tolerant version of the algorithm that we developed.

6.1 Fault-tolerant algorithm implementation

The fault-tolerant version of the algorithm is very similar to the Lai-Wu algorithm, with the exception being how reattaching to the tree is handled. We could do a more similar thing to what Lai and Wu do, which is to make all processes report to the root either saying they want to

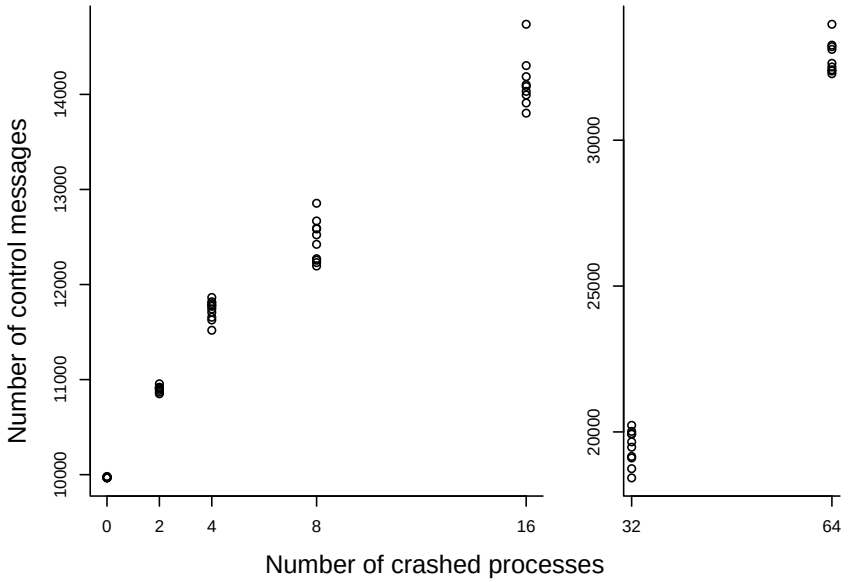


Figure 12: The number of control messages of Lai and Wu's algorithm with respect to the number of crashed processes in the network. The first part has crashes occurring before the basic computation has terminated, while the second part has crashes occurring after the basic computation has terminated.

reattach or not. We however want to avoid the situation where the root becomes a bottleneck, which is exactly why we have chosen to reset the active tree when a process crashes and let processes report on the tree size with the STOP messages rather than report directly to the root saying it attached to the tree. In the worst case the number of control messages sent because of the crash is $\mathcal{O}(n + c)$ where n is the number of processes in the system and c is the number of children that need to be reattached to the tree. This is slightly larger than Lai and Wu's $\mathcal{O}(n)$, but these messages are spread out such that each process only gets messages equal to the fan-out of the process after recovery. We are however not likely to hit the maximum number of control messages in our algorithm, as that would mean that the crash occurs very close to the time of termination. Additionally some control messages can be skipped when a process crashes, reducing the cost further. It is also worth noting that the reason we can restart the algorithm efficiently is that the tree is static. Every process is already a part of the static tree, they just have to join the active tree, which is important for termination detection. Since the active tree has the same structure as the static tree, this can be done locally at each process. This approach is very similar to what Lai and Wu do, they have every process join the active tree but they flatten the tree to only be two layers.

Reattachment is done to the root but could be done to any other process that is not in the sub-tree of the process. This is however a bit more tricky, as processes need to have more knowledge of the tree structure. This could however be done before the run, since the static tree is usually decided upon before the run as well. Each process could for example keep track of their entire parental chain up to the root and choose the first ancestor that has not crashed to attach to. If all of the ancesters have crashed, reattaching to the root is possible. This should keep the tree more hierarchical, as always attaching to the root can make the tree flat very quickly. This

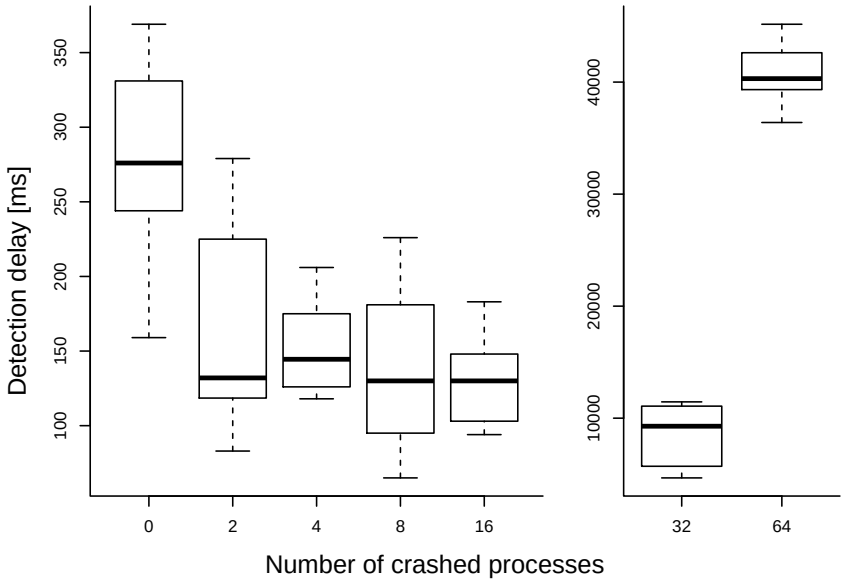


Figure 13: Detection delay of Lai and Wu's algorithm with respect to the number of crashed processes in the network. The first part has crashes occurring before the basic computation has terminated, while the second part has crashes occurring after the basic computation has terminated.

also increases the number of control messages incurred by a crash to $\mathcal{O}(n + c \cdot d)$ where d is the depth of the tree. This is because reattaching to the tree at a process that is not the root might lead to a RESUME message being sent all the way to the root after a STOP message has already been sent the whole way. We chose to attach to the root as it is the simplest to implement. This however means that when we have many crashes, the tree structure gets flatter quickly, making the root have to do more and more work.

In the simulation environment we are measuring the detection delay and message complexity of the algorithms. To get the best detection delay possible for the static tree algorithm, one could choose a tree where the root process is the parent to all other processes. This however would increase the work load of the root process significantly. We did not measure the load distribution per process. This could be done by simply calculating the number of messages being sent to and from each process. This would give us a very low-level representation of the load distribution. Another way could be to use a logical clock to measure the longest sequential chain of send and receive events. Using either of these measures, one could measure the difference in load distribution the static tree structure has in the algorithm. One could also examine what effect different reattachment policies have on the system.

6.2 Improvements on the simulation environment

The simulation environment needs a total re-design for it to become more usable. The current version has the programmer do too much for each control algorithm. There needs to be consistency in the network and basic computation between algorithms so that they can be accurately compared with one another. Currently the basic computation has to be written for each algo-

rithm, meaning that if one were to write a new control algorithm, the basic algorithm could be made completely different from how it is in the other algorithms. The basic computation should be made completely separate from the control algorithm. The control algorithm could have hooks into the send and receive events of the basic computation as well as having its own interface for sending and receiving control messages. This could be done using an abstract class that defines this interface for all control algorithms.

This would mean that all control algorithm could share the same classes for the simulation start up, network and basic algorithm. Changes in any of these algorithms would then be shared among the control algorithms. After this has been done, changes to the network or basic algorithm become much easier to implement and test. The rest of the discussion is on changes that would be much simpler to implement after the code had been re-designed.

Having done this it would be no problem having different basic computation models and being able to change between them using an input parameter. The way it is implemented now makes it very difficult to change the basic computation, since only changing it for one algorithm means that comparisons with other algorithms performance become invalid. For it to be valid every algorithm would have to have the same basic algorithm.

The network latency in the simulation environment is random for each message with a maximum upper bound. A better way would be to give each connection a latency distribution function, which could be determined when initiating the network. When sending a message, the message delay could then be drawn from the distribution. This takes more space, as each connection needs to be designated a distribution and there are n^2 connections, but would make for a more realistic network latency model.

The way crashes are handled is that the processes that are to crash are decided before the control algorithm starts, and a simulated crash of those nodes is initiated after a random amount of time. The random amount of time is drawn from a uniform distribution of the range $0 - 2\,000$ ms. The upper limit of this time has been chosen arbitrarily and does not depend on the number of processes in the network, how many processes will crash, or the level of activity. The reason for this short time interval is that we do not know how long the computation will last and therefore the processes are crashed sooner rather than later. A better approach would be to have it depend on some of the input variables. One could for example model the computation time of the algorithm with respect to the number of processes in the network, level of activity, and maximum number of basic messages. This model could then be used to calculate the expected run time of the algorithm, and each process that should crash is crashed within the expected end of the computation. The model for this would then also depend on the basic algorithm, if the option for multiple basic algorithms has been implemented.

7 Conclusion

We created a fault-tolerant variant of the Mahapatra-Dutt static tree termination detection algorithm. We implemented it in a simulation environment and ran simulations using said environment. Implementing the algorithms helped a lot in understanding the issues that we faced when making the algorithm fault-tolerant. We were able to test out ideas very quickly and see what their advantages and disadvantages are. The resulting algorithm is an $(n-1)$-fault-tolerant termination detection algorithm.

We implemented the fault-sensitive static tree termination detection algorithm as well as the fault-tolerant variant. In addition to that, G. Karlos had already implemented Safra's algorithm, an improved version of Safra's termination detection algorithm and a fault-tolerant variant based on the improved version. We wanted to have another tree algorithm to compare with our algo-

rithm. We therefore implemented Lai and Wu's termination detection algorithm as well as the algorithm they based their algorithm on, the LTD variant of the Dijkstra-Scholten termination detection algorithm.

We ran two sets of simulations on all the algorithms; one where there were no process crashes and the number of processes in the network was varied and one where the number of process crashes was varied, keeping the number of processes in the network constant. The first set of simulations was done on all algorithms, while the second set was only done on the fault-tolerant algorithms.

The simulations show us that Safra's algorithm has very few control messages when compared to the other algorithms, but has a high detection delay. The improved and fault-tolerant versions of Safra's algorithm lower both the number of control messages and detection delay. The detection delay is however still quite high when compared with the other algorithms.

The two static tree algorithms both have the highest number of control messages but the number actually decreases when we introduce crashes. The detection delay of both algorithms was very low, even when compared to the Lai-Wu algorithm. The detection delay reduced slightly when crashes were introduced to the network.

The number of control messages for the Lai-Wu algorithm was much lower than for the static tree algorithms, but when crashes were introduced the number of control messages increased. The detection delay was not very high, but when crashes were introduced this actually decreased, as each time a crash occurs the tree structure is flattened.

All algorithms had increased control messages and detection delay when crashes occurred after the basic computation had terminated. This is because termination can not be declared until all processes have been notified of all crashes that have occurred.

The simulation environment has huge potential but is at the moment of writing not very user friendly. The reason for this is that the code needs an overhaul such that introducing new control algorithms or changing any of the features of either the basic algorithm or the network becomes simple. I can see this simulation environment or something similar to this becoming an important part in developing other distributed control algorithms or even adjusting distributed control algorithms to other basic computation models.

The overhaul on the simulation environment might make a good BSc project. Many of the ideas have already been outlined in section 6. This would hopefully attract more people to using the simulation environment when developing new algorithms.

Using the simulation environment was very advantageous while developing the fault-tolerant static tree algorithm. We were able to quickly test out ideas for the algorithm and see whether they worked or not. While this environment is advantageous for developing termination detection algorithms and testing them it is not able to prove that termination is always declared. The environment can be used to find flaws in the algorithm, but it was not designed to try every possible path of the execution and can therefore not be used to prove termination. There are other techniques available for exactly that purpose, notably model checking [1] and theorem proving [2], which should be used instead of this one if proving correctness is desired.

References

[1] Christel Baier, Joost-Pieter Katoen, and Kim Guldstrand Larsen. *Principles of Model Checking*. The MIT Press, Cambridge, Mass, April 2008.

[2] Jasmin Christian Blanchette and Stephan Merz, editors. *Interactive Theorem Proving*, volume 9807 of *Lecture Notes in Computer Science*. Springer International Publishing, Cham, 2016. DOI: 10.1007/978-3-319-43144-4.

[3] Edsger W. Dijkstra and Carel S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, 1980.

[4] W. H. J. Feijen and A. J. M. van Gasteren. Shmuel Safra's Termination Detection Algorithm. In W. H. J. Feijen and A. J. M. van Gasteren, editors, *On a Method of Multiprogramming*, pages 313–332. Springer New York, New York, NY, 1999. DOI: 10.1007/978-1-4757-3126-2_29.

[5] Nissim Francez. Distributed Termination. *ACM Trans. Program. Lang. Syst.*, 2(1):42–55, January 1980.

[6] George Karlos. *A fault-tolerant variant of Safra's termination detection algorithm.* BSc thesis, Vrije Universiteit, Amsterdam, 2016.

[7] Ten-Hwang Lai, Yu-Chee Tseng, and Xuefeng Dong. A more efficient message-optimal algorithm for distributed termination detection. In *Parallel Processing Symposium, 1992. Proceedings., Sixth International*, pages 646–649. IEEE, 1992.

[8] Ten-Hwang Lai and Li-Fen Wu. An (N-1)-resilient algorithm for distributed termination detection. *IEEE Transactions on Parallel and Distributed Systems*, 6(1):63–78, 1995.

[9] Jonathan Lifflander, Phil Miller, and Laxmikant Kale. Adoption Protocols for Fanout-optimal Fault-tolerant Termination Detection. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 13–22, New York, NY, USA, 2013. ACM.

[10] Nihar R. Mahapatra and Shantanu Dutt. An efficient delay-optimal distributed termination detection algorithm. *Journal of Parallel and Distributed Computing*, 67(10):1047–1066, October 2007.

[11] Jeff Matocha and Tracy Camp. A taxonomy of distributed termination detection algorithms. *Journal of Systems and Software*, 43(3):207–221, 1998.

[12] Jayadev Misra. Detecting Termination of Distributed Computations Using Markers. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, PODC '83, pages 290–294, New York, NY, USA, 1983. ACM.

[13] Neeraj Mittal, Felix C. Freiling, Subbarayan Venkatesan, and Lucia Draque Penso. On termination detection in crash-prone distributed systems with failure detectors. *Journal of Parallel and Distributed Computing*, 68(6):855–875, 2008.

[14] Kristinn Björgvin Árdal. *Fault-tolerant distributed termination detection algorithms.* Literature study, Vrije Universiteit, Amsterdam, 2017.